Synchronisation dans les systèmes distribués

Université Nice SOPHIA ANTIPOLIS

Motivation

- Un système distribué étant une collection de programmes indépendant, il est nécessaire de les faire coopérer
- Implique une synchronisation
  - Plusieurs processus se partagent une ressource
  - Un événement sur un processus P1 arrive à la suite d'un événement sur P2
- Plusieurs formes de synchronisation
  - Horloges physiques
  - Horloges logiques
  - Élections

1 - Horloges physiques

TAI et UTC

- La méthode la plus précise pour mesurer le temps est l'horloge atomique (1948)
- 1 seconde vaut 9192631770 transitions d'un atome de Cesium 133
- 50 horloges existent dans le monde et reportent périodiquement l'heure au Bureau International de l'Heure
- La moyenne donne l'International Atomic Time (TAI)
- Mais 86400 secondes TAI sont inférieures de 3ms à une journée solaire moyenne
  - Risque de décalage sur le long terme
  - Il faut introduire des sauts
- Cette correction donne naissance au Universal Coordinated Time (UTC) qui est la base de l'heure civile
- Le National Institute of Standard Time emet un signal onde courte à la fin de chaque seconde UTC
  - Précision de 10ms pour la radio
  - Utilisation de satellites (précision de 0.5 ms)

Horloges Physiques

- Pas de problèmes en centralisé, même en multiprocesseurs
- Si P1 demande l'heure, et qu'ensuite P2 demande l'heure aussi, alors il aura une valeur > à celle de P1
- Chaque ordinateur a un unique circuit qui gère l'écoulement du temps
  - Système à base de quartz
  - Tous les processus utilisent la même horloge
- Dans le cas d'un système distribué, on a plusieurs sources pour l'heure
  - Même très identiques, elles dérivent les uns par rapport aux autres
  - Les programmes utilisant le temps pour se synchroniser peuvent échouer

Problème de la synchronisation par horloge physique

- Certains outils se basent sur l'heure pour effectuer une opération
- Ex: Ant et Make décident de recompiler une classe si la date de la version compilée est antérieur à la version source
- Problème si les deux fichiers sont situés sur deux machines différentes

Synchronisation d'horloges

- Chaque machine P a un compteur de temps qui produit une interruption H fois/s et le décrémente
- Quand ce compteur arrive à 0, une horloge logicielle C est augmentée
- Quand le temps UTC vaut t, l'horloge a pour valeur Cp(t)
- Idéalement, Cp(t)=t pour tout t et tout P
- En pratique, il existe un léger décalage, appelé taux de dérive maximum et noté ϱ

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

- Cela signifie juste que lorsqu'une seconde UTC s'écoule, l'horloge de l'ordinateur a vu s'écouler $1 \pm \varrho$ secondes
- Si deux horloges sont décalées dans deux directions différentes, au bout d'un temps Δt, elles seront au plus décalées de $2\varrho\Delta t$
- Pour avoir un décalage maximal de δ il faut les resynchroniser toutes les $\delta/2\varrho$ secondes
- La plupart des algorithmes diffèrent juste dans la façon dont est fait cette resynchronisation

Algorithme de Cristian (1989)

- Une machine sert de serveur de temps, i.e les autres essaient de se synchroniser avec elle
- Périodiquement (pas plus que toutes les $\delta/2\varrho$ secondes), chaque machine demande l'heure au serveur
- Celui-ci répond aussi vite que possible avec l'heure CUTC
- L'appelant met alors son heure a CUTC
- Problèmes
  - Risque de retour en arrière de l'horloge
  - Latence

Algorithme de Cristian (1989)

- Éviter le retour en arrière
  - Incorporer le décalage graduellement
  - Ex: l'appelant a du retard et son horloge est a 100Hz, chaque interruption provoquera un changement de 11ms de l'horloge logicielle au lieu de 10ms
- Latence
  - Il faut l'estimer du mieux possible
  - Cela ne peut être fait que par l'appelant
  - Soit T0 heure d'émission et T1heure de réception de la réponse, la latence est de (T1–T0)/2
- Amélioration
  - Tenir compte du temps de traitement côté serveur
  - Éviter les congestions réseau

Algorithme de Berkeley (1989)

- Contrairement à l'algorithme de Cristian, le serveur est actif
- Régulièrement, le serveur demande l'heure à toutes les machines
  - Il effectue une moyenne
  - Il demande aux machines de ralentir/accélérer leur horloge pour arriver à la nouvelle heure

