
Master d'Informatique 2009 -2010
spécialité « SAR »
Algorithmique Répartie (MI048)

Cours 1 à 4

L. Arantès, C. Dutheillet, M. Potop-Butucaru, S. Dubois



Organisation

10 semaines:

Cours, TD et TME

TME : MPI et Promela (spin)

Evaluation:

Contrôle continu : 40%

Examen : 60 %

2

Sujets

- Terminologie, topologie
- Temps, causalité, horloge logique
- Exclusion mutuelle
- Algorithmes à Vagues
- Election
- Terminaison
- Etat Global
- Tolérance aux fautes et point de reprise
- Introduction au large échelle (P2P)

3

Bibliographie

- Gerard Tel, *Introduction to Distributed Algorithms*, Cambridge University Press, 1994, 2000 (2ème édition).
- Nancy Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.
- Michel Raynal, *Synchronisation et état global dans les systèmes répartis*, Eyrolles, 1992
- H. Attiya and J. Welch, *Distributed Computing. Fundamentals, Simulations and Advanced Topics*, McGraw-Hill, 1998.

4

Introduction Algorithmique répartie - Plan

- Système réparti
 - ➔ mémoire partagée vs échange de messages
- Topologie des systèmes répartis
- Modèles de Fautes et Modèles Temporels
- Problèmes inhérents à la répartition
- Evaluation et vérification d'un algorithme réparti

5

Système réparti

6

Qu'est-ce qu'un système réparti ?

Ensemble interconnecté d'entités autonomes
qui communiquent via un médium de communication (G. Tel)

Entités :

ordinateurs, processeurs, processus

Autonomes :

chacune des entités possède son propre contrôle

Interconnexion :

capacité à échanger de l'information : canaux de communication ou mémoire partagée.

7

Caractérisation d'un calcul réparti

Non séquentiel :

deux instructions peuvent être exécutées simultanément

Non centralisé :

les paramètres décrivant l'état du système sont répartis

Non déterministe :

deux actions concurrentes peuvent être exécutées dans n'importe quel ordre.

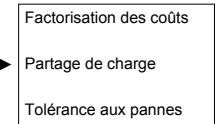
le comportement d'une entité peut dépendre de ses interactions avec les autres entités.

8

Buts d'un système réparti

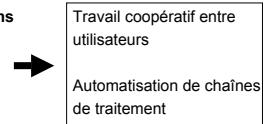
But technique : mise en commun des **ressources** matérielles de plusieurs machines

- processeur et mémoire : + de capacité de calcul
- disques : + de capacité de stockage
- imprimantes
- ...



But fonctionnel : mise en commun d'**informations** entre plusieurs utilisateurs ou systèmes

- fichiers ou bases de données
- événements ou alarmes
- ...



9

Classification des applications réparties

Deux classes d'applications :

Processus coopérants

Les processus interagissent via des mémoires ou des variables partagées.

- gérer les conflits d'accès aux ressources communes (exclusion mutuelle)

Processus communicants :

Les processus s'échangent des messages par l'intermédiaire de canaux.

- gérer l'échange de la connaissance

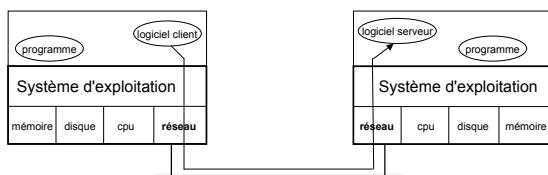
10

Exemples

- **Système étendu (client/serveur)**
- **Système à architecture répartie**
- **Système d'exploitation réparti**

11

Système étendu (client–serveur)



Caractéristiques :

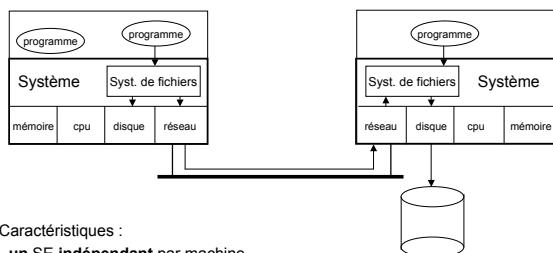
- un SE indépendant par machine
- un logiciel explicite de communication (login, telnet, ftp, rcp, ...)

Exemples :

Unix, MacOs,
Dos, Windows 95

12

Système à architecture répartie

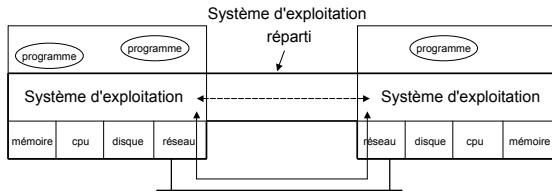


Caractéristiques :

- un SE indépendant par machine
- un accès transparent à certaines ressources distantes

13

Système d'exploitation réparti



Caractéristiques :

- les SE locaux **coopèrent** pour rendre leurs services (cpu, memoire, disque)
- et donner l'illusion d'une **machine virtuelle unique** composée des **ressources de tous les sites**

Exemples :

- Chorus (INRIA – 80')
- V-System (Stanford U. – 84)
- Mach (Carnegie Mellon U. – 86)
- Amoeba (Amsterdam U. – 90)

14

Canaux de communication

Risque de déséquancement des messages

- P envoie M1 puis M2 à Q. Q reçoit M2 puis M1
- Pas de déséquancement = canal FIFO

Risque de perte de messages

- Un message émis n'est jamais reçu
- Communications fiables : tout message émis est reçu exactement une fois
 - au bout d'un temps quelconque, mais fini.
 - sans détérioration.

Certains canaux peuvent avoir une capacité bornée



On se place ici au niveau logique et pas au niveau physique !

15

Topologies de systèmes répartis

16

Topologie des systèmes

Représentation sous forme de graphe :

- nœuds = processus
- arêtes = canaux de communication
 - Canal unidirectionnelle (dirigée) de p vers q = arc de p vers q
 - canal bidirectionnel entre p et q = arête entre p et q

Caractéristiques du graphe

- connexe : chaque paire de nœuds est reliée par un chemin.
- fortement connexe : graphe orienté, où il existe un chemin entre chaque paire de nœuds, en respectant le sens des arcs.
- incomplet : tous les processus ne communiquent pas deux à deux directement.

La topologie du graphe peut être un critère pour l'existence d'une solution répartie à un problème

17

Paramètres définis sur le graphe

Distance entre deux nœuds :

Longueur du plus court chemin entre ces deux nœuds

Diamètre du graphe :

La plus longue des *distances* entre deux nœuds du graphe

Degré d'un nœud

Nombre de voisins du nœud

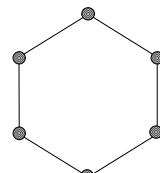
Ces paramètres peuvent être utiles dans le calcul de la complexité
d'une application répartie

18

Topologies particulières (1)

Anneau unidirectionnelle:

- ➔ N nœuds
- ➔ N arêtes
- ➔ Tous les nœuds sont de degré 2 (ont 2 voisins)
- ➔ Diamètre : $n-1$

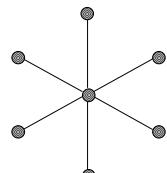


19

Topologies particulières (2)

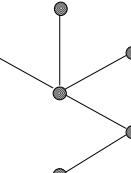
Etoile :

- ➔ N nœuds
- ➔ (N-1) arêtes
- ➔ Tous les nœuds sont de degré 1, sauf le nœud central qui est de degré N-1.
- ➔ Diamètre : 2



Arbre :

- ➔ N nœuds
- ➔ (N-1) arêtes
- ➔ Pas de cycle

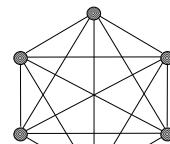


20

Topologies particulières (3)

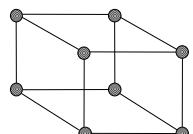
Clique (graphe complet) :

- ➔ N nœuds
- ➔ Tous les nœuds sont reliés deux à deux par une liaison
- ➔ $n(n-1)/2$ arcs
- ➔ Diamètre : 1



Hypercube :

- ➔ N = 2^n nœuds
- ➔ Chaque numéro de nœud est codé sur n bits
- ➔ Il existe une liaison entre les nœuds dont les numéros ne diffèrent que d'un bit



21

Topologie physique et topologie logique

Topologie physique :

- ➔ Câblage du réseau

Topologie logique (structure de contrôle) :

- ➔ Manière dont la communication entre les sites est organisée

La topologie logique et la topologie physique peuvent différer.

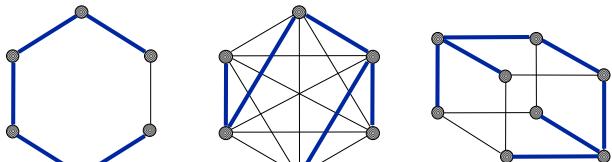
Dans ce cas, certaines liaisons physiques ne sont pas utilisées pour la communication entre les sites.

- ➔ Possibilité d'implanter une structure X sur une topologie Y
- ➔ Augmentation de la fiabilité

22

Structure de contrôle en arbre

Sur une topologie physique connexe quelconque, on peut TOUJOURS implanter une structure de contrôle en arbre.



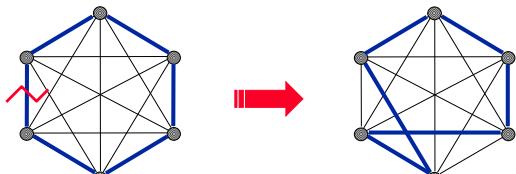
Un arbre est un graphe connexe sans cycle.

Exemple : Diffusion d'information

23

Reconfiguration d'un anneau

Si une liaison de l'anneau logique tombe en panne, on peut reconstruire un anneau en passant par une autre liaison



Reconfiguration dynamique ➔ coopération des processus pour reconstruire l'anneau

24

Modèles de fautes et modèles temporels

25

Modèles de fautes

Origines des fautes

- ➔ **fautes logicielles (de conception ou de programmation)**
 - quasi-déterministes, même si parfois conditions déclenchantes rares
 - très difficiles à traiter à l'exécution : augmenter la couverture des tests
- ➔ **fautes matérielles (ou plus généralement système)**
 - non déterministes, transitaires
 - corrigées par point de reprise ou *masquées* par réplication
- ➔ **piratage**
 - affecte durablement un sous-ensemble de machines
 - masqué par réplication

Composants impactés

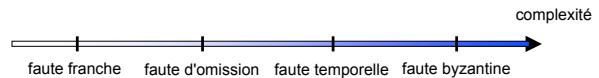
- ➔ processus
- ➔ calculateurs
- ➔ canaux de communication

26

Modèles de fautes

Classification des fautes

- ➔ **faute franche** : arrêt définitif du composant, qui ne répond ou ne transmet plus
- ➔ **faute d'omission** : un résultat ou un message n'est transitoirement pas délivré
- ➔ **faute temporelle** : un résultat ou un message est délivré trop tard ou trop tôt
- ➔ **faute byzantine** : inclut tous les types de fautes, y compris le fait de délivrer un résultat ou un message erroné (intentionnellement ou non)



27

Modèles temporels

Constat

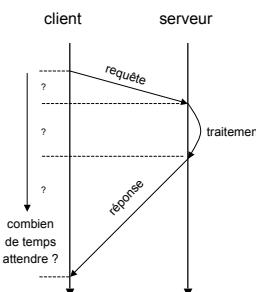
- ➔ vitesses processus différentes
- ➔ délais de transmission variables

Problème

- ➔ ne pas attendre un résultat qui ne viendra pas (suite à une faute)
- ➔ combien de temps attendre avant de reprendre ou déclarer l'échec ?

Démarche

- ➔ élaborer des modèles temporels
- ➔ dont on puisse tirer des propriétés



28

Modèles temporels (2)

Modèle temporel = hypothèses sur :
- délais de transmission des messages
- écart entre les vitesses des processus

système synchron

Modèle Délais/écart Bornés Connus (DBC)
- permet la détection **parfaite** de faute

système partiellement synchron

Modèle Délais/écart Bornés Inconnus (DBI)

système asynchrone

Modèle Délais/écart Non Bornés (DNB)



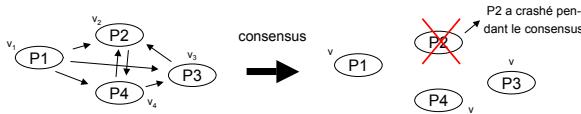
29

Modèles temporels (3)

Résultat fondamental :

Fischer, Lynch et Paterson 85 : le problème du consensus ne peut être résolu de façon déterministe dans un système **asynchrone** en présence de ne serait-ce qu'une faute franche.

Problème du consensus : N processus se concertent pour décider d'une valeur commune, chaque processus proposant sa valeur initiale v_i .



Spécification formelle du consensus :

- terminaison : tout processus correct finit par décider
- accord : deux processus ne peuvent décider différemment
- intégrité : un processus décide au plus une fois
- validité : si v est la valeur décidée, alors v est une des v_i

30

Notre modèle

Ensemble de processus séquentiels indépendants

- Chaque processus n'exécute qu'une seule action à la fois

Communication par échange de messages

- Aucune mémoire partagée
- Les entrées des processus sont les messages reçus, les sorties sont les messages émis

Système asynchrone (souvent considéré) :

- Asynchronisme des communications
 - Aucune hypothèse sur les temps d'acheminement des messages (Pas de borne supérieure)
- Asynchronisme des traitements
 - Aucune hypothèse temporelle sur l'évolution des processus

Pas d'horloge commune

31

Critères d'évaluation

Pourquoi existe-t-il plusieurs solutions à un problème de répartition ?

Parce qu'elles ne sont pas forcément comparables.

Problèmes inhérents à la répartition

32

Les solutions peuvent diverger par

les hypothèses qu'elles font sur :

- le système de communication
- la connaissance qu'un processus doit avoir de son environnement
- la connaissance qu'un processus doit avoir de l'état des autres processus

leur capacité à tolérer les pannes

33

Le système de communication

Un algorithme réparti fait souvent des hypothèses sur

La topologie du système de communication

Les propriétés comportementales de ce système :

- fiabilité des liaisons
- modèles de synchronisme (temporel)
- possibilité de déséquenchement des messages

Moins un algorithme fait d'hypothèses, plus il est général

MAIS...

Un algorithme défini pour un environnement particulier peut être le plus efficace dans cet environnement

34

Connaissance de l'environnement

Quelle est l'information connue par chaque processus AVANT le lancement de l'algorithme ?

En général, un processus possède des informations sur

- la topologie du réseau
 - Nombre de sites
 - Diamètre
 - Topologie complète
- l'identité des processus
 - Les noms des sites sont uniques
 - Chaque site connaît son propre nom
 - Un site peut ne pas connaître l'identité de ses voisins, mais il connaît au moins leur existence

35

Connaissance de l'état des autres

L'état global de l'application n'est pas connu

Un processus ne peut pas prendre une décision sans échanger de l'information avec les autres

Plus la quantité d'information nécessaire à la décision est importante, plus il est difficile de parvenir à cette décision

- Nombreux échanges de messages
- Instabilité de l'information obtenue

36

Résistance aux pannes

Les défaillances peuvent se produire

- au niveau des processus
- au niveau des canaux de communication

La capacité de résistance de l'application dépend

- du type de défaillance observée
 - Perte de la fiabilité de la liaison
 - Panne franche de processus (arrêt)
 - Défaillances byzantines
- du degré de symétrie de l'application
 - Aucune symétrie
 - Symétrie de texte (comportement dépend de l'identité)
 - Symétrie totale

37

Type de Problème

Réaliser une opération qui n'existe pas sans répartition ou concurrence de plusieurs processus

- Routage
- Exclusion mutuelle
- Election

Réaliser une opération qui est compliquée parce que l'application est répartie

- Obtention de l'état du système
- Détection de la terminaison de l'application

38

Routage

Calculer le chemin « optimal » sur lequel le processus i envoie ses messages au processus j.

Problème :

- i ne connaît du réseau que ses voisins
- i doit trouver quel est le voisin « le plus adapté » pour transmettre ses messages à j

Solution :

- calcul du plus court chemin dans un graphe en cas de communication point-à-point
- calcul d'un arbre de recouvrement de poids minimal pour une diffusion

39

Exclusion mutuelle

Ne pas autoriser les accès simultanés à une ressource pour préserver sa cohérence.

Problème :

si les processus sont géographiquement distants, on ne peut plus utiliser de mécanisme de type sémaphore centralisé.

Solutions :

- utiliser un jeton qui fait office d'autorisation d'accès
 - Mécanisme de régénération du jeton en cas de perte
- demander l'autorisation pour accéder à la ressource à tous les autres sites
 - Ne passe pas à l'échelle
- gérer une file d'attente (centralisée ou distribuée) des requêtes
 - Fragilité d'une file centralisée
 - Difficulté à gérer la cohérence d'une file distribuée

40

Election

Dans certaines applications, un processus joue un rôle particulier.

Comment choisir ce processus initialement ? Comment le remplacer s'il tombe en panne ?

Solution :

- Définir un critère d'élection unique et qui puisse toujours être satisfait
- Faire participer tous les processus à l'élection
- Gérer la communication de sorte qu'à la fin de l'élection, un processus est dans l'état ELU, tous les autres sont dans l'état BATTU

Contrainte sur l'application :

- Tous les processus éligibles doivent posséder la partie de code correspondant au rôle particulier

41

Calcul d'état global

Qu'est-ce qui définit l'état du système ?

- ➔ L'état de chacun des processus à un instant donné
- ➔ L'état à ce même instant de tous les canaux de communication

Pourquoi cet état est-il difficile à obtenir ?

- ➔ Les informations sont réparties géographiquement
- ➔ Il n'y a pas d'horloge globale

Quelles est la solution ?

- ➔ Collecter « intelligemment » les informations auprès de chaque processus
 - Synchroniser la prise d'information
 - Ne pas perturber l'application observée

42

Evaluation et vérification d'un algorithme réparti

44

Détection de la terminaison

Comment être certain qu'à un instant t tous les processus sont définitivement inactifs ?

L'inactivité des processus correspond-elle à la terminaison correcte de l'application ?

- ➔ Déetecter qu'un processus inactif ne redeviendra jamais actif
- ➔ Déetecter que les canaux sont vides

Difficulté :

un processus ne peut pas toujours savoir à partir de son information locale si l'application est terminé ou non.

Solution :

coordonner la prise d'information

43

Critères d'évaluation

La complexité en nombre d'opérations est peu significative

- ➔ Les opérations sont exécutées par des sites différents
- ➔ Tous les sites n'exécutent pas le même nombre d'opérations

Complexité en messages :

- ➔ Nombre total de messages échangés au cours de l'exécution
- ➔ Critère biaisé si les tailles de messages sont très différentes
 - Complexité en nombre de bits d'information échangés

Complexité en temps :

- ➔ Comment définir une complexité en temps lorsque le système ne possède pas d'horloge globale ?