Synchronisation en moyenne

- Les deux algorithmes précédents sont centralisés
- Approche décentralisé
  - Périodiquement (intervalles décidés à l'avance), une machine broadcast son heure
  - Elle attend ensuite l'heure de toutes les autres
  - Les machines n'étant pas parfaitement synchronisées, les broadcast ne sont pas exactement en même temps
  - La nouvelle heure est calculée en fonction des heures reçues (attente d'un intervalle de temps max)
- Améliorations
  - Éliminer les m valeurs les plus hautes/basses
  - Estimer la latence

2 - Horloges logiques

Principe

- Dans beaucoup de situations, il n'est pas nécessaire d'avoir une bonne synchronisation avec le temps physique
- Pour certains programmes, seule la consistance interne des horloges compte
- On parle dans ce cas d'horloges logiques
- En 1978 Lamport a montré qu'une synchronisation absolue n'est pas nécessaire
  - Si deux processus n'interagissent pas, leurs horloges n'ont pas besoin d'être synchronisées
  - Les processus n'ont pas besoin d'être d'accord sur l'heure mais sur l'ordre des événements

Lamport

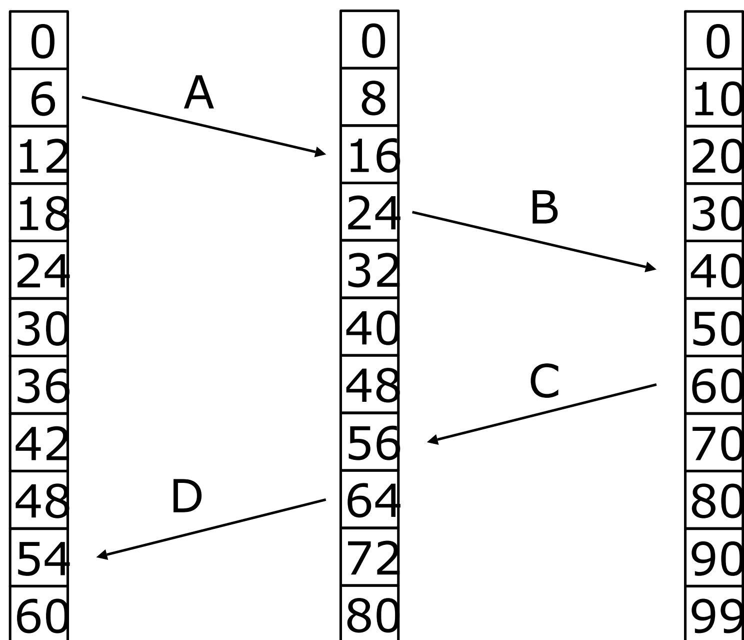
- On définit la relation « se produit avant » notée →
- Observable directement dans 2 situations
  - Si a et b sont dans le même processus est a arrive plus tôt que b, alors a → b est vrai
  - Si a correspond à l'envoie d'un message et b sa réception dans un autre processus alors a → b est vrai (un message ne peut être reçu avant d'être envoyé)
- Cette relation est transitive
- Dans le cas de 2 événements x et y se produisant sur 2 processus différents n'échangeant pas de messages
  - x → y n'est pas vrai mais y → x non plus
  - Ces événements sont dits concurrents

Lamport

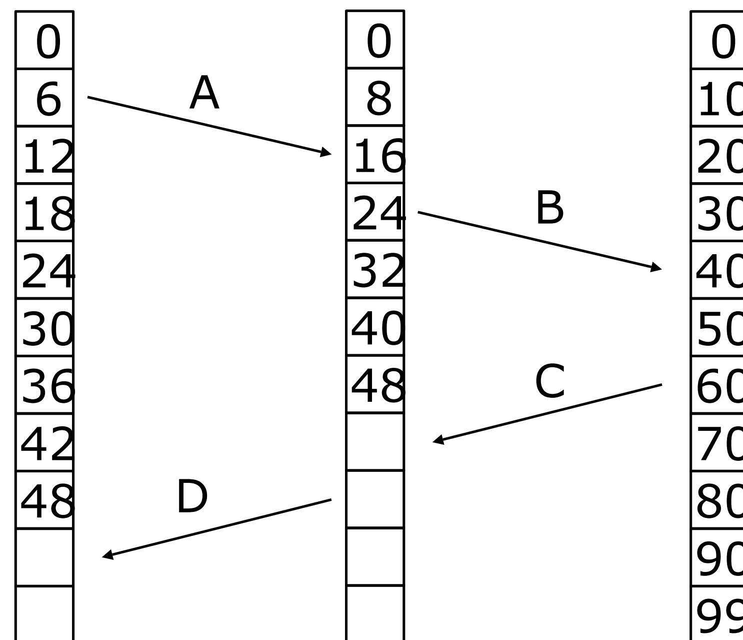
- Principe: on utilise un compteur qui ne peut que croître
  - Évite que des événements se produisent dans le passé
- Chaque processus maintient une horloge locale logique
- Quand un processus effectue une opération locale, on augmente l'horloge de n
- Quand un processus envoie un message, il y met la valeur de son horloge locale
- Quand un processus reçoit un message, il met son horloge à max(horloge, heure message)+ n
  - Il se resynchronise donc si nécessaire
  - Jamais de retour en arrière