45

Complexité en temps

En l'absence d'horloge globale, utilisation d'une notion idéalisée du temps :

- le temps d'exécution d'un pas de calcul est nul
- le temps de transmission d'un message nécessite une unité de temps

Dans ce cas

Complexité en temps =
longueur de la plus longue chaîne de messages

46

Vérification d'une application répartie

Problème : l'application est non déterministe

- Multiplication des traces d'exécution
 - Pas de construction exhaustive des états
- Difficulté (impossibilité) de reproduire une exécution
 - Extrêmement difficile à débugger

Nécessité de prouver l'application *a priori* sans construction de tous les états

- Utilisation de méthodes (inductives) basées sur des propriétés
 - Invariants et propriétés stables pour la sûreté
 - Fonctions de progrès pour la vivacité

47

Propriétés de sûreté et de vivacité

Sûreté :

rien de mauvais ne se produira dans l'exécution de l'application

- *Exclusion Mutuelle* : toujours au plus un processus en section critique
- *Détection de la Terminaison* : si on détecte la terminaison de l'application, alors celle-ci est réellement terminée (pas de fausse détection)

Vivacité :

quelque chose de bien finira par se produire dans l'exécution

- *Exclusion Mutuelle* : si un processus demande la section critique, il finira par l'obtenir
- *Détection de la Terminaison* : si l'application se termine, alors cette terminaison sera détectée

48

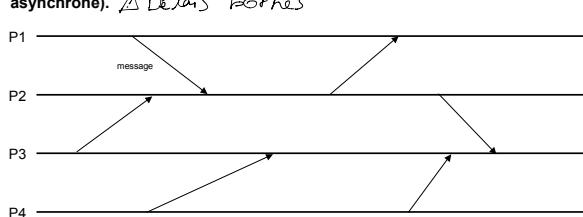
Gestion du temps dans les applications réparties

Plan

- Relation de précédence
 - Principe de la causalité
 - Causalité des horloges
 - Forte causalité (caractérisation de la causalité)
- Modèles d'horloges
 - scalaires (Lamport)
 - vectorielles (Mattern et Fidge)
- Implémentation d'horloges vectorielles
 - Singhal / Kshemkalyani
- Facteur de dépendance

Contexte (1)

On considère un système réparti comme un ensemble de **processus asynchrones** s'exécutant sur différents sites. Pas d'horloge globale. Les processus communiquent **uniquement par message** (pas de mémoire commune). Les délais de communication sont finis, mais **non connus** et fluctuants (**modèle asynchrone**). Δ Délais bornés

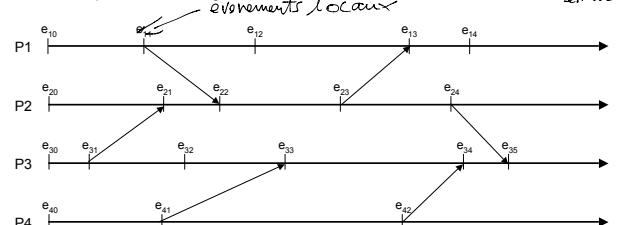


Contexte (2)

Pour étudier le système réparti, on s'intéresse à la succession des **événements** se produisant sur les différents processus. On définit trois types d'événements :

- les événements locaux (changement de l'état interne d'un processus)
- les émissions de message \rightarrow une émission est forcément suivie d'une réception
- les réceptions de message

évenements locaux



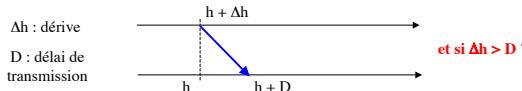
Le temps

- A-t-on besoin du temps ? On veut pouvoir

- Tracer une exécution
 - debug
- ...

Relation de causalité entre événements

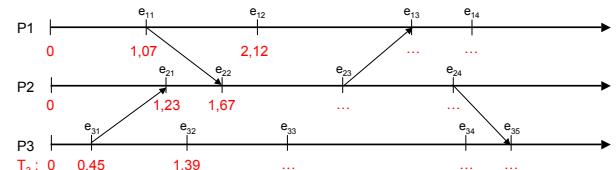
- Pourquoi ne peut-on pas utiliser le temps physique ?



Horloge Physique

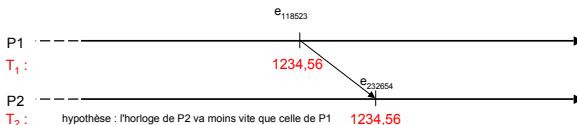
Les événements sont datés à l'aide de l'horloge physique de chaque site S_i . On suppose que :

- toutes les horloges sont parfaitement synchronisées au départ,
- leur résolution est suffisante pour dater distinctement tous les événements locaux.



Limites Horloges Physiques

Les horloges physiques dérivent au cours du temps, et le sens et l'amplitude de la dérive est propre à chaque machine. Les événements locaux restent correctement ordonnés par leur date, mais pas nécessairement ceux se produisant sur des machines distinctes. Par conséquent, la datation par horloge physique ne tient pas compte de la causalité des événements non locaux.



Conséquence : la précedence causale n'est pas respectée.

$e_{118523} \rightarrow e_{232654}$ et pourtant $T(e_{118523}) = T(e_{232654})$

Relation de Précedence : Causalité

Principe de la causalité : la cause précède l'effet.

Pour étudier la causalité entre les événements, on définit la relation de **précedence causale**, notée \rightarrow traduisant une causalité **potentielle** entre les événements.

Définition

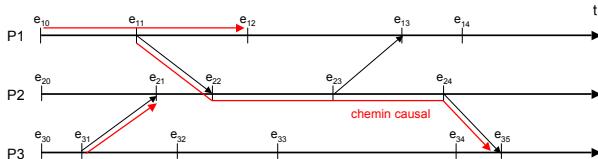
$a \rightarrow b$ si et seulement si :

- a et b sont des événements du **même processus** et a a lieu **avant** b
- a est l'**envoi** d'un message m et b est la **réception** de m
- il existe c tel que $a \rightarrow c$ et $c \rightarrow b$ (relation transitive)

$a \rightarrow b$
 a a **potentiellement causé** b
 a **précède causalement** b
 a **happened before** b

Relation de Précérence: causalité

Deux événements qui ne sont pas causalement dépendants l'un de l'autre sont dit **concurrents**, noté \nparallel : $e \nparallel e' \Leftrightarrow e \not\rightarrow e'$ et $e' \not\rightarrow e$



Relations :

$$\begin{array}{ll} e_{10} \rightarrow e_{12} & e_{12} \nparallel e_{22} \\ e_{31} \rightarrow e_{21} & e_{32} \nparallel e_{23} \\ e_{11} \rightarrow e_{35} & e_{33} \nparallel e_{14} \end{array}$$

Précérence causal : $e \rightarrow f$

- e_i^n : $n^{\text{ème}}$ événement sur le site i

Definition (Lamport 78)

$e_i^n \rightarrow e_j^m$ (e_i^n précède causalement e_j^m) ssi :

- $i = j$ et $n < m$: **précérence locale**

OU

- \exists mes, $e_i^n = \text{send}(\text{mes})$ et $e_j^m = \text{receive}(\text{mes})$: **précérence par message**

OU

- $\exists e_k^p$, $e_i^n \rightarrow e_k^p$ et $e_k^p \rightarrow e_j^m$: **relation transitive**

- l'ordre défini est **partiel** (existence d'événements $_$)

Datation

A chaque événement on associe sa date d'occurrence :

$$a \xrightarrow[D]{} D(a)$$

Objectif : trouver un système de datation D qui **respecte** et **représente** au mieux la relation de précédente causale.

- respecte : $(a \rightarrow b) \Rightarrow D(a) < D(b)$ exigence minimale, naturelle
- représente : $(a \rightarrow b) \Leftarrow D(a) < D(b)$ plus difficile à obtenir

Définition d'une horloge logique

- H** : ensemble des événements de l'application, muni de l'ordre partiel \rightarrow

si ($e \not\rightarrow e'$ et $e' \not\rightarrow e$) alors $e \nparallel e'$ (**concurrence**)

- T** : domaine de temps, muni de l'ordre partiel $<$

- Horloge logique :**

$$\begin{aligned} C : & H \rightarrow T \\ & e \rightarrow C(e) \end{aligned}$$

tel que $e \rightarrow e' \Rightarrow C(e) < C(e')$ (**consistance**)*

* L'horloge capture la causalité

cohérente

Causalité et horloges logiques

- On utilise souvent les horloges pour reconstruire (en partie) la relation de causalité

- $C(e) < C(e') \Rightarrow ?$

 - Soit $e \rightarrow e'$
 - Soit $e \parallel e'$

- Si

$$C(e) < C(e') \Rightarrow e \xrightarrow{\text{causalité}} e' \quad (\text{consistance forte})$$

les horloges caractérisent la causalité

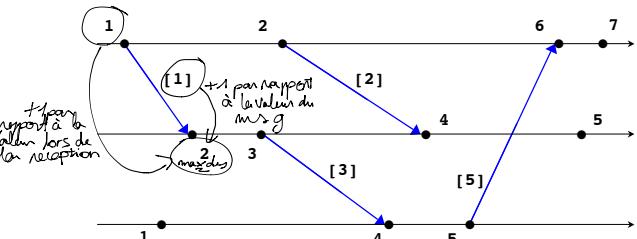
Choisir le type d'horloge en fonction des besoins de l'application

Modèles d'horloges

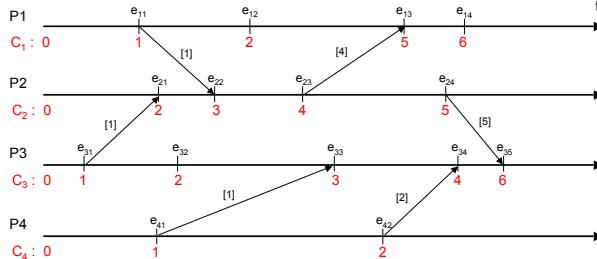
Horloges scalaires (Lamport [78])

- $T = \text{entiers naturels } N$
- Tous les messages portent l'horloge de leur émetteur à l'instant d'émission (**estampillage**)
- 2 règles de mise à jour :**
 - R1** : avant tout événement (interne, émission, réception), le site i exécute
 - $C_i = C_i + d$ ($d > 0$, généralement $d = 1$)
 - R2** : lorsque le site i reçoit un message portant une estampille C_{msg} , il exécute
 - $C_i = \max(C_i, C_{msg})$
 - Appliquer R1
 - Délivrer le message -> événement « réception »

Exemple



Horloges scalaires : exemple



Ordre total des événements

- Deux événements peuvent avoir la même date logique (ils sont alors concurrents).
- Horloges scalaires peuvent être utilisées pour définir un **ordre total** << sur les événements
- Pour ordonner totalement les événements, on complète la date logique d'un événement par le numéro du processus où il s'est produit : $D(e) = (C_i, i)$, et on utilise la relation d'ordre :

$$e_i \ll e_j \Leftrightarrow C(e_i) < C(e_j) \text{ ou } [C(e_i) = C(e_j) \text{ et } i < j]$$



Exemples utilisation :

- Exclusion mutuelle répartie (la datation logique permet d'ordonner totalement les requêtes pour garantir la vivacité de l'algorithme).
- Diffusion fiable et totalement ordonnée

Horloges scalaires : propriétés

- C respecte la causalité : $(a \rightarrow b) \Rightarrow C(a) < C(b)$
 - vrai pour deux événements locaux
 - vrai pour une émission et une réception
 - vrai dans tous les cas par induction le long du chemin causal
- C capture la causalité mais ne la caractérise pas : $C(a) < C(b) \not\Rightarrow a \rightarrow b$
 - exemple : $C(e_{12}) = 2 < C(e_{23}) = 4$ et pourtant $e_{12} \parallel e_{23}$
- $C(e) = n$ indique que $(n - 1)$ événements se sont déroulés séquentiellement avant e .

Horloges vectorielles : définition

Fidge, Mattern [88]

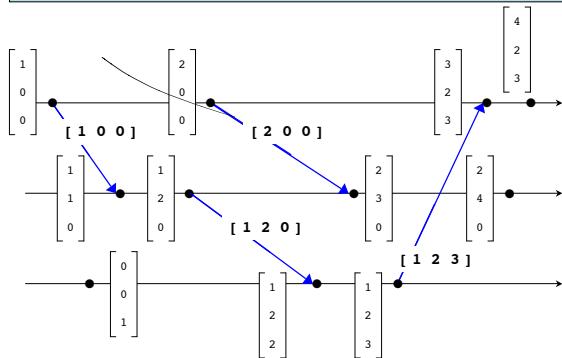
- Supposent un nombre fixe N de processus
 - $T = IN^N$
 - Chaque processus i gère un vecteur VC_i de taille N
 - $VC_i[i]$: nombre d'événements du processus i
 - $VC_i[j]$: connaissance qu'a i de l'avancement de l'horloge de j
 - A tout instant, l'état réel d'avancement du système est donné par $W = (VC_1[1], \dots, VC_1[N], \dots, VC_N[1], \dots, VC_N[N])$
- mais aucun processus ne peut l'obtenir...

Horloges vectorielles : algorithme de mise à jour

- Pour le processus P_i :**

- A chaque événement (émission, réception ou événement interne) de P_i , incrémentation de $VC_i[i]$:
 - $VC_i[i] = VC_i[i] + 1$
- A l'émission d'un message m :
 - Le message porte la valeur courante de VC_i
- A la réception d'un message m portant une horloge $m.VC$:
 - Mise à jour de l'horloge courante :
 - $\forall x, VC_i[x] = \max(VC_i[x], m.VC[x])$

Exemple



Horloges vectorielles et causalité

- Les relations suivantes sont définies:**

$$VC_i \leq VC_j \Leftrightarrow \forall k \quad VC_i[k] \leq VC_j[k]$$

$$VC_i < VC_j \Leftrightarrow VC_i \leq VC_j \text{ et } VC_i \neq VC_j$$

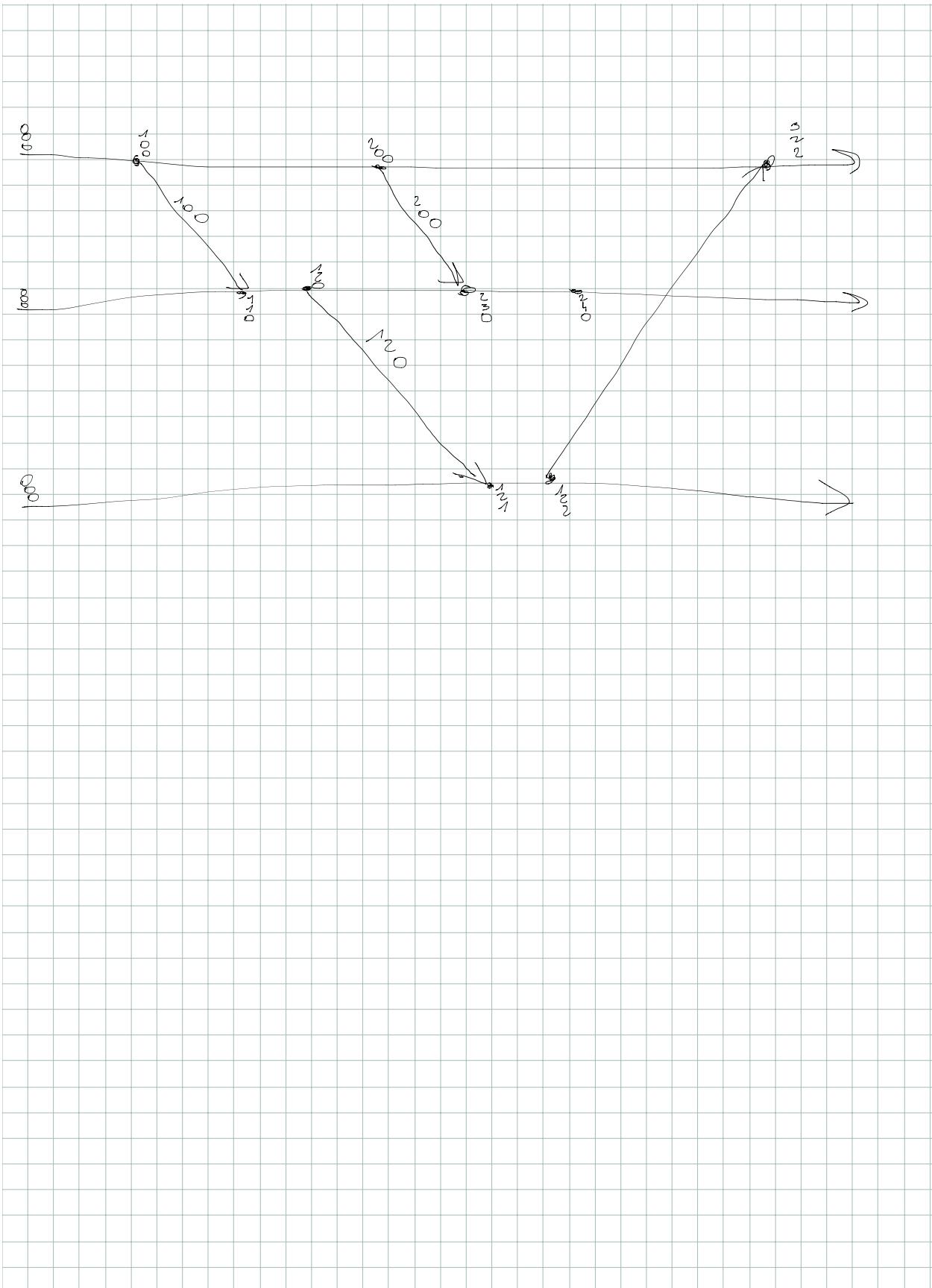
$$VC_i \parallel VC_j \Leftrightarrow !(VC_i \leq VC_j) \text{ et } !(VC_j \leq VC_i)$$
- Les horloges vectorielles définissent un ordre partiel sur les événements**
 - tous les vecteurs ne sont pas comparables
 - $V \parallel V'$ dénote 2 vecteurs V et V' non comparables
- Les horloges vectorielles caractérisent la causalité (consistance forte):**
 - Soit $VC(e)$ l'horloge vectorielle d'un événement e
 - $e \rightarrow e' \Leftrightarrow VC(e) < VC(e')$
 - $e \parallel e' \Leftrightarrow VC(e) \parallel VC(e')$

Simplification du test de causalité

- Si e a lieu sur le site i , e' sur le site j :**

$$e \rightarrow e' \Leftrightarrow VC(e)[j] < VC(e')[j]$$

$$e \parallel e' \Leftrightarrow VC(e)[i] > VC(e')[i] \text{ et } VC(e')[j] > VC(e)[j]$$



Horloges vectorielles : LA solution ?

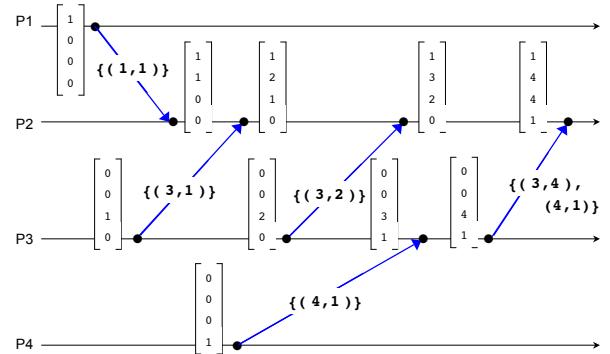
- Caractérisent la causalité mais...
 - Augmentent la quantité d'information générée localement
 - Augmentent l'information circulant sur le réseau
 - Fonction du nombre de processus
 - Taille minimum : Charron-Bost a prouvé que la causalité des événements d'un système réparti avec N processus ne peut être caractérisé qu'avec des horloges vectorielles ayant au moins N entrées.
- Peut-on arriver au même résultat en manipulant moins d'information ?
 - On ne peut pas diminuer l'information générée localement
 - On peut optimiser l'information envoyée sur le réseau

Implémentation d'horloges vectorielles

Singhal / Kshemkalyani [92]