Exemple

Horloges non synchronisées



Lamport



Lamport

- L'algorithme de Lamport nous fournit un ordre partiel sur les événements
- Mais on ne peut rien dire de deux événements concurrents
- On peut construire un ordre total au dessus
  - Il suffit d'ordonner arbitrairement les événements concurrents (et seulement ceux là)
  - On donne un numéro unique à chacun des processus, qui sert pour l'ordonnancement (par exemple on l'ajoute en partie fractionnaire de l'horloge logique)
- Nouvelle propriété : 2 événements ne peuvent avoir le même temps logique

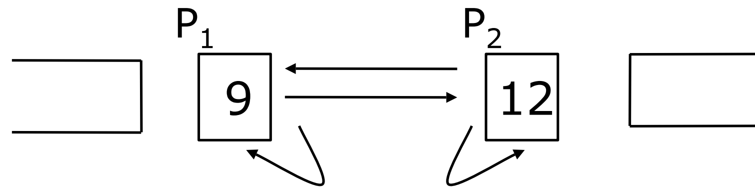
Application : Multicast totalement ordonné

- Exemple : gestion de compte
  - Un compte en banque est géré par une base de données
  - La base de données est répliquée sur plusieurs sites
  - Un client fait un retrait de 1000 euro
  - Le banquier veut créditer les intérêts
  - Ces deux opérations sont initiées en même temps
- Une opération sera exécutée par une communication multicast sur l'ensemble des BDs
- Problème : comment éviter un croisement des messages ?

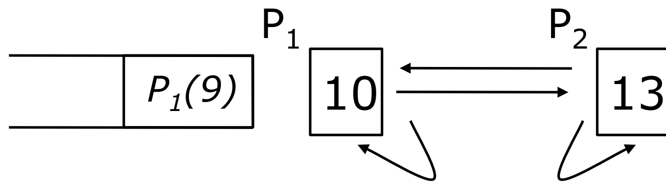
Application : Multicast totalement ordonné

- On suppose des communications sans pertes et que deux messages émis par le même émetteur seront reçus dans le même ordre par un receveur
- On utilise les horloges de Lamport avec ordre total
  - Chaque message contient l'heure locale (Lamport en ordre total) de l'émetteur
  - Un message multicast est aussi envoyé à tous les processus, y compris l'émetteur
  - Chaque processus maintient une queue de messages ordonnés par leur horloge logique
  - Chaque réception de message provoque l'envoie d'un ACK en multicast
  - On a toujours Horloge(message reçu) < Horloge(ACK) car l'envoie du ACK suit la réception
- Un message reçu ne peut être traité par l'application que lorsque tous les ACKS ont été reçus
- Au final, les processus ont exactement la même queue, donc nous avons un ordre total