- **Principe : Gestion incrémentale des horloges vectorielles**
 - Dans un message envoyé par j à i , j n'inclut que les composantes de son horloge qui ont été modifiées depuis le dernier message qu'il a envoyé à i .
 - L'estampille du message est une liste de couples (id, val) tels que :
 - $VC_j[id]$ a été modifié depuis le dernier message envoyé par j à i .
 - La valeur courante de $VC_j[id]$ est val .
 - Pour tout couple (id, val) contenu dans le message de j :
 $VC_i[id] = \max(VC_i[id], val)$
- **Conditions d'application :**
 - Canaux FIFO

Exemple



Implémentation de la méthode S/K

- Quelle information maintenir sur chaque site ?**
 - Celle qui permet à j de savoir quelles composantes de son vecteur d'horloge envoyer à i :
 - Horloge vectorielle de dernière émission vers i
- Sous quelle forme ?**
 - Matrice ?
 - La colonne i est le vecteur d'horloge du dernier envoi à i
 - PB : stockage local en N²
 - Vecteurs ?
 - On peut stocker les dates d'émission sous forme scalaire : LS
 - On stocke aussi les dates (scalaires) de modification de chacune des composantes du vecteur d'horloge : LU

→ 2 vecteurs sur chaque site en plus de l'horloge vectorielle

Implémentation (suite)

- Pourquoi des dates scalaires ?**
 - Tous les événements à considérer ont lieu sur le même site j
 - ⇒ ils sont identifiés de manière unique par $VC_j[i]$
- Gestion des dates d'émission : vecteur LS**
 - LS : Last Sent
 - $LS[i] =$ valeur de $VC_j[i]$ lors du dernier envoi de j à i
- Gestion des dates de modification : vecteur LU**
 - LU : Last Update
 - $LU[k] =$ valeur de $VC_j[i]$ lors de la dernière modification de $VC_j[k]$

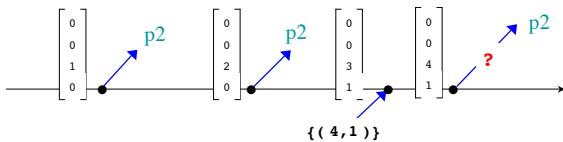
Identification des valeurs émises

- j envoie à i toutes les composantes k de VC_j telles que :

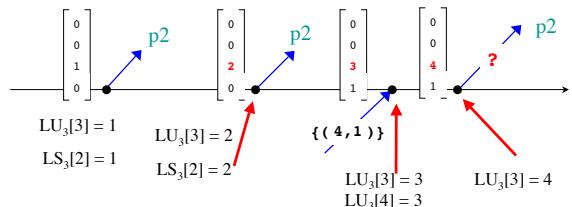
$$LU_j[k] > LS_j[i]$$

sous la forme $(k, VC_j[k])$

- Exemple : calcul des valeurs envoyées par p3**



Exemple



Valeur des vecteurs pour l'événement e_3^4 :

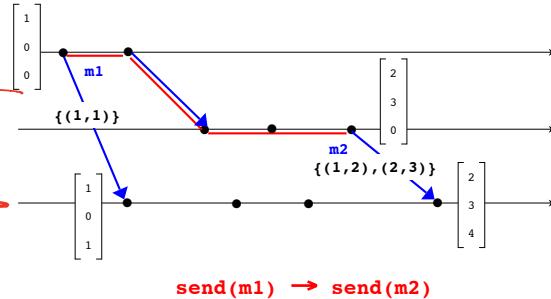
$$LS_3 = \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \end{bmatrix} \quad LU_3 = \begin{bmatrix} 0 \\ 0 \\ 4 \\ 3 \end{bmatrix} \quad VC_3 = \begin{bmatrix} 0 \\ 0 \\ 4 \\ 1 \end{bmatrix}$$

envoi de $\{ (3, VC_3[3]), (4, VC_3[4]) \}$

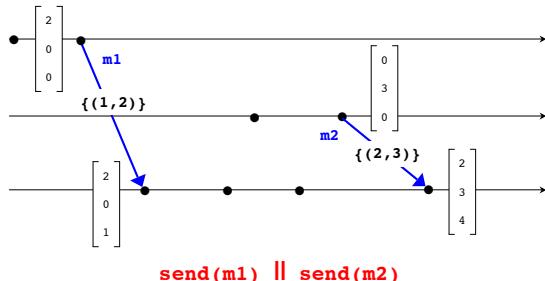
Efficacité de la méthode

- Efficace si les interactions entre processus sont localisées :**
 - Chaque processus ne communique qu'avec un petit nombre d'autres processus
 - le nombre d'entrées modifiées est faible
- Mais perte d'information au niveau de la causalité :**
 - Un processus qui reçoit deux messages en provenance d'émetteurs différents ne peut pas établir la relation de causalité entre leurs émissions
 - ce n'est pas forcément un problème...

Perte d' information sur la dépendance causal entre émissions de message



Perte d' information sur la dépendance causal entre émissions de message (suite)



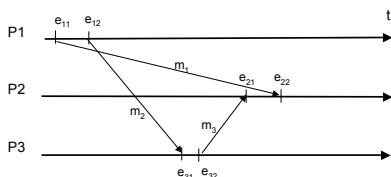
Dépendance causale entre messages

- On considère que $send(m)$ et $receive(m)$ correspondent l'émission et la réception d'un message m et que les messages doivent respecter la dépendance causale.

Définition :

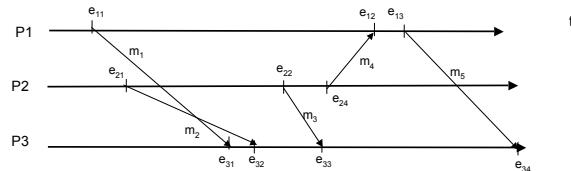
Les communications sont ordonnées causalement ("Causal Ordered" - CO), si, pour tout message m et m' , on a :
 $send(m) \rightarrow send(m') \Rightarrow receive(m) \rightarrow receive(m')$

Dépendance causale entre messages : Exemple (1)



La réception sur P2 ne respecte pas la dépendance causale des messages :
 $\text{send}(m_1) \rightarrow \text{send}(m_3)$ mais $\text{receive}(m_1) \not\rightarrow \text{receive}(m_3)$

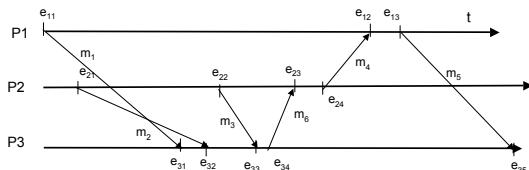
Dépendance causale entre messages : Exemple (2)



Toutes les communications ne sont pas ordonnées causalement :
 $\text{send}(m_3) \rightarrow \text{send}(m_4)$ mais $\text{receive}(m_3) \not\rightarrow \text{receive}(m_4)$
 $\text{send}(m_2) \rightarrow \text{send}(m_4)$ mais $\text{receive}(m_2) \not\rightarrow \text{receive}(m_4)$

Observation : les émissions de m₁ et m₂ ne sont pas en relation de précédence

Dépendance causale entre messages : Exemple (3)



Les communications sont ordonnées causalement

Observation : les émissions de m₁ et m₂ ne sont pas en relation de précédence

Conclusions

- La causalité est la relation qui permet d'analyser une application répartie
 - Pas d'exécutions identiques
 - Nécessité de tracer le lien entre les événements
- Le type d'horloge dépend de l'application
 - ↳ Consistance forte ? Horloges vectorielles
 - ↳ Consistance simple ? Horloges scalaires
 - ↳ Cotemporel
- Approche Foulger/Zwonepoel
 - Dépendance Directe

Bibliographie

- Lamport, L. *Time, clocks, and the ordering of events in a distributed system.* Communications of the ACM, 21(7), 1978.
- Mattern, F. *Virtual Time and Global States of Distributed Systems.* In Proceedings. Workshop on Parallel and Distributed Algorithms, 1989.
- C. Fidge. *Logical time in distributed computing systems.* IEEE Computer, pages 28-33, July 1991.
- M.Singhal, A. Kshemkalyani. *An efficient implementation of vector clocks,* Information Processing Letters, 992, vol. 43, no1, pp. 47-52.
- M. Raynal, M. Singhal. *Logical time: a way to Capture Causality in Distributed Systems.* Technical Report n. 2472, INRIA - Rennes,1995
- Bernadette Charron-Bost: *Concerning the Size of Logical Clocks in Distributed Systems.* Information. Processing Lett. 39(1): 11-16 (1991)

Exclusion Mutuelle en Réparti

18/02/10

AR: Exclusion mutuelle

1

Exclusion Mutuelle

■ Objectif:

- Coordonner des processus se partageant une ressource commune pour qu'à tout instant, au plus un processus ait accès à cette ressource.
- L'accès se fait dans une section critique (SC), dont les processus demandent l'entrée et signalent la sortie.

■ Exclusion Mutuelle en réparti:

- Les processus sont répartis et ne communiquent que par passage de messages.

18/02/10

AR: Exclusion mutuelle

3

Plan

■ Exclusion mutuelle en réparti

- Types d'algorithmes
- Propriétés
- Classes d'algorithmes
 - Algorithmes à permission
 - Lamport
 - Ricart-Agrawala
 - Maekawa (quorum)
 - Algorithmes à jeton
 - Martin
 - Naimi-Trehel
 - Susuki-Kasami

18/02/10

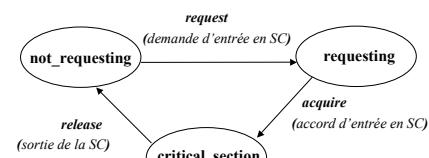
AR: Exclusion mutuelle

2

Transitions d'un processus

■ N processus: P_1, \dots, P_N

- état_i : {requesting, not_requesting, critical_section}



- **request** et **release**: fonctions invokées par les processus
- **acquire**: transition contrôlée par l'algorithme.