Exemple



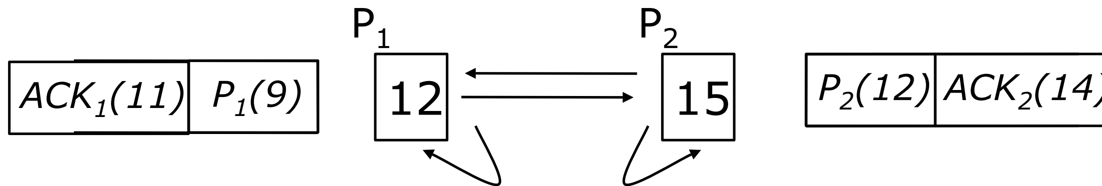
Chacun commence un multicast

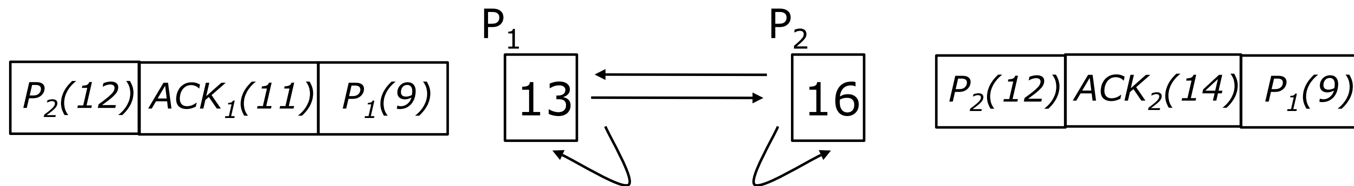


P2(12)

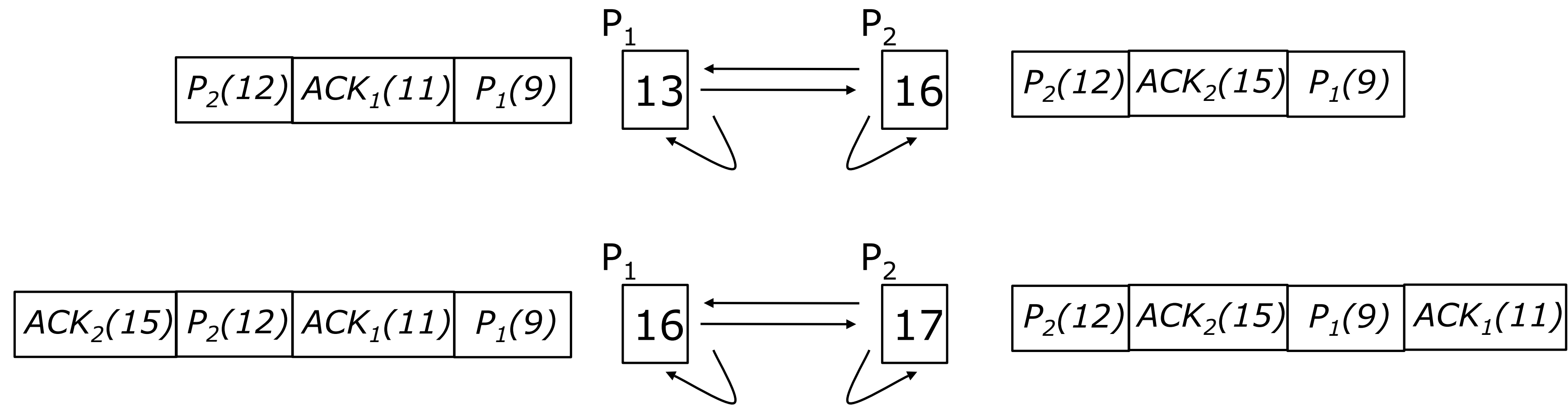
Réception du premier message

Envoie des ACKS





Exemple



Tous les messages et les ACKS ont été reçus
Le traitement des messages peut être effectué, avec en premier le message venant de P1

3 – État global

Définition

- Il peut être nécessaire de connaître l'état global d'un système distribué
- État global: état local de chacun des processus et messages en transit (émis mais non encore reçus)
- Application
  - Savoir qu'un système est arrêté
  - Sauvegarder l'état d'un système pour bien gérer les pannes (on redémarre l'ensemble au dernière état global)
- Probabilité de panne
  - Très élevée dans un SD
- Ex: si une machine a 5% de chance de tomber en panne pendant un calcul
  - Avec 10 machines, 50% de chance qu'au moins une tombe en panne
  - Avec 100 machines, on ne finira jamais le calcul sans aucune panne

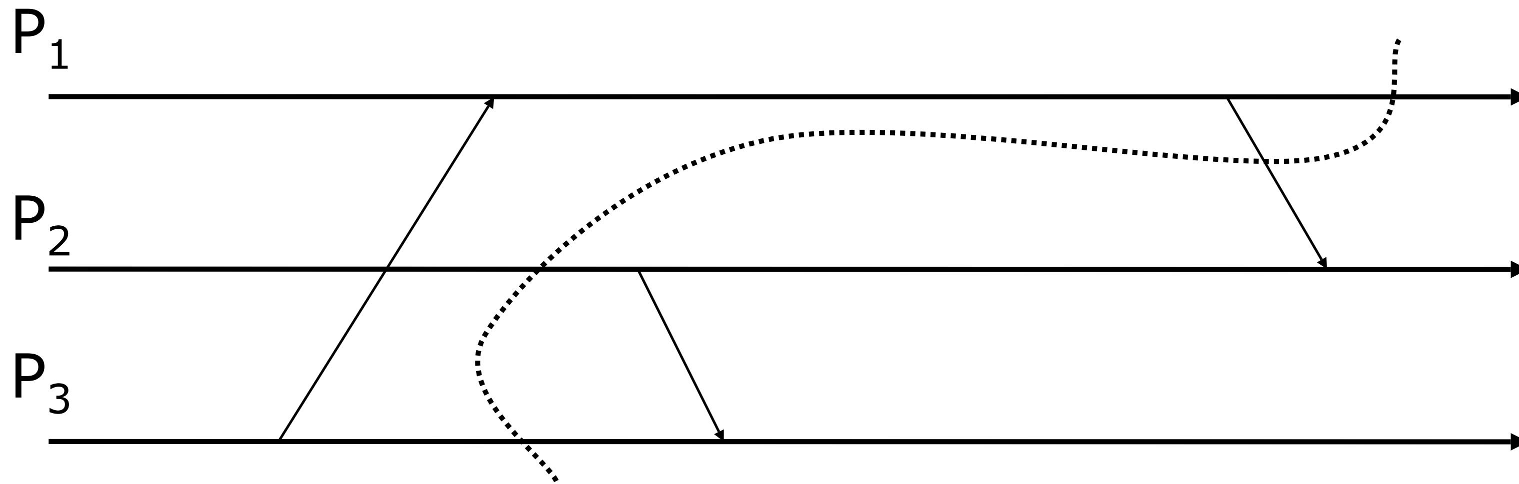
Sauvegarde et *rollback*

- Il faut pouvoir sauvegarder l'état d'un SD
- Mais pas de vue globale instantanée du système
- Sauvegarder chacune des machine n'importe quand ?
  - Si on redemarre depuis la dernière sauvegarde, est-ce que ça va bien marcher?
- On considère 2 machines, A et B avec le programme suivant
  - A demande à B d'incrémenter une valeur
- Si sauvegarde arbitraire, peut-on avoir une mauvaise exécution?
  - On sauvegarde A
  - A demande à B qui fait l'incrémentation
  - On sauvegarde B

Coupes

- Utilisation d’un formalisme pour représenter l'exécution distribuée
- Chaque processus est représenté par une ligne
  - Flèche du temps
- Chaque communication est une ligne reliant 2 processus
- On ne considère par les opérations internes à chaque processus
  - Aucun soucis avec la sauvegarde
- Une sauvegarde (*état global*) est la sauvegarde individuelle de chacun des processus
  - Ces points de sauvegarde forment une coupe
- Représentation
  - Un état global est représenté sous forme de coupe
  - Ce qui se trouve à gauche de la coupe s’est exécuté
  - Ce qui se trouve sur la droite sera re-exécuté en cas de panne
  - Cette coupe doit représenter un état global possible du système