18/02/10

AR: Exclusion mutuelle

4

Algorithme d'Exclusion Mutuelle

■ Un algorithme d'exclusion mutuelle doit garantir :

- Au plus un processus exécute la section critique à un instant donné.
- Pas d'**interblocage**
 - Si des processus demandent concurremment à entrer en section critique, la sélection ne peut pas être ajournée indéfiniment.
- Pas de **famine**
 - La demande d'un processus ne peut pas être différée indéfiniment. Autrement dit, un processus qui demande à entrer en section critique doit être autorisé à le faire dans un temps fini.

18/02/10

AR: Exclusion mutuelle

5

Exclusion mutuelle en réparti

■ Le contexte réparti :

- N sites (nœuds ou processus).
- Pas de mémoire globale partagée ni d'horloge physique globale.
- Les sites communiquent par passage de messages, qui sont envoyés et reçus sur des canaux.
 - Aucune hypothèse temporelle n'est faite pour le délai de transmission des messages.
- Les canaux:
 - Fiables.
 - "FIFO" ou non "FIFO", selon les hypothèses de l'algorithme.

18/02/10

AR: Exclusion mutuelle

7

Propriétés à assurer

Les propriétés d'un algorithme se classent en deux catégories:

- **sûreté (safety) :**
 - jamais rien de "mauvais" n'arrive
- **vivacité (liveness) :**
 - quelque chose de "bien" finit par arriver

■ Algorithme d'exclusion mutuelle

- **Sûreté :**
 - à tout instant il y a au plus un processus dans la section critique.
 - Si $(etat_i = critical_section)$ alors $(etat_j \neq critical_section)$ pour tout $j < i$
- **Vivacité :**
 - Tout processus qui demande la section critique doit l'obtenir au bout d'un temps fini.
 - $(etat_i = requesting) \rightarrow (etat_i = critical_section)$
 - Garantir l'absence d'interblocage et de famine.
 - Propriétés auxquelles on ajoute l'**équité**.

18/02/10

AR: Exclusion mutuelle

6

Algorithme centralisé x réparti

■ Algorithme centralisé - coordinateur

- Le processus coordinateur est le seul à prendre une décision sur les accès à la section critique.
- Tous les informations nécessaires pour l'algorithme sont concentrées dans le coordinateur.

■ Solutions entièrement réparties :

- Tous les processus peuvent participer à la décision sur l'accès à la section critique.
- Les informations pour réaliser l'algorithme sont réparties entre les processus.

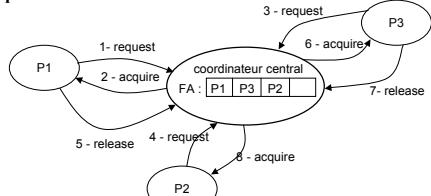
18/02/10

AR: Exclusion mutuelle

8

Algorithme centralisé

- Tous les processus s'adressent à un coordinateur pour demander l'entrée et signaler la sortie de SC. Le coordinateur maintient une file d'attente (FA) dans laquelle il range par ordre d'arrivée les requêtes d'entrée en SC.



18/02/10

AR: Exclusion mutuelle

9

Algorithme centralisé (Evaluation)

■ Nombre de Messages par exécution de SC :

- 3 messages.

■ Equitable :

- requêtes traitées par ordre d'arrivée sur le coordinateur.

■ Avantages :

- simplicité (en fonctionnement normal).
- faible complexité en messages.

■ Inconvénients :

- goulot d'étranglement sur le coordinateur.
- panne du coordinateur relativement complexe à résoudre.

18/02/10

AR: Exclusion mutuelle

10

Algorithmes répartis

■ Classes d'algorithmes

- A base de permission :
 - Afin d'entrer en section critique, un processus P_i doit demander la permission à d'autre processus.
 - Le droit d'entrée en SC est acquis lorsque le processus a obtenu un nombre suffisant de permissions.
- A base de jeton :
 - Seul le processus possédant le jeton peut entrer en section critique.
 - L'unicité du jeton garantit la propriété de sûreté
 - Différentes façons de réaliser la vivacité:
 - informer le site qui possède le jeton des requêtes en cours.
 - assurer le routage du jeton vers les processus demandeurs.

18/02/10

AR: Exclusion mutuelle

11

Algorithmes à base de Permission

■ Lamport

■ Ricart-Agrawala

■ Maekawa (quorum)

18/02/10

AR: Exclusion mutuelle

12

Algorithme à Base de Permission

Lamport (1978) et Ricart/Agrawala (1981)

- Un message de demande d'entrée en SC est envoyé à tous les autres sites R_i
- $R_i = \{1, 2, 3, \dots, N\} - \{i\}$
- Ordre total des requêtes d'entrée en section critique :
- Les requêtes sont totalement ordonnées et satisfaites selon cet ordre.
- La date d'une requête est la valeur de l'**horloge logique scalaire** du processus émetteur P_p , complétée par son identifiant : (H_p, i) .
 - $(H_i, i) < (H_j, j) \Leftrightarrow (H_i < H_j \text{ ou } (H_i = H_j \text{ et } i < j))$
- Garantie de la **sûreté** et de la **vivacité**.
- Chaque processus P_i gère une horloge logique, une file d'attente FA_i de requêtes **classées par date** et les attentes de permission At_i .

18/02/10

AR: Exclusion mutuelle

13

Algorithme de Lamport

Hypothèses :

- Le nombre N de processus est connu de tous.
- Les **canaux** de communication sont fiables et **FIFO**.

Messages :

- **Types :**
 - *REQUEST* : demande d'entrer en SC.
 - *REPLY* : réponse à la réception d'un message *REQUEST*.
 - *RELEASE* : libération de la SC.
- **Contenu:**
 - $(type, (H_i, S_i))$;

Variables Locales du processus P_i :

- H_i : Horloge logique scalaire
- FA_i : File d'attente de requêtes
 - Dans l'ordre induit par la valeur de leurs estampilles (y compris celle de P_i)
- At_i : Attente de permission.

18/02/10

AR: Exclusion mutuelle

14

Algorithme de Lamport

Tous les processus S_i :

Variables Locales:

$FA_i = \emptyset$;
 $H_i = 0$;
 $At_i = \emptyset$;

Request_CS(S_i) :

- Placer sa requête req dans la file d'attente;
- Envoyer un message *REQUEST* à tous les autres sites ($At_i = R_i - \{S_i\}$);
- Attendre l'accord de tous les autres sites (msg *REPLY*) et que sa propre requête soit la plus ancienne de toutes ($At_i = \emptyset$ et $req = head(FA_i)$);

Release_SC(S_i) :

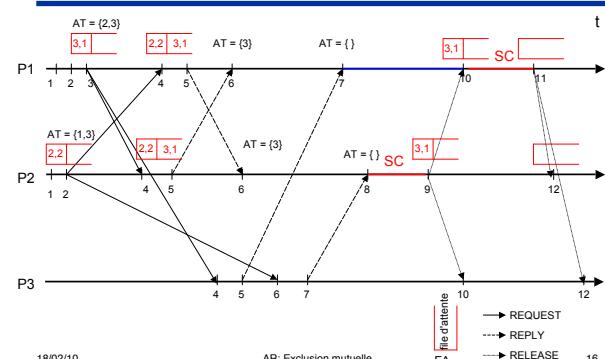
- H_i++ ;
- Diffuser un message *RELEASE* à tous les autres sites ($R_i - \{S_i\}$);
- Enlever sa requête req de la file d'attente FA_i ;
- Reception (msg de S_j) :**
 - Mettre à jour H_i : $(H_i = max(H_i, H_j) + 1)$
 - Switch (type msg) :
 - REQUEST : - placer la requête reçue dans la file d'attente FA_i dans l'ordre des estampilles : $(FA_i, U \{msg\} S_j)$;
 - envoyer un message *REPLY* à S_j .
- REPLY: - traiter la réception de l'acquittement ($At_i - \{S_j\}$)
- RELEASE: - Enlever la requête de S_j de la file d'attente ($FA_i - \{msg\} S_j\}$);

18/02/10

AR: Exclusion mutuelle

15

Algorithme de Lamport (exemple)



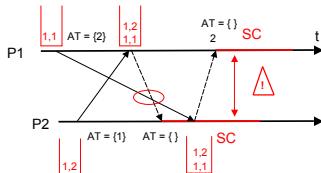
18/02/10

AR: Exclusion mutuelle

16

Algorithme de Lamport

- Si les canaux ne sont pas FIFO, l'exclusion mutuelle n'est pas garantie.



- Propriété FIFO de canaux garantie:
 - Si un site S_i a reçu un message d'accord (REPLY) de S_j , toute requête antérieure de S_j lui est forcément arrivée. Toute demande lui arrivant de S_j sera postérieure à la sienne.

18/02/10

AR: Exclusion mutuelle

17

Algorithme de Lamport

- L'ordre total sur les demandes garantit :

- La sûreté:

- Seul le site en tête de la file d'attente FA pourra rentrer en SC; les autres attendent que cette demande soit retirée (réception du message RELEASE).

- La vivacité:

- Toute demande finira par avoir la plus petite estampille et donc se trouvera en tête de la file d'attente.

18/02/10

AR: Exclusion mutuelle

18

Algorithme de Lamport (Evaluation)

- Nombre de Messages par exécution de SC:

- 3* (N-1) messages.

- Equitable :

- requêtes traitées par l'ordre total.

- Avantages :

- simplicité (en fonctionnement normal).

- Inconvénients :

- Hypothèse de canaux FIFO.

- Pas extensible.

18/02/10

AR: Exclusion mutuelle

19

Algorithme Ricart/Agrawala

- Amélioration de l'algorithme de Lamport:

- Message REPLY : possède le sens d'une autorisation d'accès, délivrée de façon conditionnelle. Un processus P_i n'acquitte une requête que si l'il n'est pas en SC et sa requête en cours n'est pas plus prioritaire.

- Message RELEASE : n'est envoyé qu'aux processus dont la requête a été différée. Remplacé par le message REPLY.

- File d'attente : chaque processus P_i ne conserve dans sa file d'attente FA_i que les requêtes dont l'acquittement a été différé.

18/02/10

AR: Exclusion mutuelle

20

Algorithme de Ricart/Agrawala

■ Hypothèses :

- Le nombre N de processus est connu de tous.
- Les canaux de communication sont fiables, mais pas FIFO.

■ Messages :

- Types :
 - REQUEST* : demande d'entrer en SC.
 - REPLY* : réponse à la réception d'un message *REQUEST*.

- Contenu :
 - (type, (H_i, S_i));

■ Variables Locales du processus P_i :

- H_i : Horloge logique scalaire
- FA_i : File d'attente
- At_i : Attente de permission.
- Etat_i : *requesting*, *not_requesting*, *critical_section*.

■ Algorithme

- Sera vu en TD et TME.

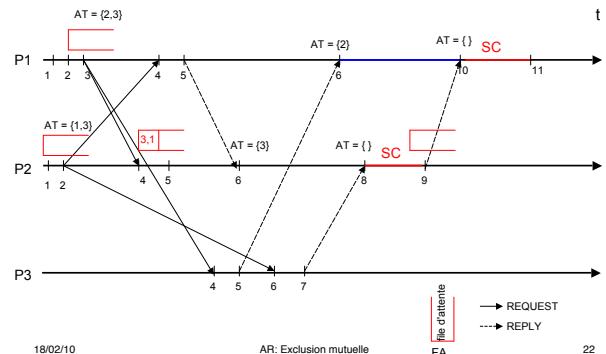
18/02/10

AR: Exclusion mutuelle

21

Algorithme de Ricart/Agrawala (exemple 1)

Algorithme de Ricart/Agrawala (exemple 1)

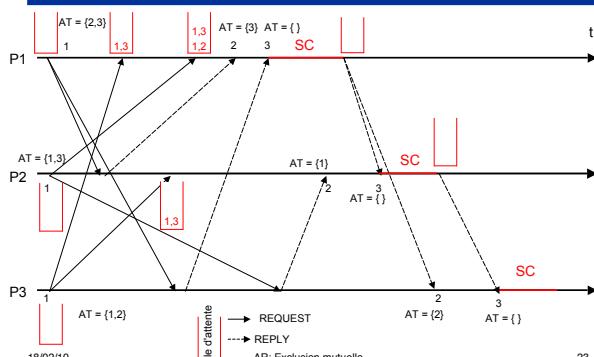


18/02/10

AR: Exclusion mutuelle

FA

22



18/02/10

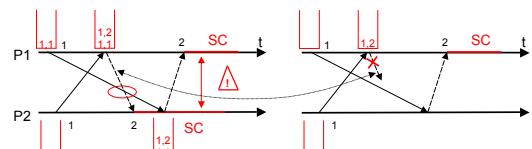
AR: Exclusion mutuelle

23

Algorithme de Ricart/Agrawala

■ Hypothèse FIFO n'est plus nécessaire

- les messages *REPLY* valent autorisation d'entrée en SC. Quand un processus a reçu tous les msg. *REPLY*, il n'y a plus de requête plus récente en cours.



18/02/10

AR: Exclusion mutuelle

24

Algorithme de Ricart/Agrawala (Evaluation)

- Nombre de Messages par exécution de SC :
 - $2*(N-1)$ messages.
- Equitable :
 - requêtes traitées par l'ordre total.
- Avantages par rapport à Lamport:
 - moins de messages envoyés.
 - taille de la file d'attente FA plus petite.
 - hypothèse FIFO non nécessaire.

18/02/10

AR: Exclusion mutuelle

25

Algorithme de Maekawa (quorum)

N= nombre de sites

K_i= nombre de sites dans RS_i

D= nombre d'ensembles auquel chaque site appartient

- Afin de minimiser le trafic des messages et de demander le même effort à tous les sites:

- |RS₁| = |RS₂| = |RS₃| ... = |RS_N| = K
- $\forall S_i \in \{S_1, \dots, S_N\}, S_i \in RS_i$
- $\forall i, j \in \{1, \dots, N\}$ tels que $i \neq j$,
 S_i et S_j appartiennent à $D RS$
/* même nombre d'ensembles */
- D = K est une possibilité

18/02/10

AR: Exclusion mutuelle

27

Algorithme de Maekawa (quorum)

- Chaque site ne peut donner sa permission qu'à un seul à la fois
 - Arbitrer un certain nombre de conflits
- Message REQUEST n'est pas diffusé à tous les sites :
 - Chaque site S_i appartient à un ensemble (quorum) RS_i (Request Set) dont il doit obtenir l'accord (msg LOCKED) de tous les membres pour pouvoir entrer en SC.
 - Il doit y avoir au moins un site commun entre deux ensembles RS_i et RS_j. Ce site arbitre les conflits.

$$\forall i, j \in \{1, \dots, N\} \text{ tels que } i \neq j, RS_i \cap RS_j \neq \emptyset \quad (1)$$

18/02/10

AR: Exclusion mutuelle

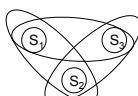
26

Algorithme de Maekawa (quorum)

- Exemples de quorum

$$\begin{aligned} RS_1 &= \{S_1, S_2, S_3\} \\ RS_2 &= \{S_2, S_4, S_6\} \\ RS_3 &= \{S_3, S_5, S_8\} \end{aligned}$$

N=3, K=2



$$\begin{aligned} RS_1 &= \{S_1, S_2, S_4\} \\ RS_2 &= \{S_2, S_4, S_6\} \\ RS_3 &= \{S_3, S_5, S_8\} \\ RS_4 &= \{S_4, S_1, S_5\} \\ RS_5 &= \{S_5, S_2, S_7\} \\ RS_6 &= \{S_6, S_1, S_3\} \\ RS_7 &= \{S_7, S_3, S_4\} \end{aligned}$$

N=7, K=3

18/02/10

AR: Exclusion mutuelle

28

Algorithme de Maekawa

- Pour entrer en SC, le site S_i doit verrouiller tous les membres de son ensemble RS_i en leur envoyant un message du type REQUEST.

- En recevant un msg REQUEST de S_j , si S_i ne se trouve pas déjà verrouillé, S_i envoie son accord (msg *LOCKED*) à S_j et se verrouille au profit de S_j .
 - S_i ne peut se verrouiller qu'au profit d'un seul site.
 - Si S_i arrive à verrouiller tous les membres de RS_p , aucun autre site ne pourra faire la même chose à cause de la propriété (1) – *intersection des ensembles*.
 - S_i rentre alors en SC. En sortant, S_i envoie un msg *RELEASE* à tous les membres de RS_p .

18/02/10

AB: Exclusion mutuelle

29

Algorithme de Maekawa

■ Solution pour le problème d'interbloquage

- Dater les messages : ordre total
 - Horloge de Lamport + identifiant
 - Reprendre la permission accordée si la nouvelle demande est antérieure à celle déjà satisfaitte
 - Si le site qui possède la permission sait qu'il n'est pas en mesure de recevoir tous les accords de son ensemble, il libère la permission obtenue.
 - Deux nouveaux types de messages :
 - *INQUIRE* : demande de la possibilité de reprendre la permission.
 - *RELINQUISH* : libération de la permission (verrou).

18/02/10

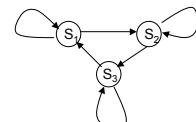
AB: Exclusion mutuelle

31

Algorithme de Maekawa

■ Risque d'interblocage :

- Le fait qu'un arbitre ne donne sa permission qu'à un seul demandeur (ne se verrouille qu'au profit d'un seul site) conduit à des situations d'interblocage.



$$\begin{aligned} RS_1 &= \{S_1, S_2\} \\ RS_2 &= \{S_2, S_3\} \\ RS_3 &= \{S_3, S_1\} \end{aligned}$$

N=3 K=2

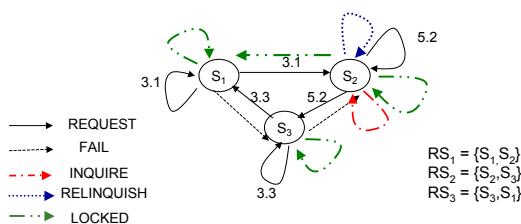
18/02/10

AB: Exclusion mutuelle

30

Algorithme de Maekawa

■ Solution pour le problème d'interblocage



$$\begin{aligned} RS_1 &= \{S_1, S_2\} \\ RS_2 &= \{S_2, S_3\} \\ RS &= \{S_1, S_2, S_3\} \end{aligned}$$

N=3 K=3

18/02/10

AB: Exclusion mutuelle

32

Algorithme de Maekawa

■ Contenu des messages :

- (type, (H_i, S_j)) : messages estampillés

■ Types de messages :

➤ REQUEST

- Demande d'entrée en SC. S_i envoie un tel message à tous les membres de son ensemble RS_i .

➤ RELEASE

- Envoyé par un site S_i à tous les membres de son ensemble RS_i lorsqu'il sort de la SC.

➤ LOCKED

- Envoyé par un site S_i en réponse à un message REQUEST de S_j , s'il ne l'a pas encore envoyé à un autre site. S_i se trouve alors verrouillé au profit de S_j .

18/02/10

AR: Exclusion mutuelle

33

Algorithme de Maekawa

■ Types de messages (cont) :

➤ FAIL

- Envoyé par un site S_i en réponse à un message REQUEST de S_j , s'il ne peut pas donner son accord (S_i se trouve déjà verrouillé). Le message de S_j est moins prioritaire et sera mis dans la file d'attente.

➤ INQUIRE

- Envoyé par un site S_i à S_j pour tenter de récupérer la permission accordée à S_j (S_j était verrouillé au profit de S_i).

➤ RELINQUISH

- Réponse à un message du type INQUIRE afin de rendre une permission non utilisable.