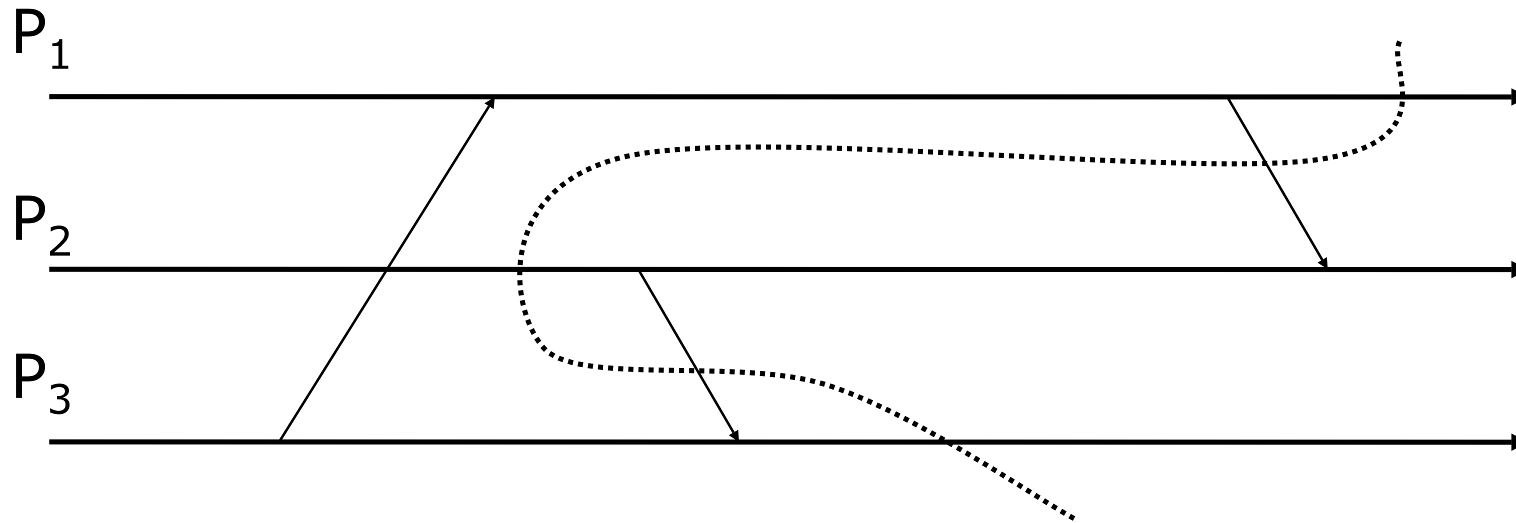
Coupes



- La coupe est représentée sous forme de courbe pointillée
  - L'intersection avec le trait d'un processus indique le point de sauvegarde de l'état local
- Tout message noté comme reçu dans la coupe a aussi été noté comme envoyé
- Cette coupe est dite cohérente
  - Elle correspond à un état possible du système

Université Nice SOPHIA ANTIPOLIS

Coupes



- Un message a été noté comme reçu mais non envoyé
- Cette coupe est dite non cohérente
- En cas de reprise après une panne, P3 considérera avoir reçu un message de P2 mais celui-ci ne l'aura pas encore envoyé
  - Pourquoi n'est-ce pas un état possible ?

Snapshot distribué

- Nous voulons enregistrer un état global, c'est-à-dire effectuer un snapshot distribué (Chandy et Lamport 1985)
- On suppose que tous les processus ont des canaux de communication ouverts et FIFO (les messages ne se doublent pas)
- Un processus P décide de démarrer le snapshot
  - Il enregistre son état local
  - Il envoie un message spécial (marqueur) à tous les autres processus
- Quand un processus Q reçoit le marqueur
  - Il fait une sauvegarde de son état local si c'est la première réception et envoie un marqueur à tous les processus
  - Si c'est une nouvelle réception d'un marqueur, il enregistre l'état du canal sur lequel il est arrivé, c'est-à-dire tous les messages qui sont arrivés depuis sa sauvegarde locale

Snapshot distribué

- Un processus a fini sa part de l'algorithme quand il a reçu un marqueur sur tout ses canaux entrants
- Dans ce cas, il peut envoyer à l'initiateur son état sauvegardé
- Le système peut toujours fonctionner pendant la phase de snapshot
  - Il peut traiter les messages, mais doit s'en souvenir en attendant un éventuel marqueur
- Chaque processus peut démarrer un snapshot, dans ce cas, il faut distinguer les marqueurs. On leur ajoute un numéro unique dépendant de l'initiateur
- On peut montrer que cet algorithme donne une coupe cohérente

Détection de la terminaison

- Le snapshot nous donne un état global où des messages peuvent être toujours en transit
- Pour qu'une application distribuée soit terminée, il faut que
  - Tous les processus aient fini
  - Les canaux de communication soient vides
- Une modification légère de l'algorithme précédent permet de s'en assurer
- Si un processus P a un canal de communication avec un processus Q, alors P (resp. Q) est le prédécesseur (resp. successeur) de Q (resp. P)

Détection de la terminaison

- Quand un processus Q finit sa partie du snapshot distribué il envoie un message à son prédécesseur
- Message DONE quand les deux conditions suivantes sont remplies
  - Tous les successeurs de Q ont envoyé un DONE
  - Q n'a pas reçu de message entre le moment où il a enregistré son état local et le moment où il a reçu le dernier marqueur pour tout ses canaux entrants
- Message CONTINUE
  - Dans tous les autres cas
- L'initiateur du snapshot recevra soit des DONE soit des CONTINUE de ses successeurs
  - Si que des DONE, plus de messages en transit, le programme est terminé
  - Si au moins un CONTINUE, il faut relancer un snapshot plus tard

4 – Élection

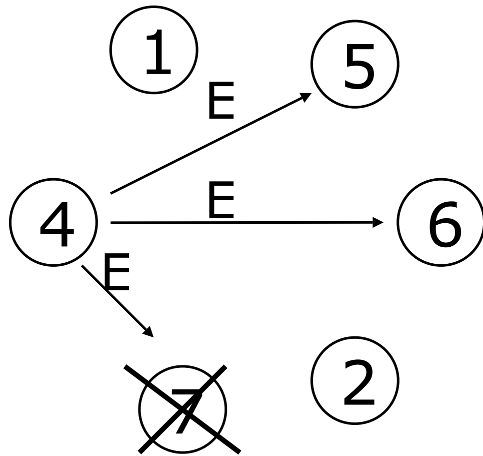
Introduction

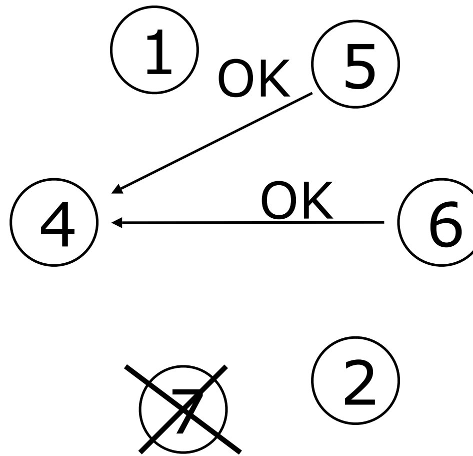
- Certains algorithmes distribués ont besoin qu'un processus agisse comme coordinateur
- Souvent, pas de propriété particulière concernant ce processus
- On suppose qu'il existe un moyen de distinguer les processus (adresse IP de la machine si 1 processus/machine…)
- Le but d'un algorithme d'élection est de localiser le processus qui a, par exemple, le numéro le plus élevé

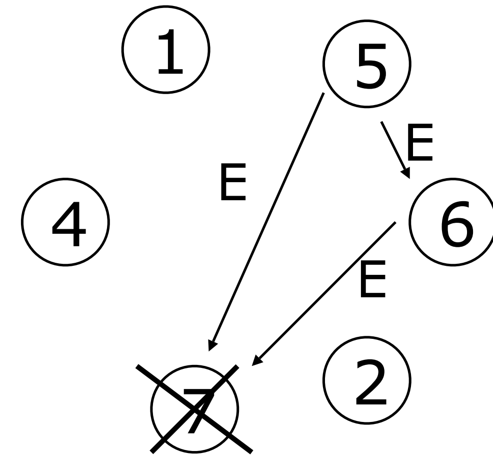
Algorithme Bully

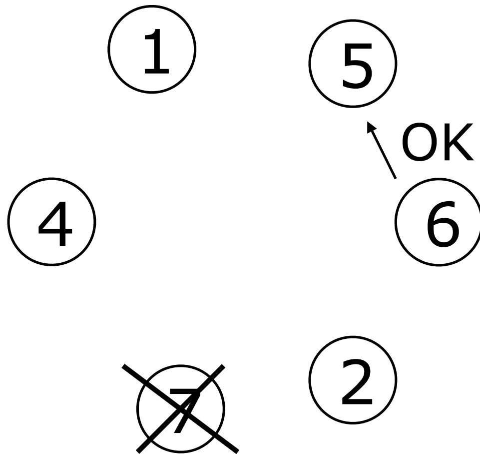
- Publié par Garcia-Molina en 1982
- Quand un processus détecte l'absence de chef, il démarre une élection
  - Il envoie un message ELECTION à tous les processus de plus haut rang
  - Si personne ne répond, il gagne l'élection et devient le nouveau chef
  - Si quelqu'un de plus haut rang lui répond, il laisse sa place
- Un processus recevant un message ELECTION d'un processus de plus bas rang
  - Il répond un OK
  - Il démarre une élection si il ne l'a pas déjà fait
- Au bout d'un certain temps, il ne restera qu'un processus
  - Il annoncera son élection aux autres avec un COORDINATOR
- Si un processus revient dans le système après une panne, il demande une nouvelle élection
- Au final, le processus le plus fort gagne, d'où le nom :)

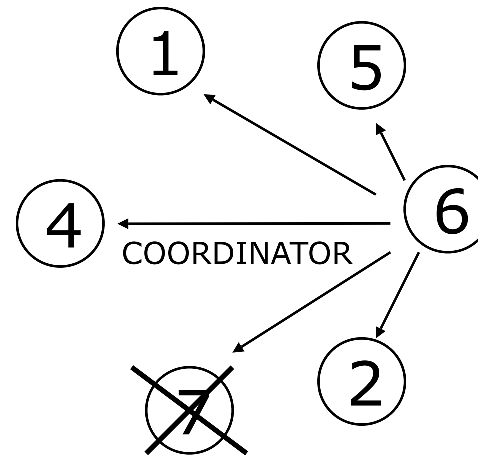
Algorithme Bully







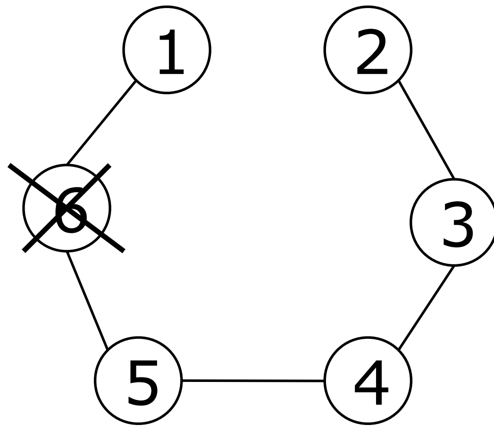




Élection sur un anneau

- Dans l'algorithme précédent, un processus connaît tous les autres
- Pas forcément réaliste
- On considère que les processus sont logiquement ou physiquement ordonnés
- Un processus connaît son successeur (ainsi que le suivant…)
- Quand un processus remarque que le coordinateur est manquant
  - Il fabrique un message ELECTION contenant son numéro
  - Et l'envoie à son successeur
  - Si le successeur est en panne, il l'envoie au suivant…
- A chaque étape sur l'anneau, un processus ajoute son numéro à la liste comme candidat potentiel pour l'élection
- Finalement, ce message retourne à l'émetteur
  - Un nouveau message COORDINATOR est émis, contenant la liste des processus ayant participé
  - Cela indique à chacun le nouveau coordinateur et les membres de l'anneau
  - Ce message effectue un tour et est arrêté

Élection sur un anneau



Que se passe-t-il si 2 et 5 détectent la panne de 6 en même temps?

5 – Exclusion mutuelle

Exclusion

- Une section critique est une partie du code qui ne peut être exécutée que par un processus à la fois
- Dans un système centralisé, utilisation de sémaphores, moniteurs…
- Plusieurs algorithmes disponibles dans les systèmes distribués

Algorithme centralisé

- Simule le comportent d'un système monoprocesseur
- Un processus est élu coordinateur
- Quand un processus veut entrer dans une section critique, il demande au coordinateur
  - Si aucun autre processus dans la section critique, alors il envoie l'autorisation
  - Sinon, il ne répond pas (ou envoie un message refusant l'accès)
- Quand un processus sort de la section critique, il envoie un message au coordinateur
- Si coordinateur FIFO, mécanisme fiable
- Seulement 3 messages nécessaires par section critique
- Mais peu fiable car 1 point de panne
  - Si blocage en attendant l'accès, comment savoir si le coordinateur tombe en panne?
  - Pas de problème si envoie d'un message de refus au lui de blocage de l'appelant
- Risque de surcharge du coordinateur

Algorithme distribué

- Lamport (1978), amélioré par Ricart et Agrawala (1981)
- Nécessite un ordre total de tous les évènements dans le système
- Un processus voulant entrer dans une section critique
  - Fabrique un message indiquant la section, son numéro de processus et l'heure actuelle
  - Envoie ce message à tous les autres processus (lui y compris)
- Quand un processus reçoit une demande d'un autre processus, il y a 3 réactions possibles
  - Si le receveur n'est pas dans la section critique et ne veut pas y entrer, il répond OK
  - Si le receveur est dans la section critique, il diffère sa réponse
  - Si le receveur veut entrer dans la section critique, il compare l'heure du message entrant avec l'heure de sa demande. La plus basse gagne (ok ou retardement)
- Un processus qui a reçu l'autorisation de tous les autres entre dans la section critique
- Une fois finie, il envoie un OK à tout ceux en attente

Algorithme distribué

- Si n processus dans le système, il faut 2(n-1) messages (un message envoyé à soi-même non comptabilisé)
- Mais nous avons maintenant n points de pannes possibles
  - Possibilité de contourner le problème avec envoie de refus
- Nécessité de maintient d'une liste des autres processus
- Risque de surcharge des processus
  - Amélioration: fonctionner à la majorité plutôt qu'à l'unanimité
- Mais au final, plus compliqué, plus lourd et moins robuste que le centralisé
  - Pourquoi le faire? Parce que!