18/02/10

AR: Exclusion mutuelle

34

Algorithme de Maekawa

Grandes Lignes

Pour tous les processus S_i :

Request_CS()

- H_i++ ;
- $\forall j \in RS_i$ envoyer un message REQUEST à j ;
- $At_i = RS_i$;
- $\forall j \in RS_i$, attendre la réception d'un msg. LOCKED : ($At_i = \emptyset$);

Release_CS()

- $Hi++$;
- $\forall j \in RS_i$ Envoyer un message RELEASE à j ;

18/02/10

AR: Exclusion mutuelle

35

Variables Locales:

$$\begin{aligned} FA_i &= \emptyset; \\ H_i &= 0; \\ At_i &= \emptyset; \end{aligned}$$

Algorithme de Maekawa

Reception (msg de S_j) :

➤ REQUEST :

- Mettre à jour Hi ($H_i = \max(H_i, H_j) + 1$);
- Si S_i n'est pas verrouillé {
 - Verrouiller S_i au profit de S_j ;
 - Envoyer à S_j un message LOCKED ;}

}
sinon /* S_i verrouillé au profit de S_k */ {
• $(FAi \cup \{\text{msg } S_j\}) / \text{ inserer la demande dans la file d'attente dans l'ordre} */$
• Si la demande de S_k ou une autre dans la file FA_i est antérieure à celle de S_j

- envoyer un message FAIL à S_j

sinon

- si un message de INQUIRE n'a pas encore été envoyé à S_k
 - envoyer un message INQUIRE à S_k

}
• **LOCKED :**
• $(At_i = At_i - \{S_j\}) / \text{ comptabiliser la réception d'une permission en plus */}$

18/02/10

AR: Exclusion mutuelle

36

Algorithme de Maekawa

INQUIRE :

Si un message du type FAIL a été reçu

Envoyer message RELINQUISH à S_j : ($At=At_i, U(S_j)$);

RELINQUISH :

libérer le verrou ;

$FA_i \cup \{S_j\}$; / ajouter la requête de S_j dans la file dans l'ordre*/

se verrouiller au profit de $S_{k'}$, le site qui se trouve en tête de la file;

$FA_i - \{S_j\}$; / *retirer la requête S_j de la file d'attente */

envoyer un message LOCKED à $S_{k'}$;

RELEASE :

libérer le verrou;

se verrouiller au profit de $S_{k'}$, le site qui se trouve en tête de la file;

$FA_i - \{S_{k'}\}$; / *retirer la requête $S_{k'}$ de la file d'attente */

envoyer un message LOCKED à $S_{k'}$;

FAIL :

enregistrer la réception d'un échec d'accord de la part de S_j ;

Si INQUIRE de S_k , pendant

envoyer msg RELINQUISH à S_k ($At=At_i, U(S_k)$);

}

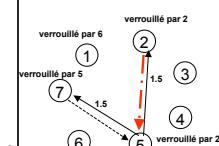
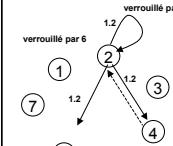
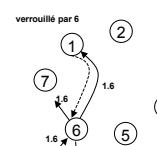
18/02/10

AR: Exclusion mutuelle

37

Algorithme de Maekawa

■ Exemple



→ REQUEST

----- → LOCKED

→ FAIL

RS₂ = {S₂, S₄, S₃}

RS₅ = {S₅, S₂, S₇}

RS₆ = {S₆, S₁, S₇}

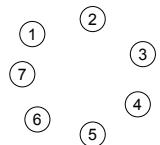
18/02/10

AR: Exclusion mutuelle

39

Algorithme de Maekawa

■ Exemple



RS₁ = {S₁, S₂, S₃}
RS₂ = {S₂, S₄, S₃}
RS₃ = {S₃, S₁, S₆}
RS₄ = {S₄, S₁, S₇}
RS₅ = {S₅, S₂, S₇}
RS₆ = {S₆, S₁, S₇}
RS₇ = {S₇, S₃, S₄}

N=7, K=3

Sites 2,5 et 6 exécutent Request_CS
H₂, H₅ et H₆ = 1

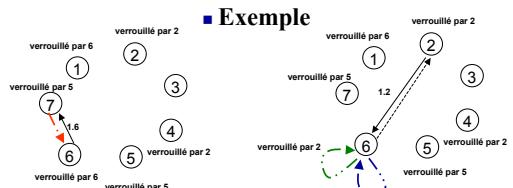
18/02/10

AR: Exclusion mutuelle

38

Algorithme de Maekawa

■ Exemple



→ REQUEST

----- → LOCKED

→ FAIL

→ INQUIRE

→ RELINQUISH

Site 6 reçoit msg REQUEST du site 2

Site 6 envoie Msg INQUIRE au site 6

Site 6 libère le verrou – msg RELINQUISH au site 6

Site 6 envoie msg LOCKED au site 2

Site 2 rentre en SC

18/02/10

AR: Exclusion mutuelle

40

Algorithme de Maekawa

■ Nombre de Messages par exécution de SC : $O(\sqrt{N})$

- Faible demande : $3*(K-1)$
 - (K-1) msg REQUEST + (K-1)msg LOCKED + (K-1)msg RELEASE
- Forte demande : $5*(K-1)$
 - (K-1) msg REQUEST + (K-1)msg LOCKED + (K-1)msg RELEASE + (K-1)*msg INQUIRE + (K-1)*RELINQUISH
- La valeur de K est approximativement égale à \sqrt{N}
 - Nombre de message entre $3*\sqrt{N}$ et $5*\sqrt{N}$

■ Avantages:

- Si pas de conflit, moins de messages envoyés par rapport à Lamport et Ricart-Agrawala.

■ Inconvénients

- Possibilité d'interblocage
- Construction des ensembles

18/02/10

AR: Exclusion mutuelle

41

Algorithmes à base de Jeton

■ Martin (anneau)

■ Susuki/Kasami (graphe complet)

■ Naimi-Trehel (arbre)

18/02/10

AR: Exclusion mutuelle

42

Algorithme à base de jeton

■ La permission pour rentrer en section critique est réalisée par la possession d'un jeton.

- L'unicité du jeton assure la sûreté.

■ Algorithmes doivent mettre en oeuvre la vivacité

- Déplacement du jeton

■ Mouvement perpétuel du jeton

- Lorsque le jeton arrive sur un site, il passera au suivant si le site est dans l'état *not_requesting*; si le site est dans l'état *requesting*, il passe à l'état *section_critique* et rentre en section critique.

▫ Exemple: anneau de communication (garantie de la vivacité).

■ Envoi de requêtes

- Anneau : Martin
- Arborescence: Naimi/Trehel
- Diffusion : Suzuki/Kasami

18/02/10

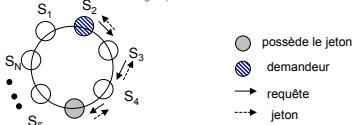
AR: Exclusion mutuelle

43

Algorithme de Martin (anneau)

■ Sites organisés en anneau logique statique

- Jeton circule dans le sens inverse des requêtes.
- Un site demandeur entre en SC critique lorsqu'il possède le jeton
- Quand S_i veut entrer en section critique, il envoie une requête à son successeur, $S_{(i+1) \bmod N}$ et attend le jeton. En recevant une requête de son prédécesseur, si S_j ne possède pas le jeton, il retransmet la requête à son successeur $S_{(j+1) \bmod N}$. Sinon, s'il le possède et ne l'utilise pas, il l'envoie à son prédécesseur $S_{(j-1) \bmod N}$.



18/02/10

AR: Exclusion mutuelle

44

Algorithme de Martin Evaluation

■ Nombre de Messages par exécution de SC :

- Si $K = \text{nombre de sites entre } S_i (\text{site qui demande la SC}) \text{ et le site } S_p (\text{site qui possède le jeton}), \text{ alors :}$
 - Nb messages = $2*(K+1)$;

■ Avantages :

- Simplicité.
- Pas de diffusion.

■ Inconvénients :

- Pas extensible.
- un site qui n'est pas intéressé par la section critique est souvent sollicité à transmettre les requêtes et le jeton.

18/02/10

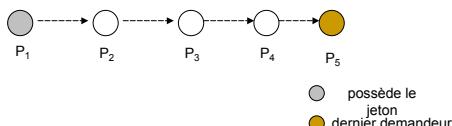
AR: Exclusion mutuelle

45

Algorithme de Naimi/Trehel

■ File de requêtes : "next"

- Processus en tête de la file possède le jeton.
- Le processus à la fin de la file est le dernier processus qui a fait une requête pour entrer en section critique.
- Une nouvelle requête est toujours placée en fin de la file.



18/02/10

AR: Exclusion mutuelle

47

Algorithme de Naimi/Trehel arborescence

■ Deux structures de données:

- File de requêtes : "next"

- Arbre de chemins vers le dernier demandeur : "father"

18/02/10

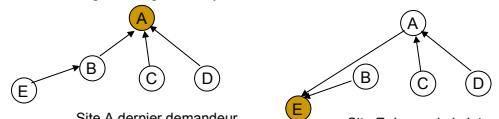
AR: Exclusion mutuelle

46

Algorithme de Naimi/Trehel

■ Arbre de chemins vers le dernier demandeur : "father"

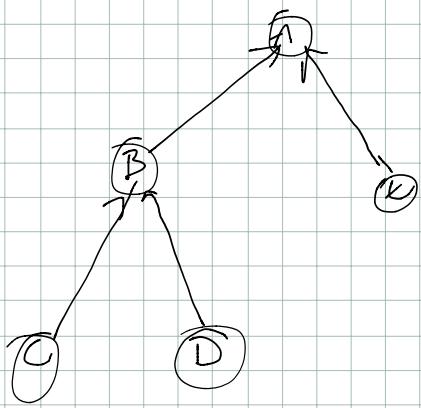
- Racine de l'arbre : dernier demandeur (dernier élément de la file des "next").
- Une nouvelle requête est transmise à travers un chemin de pointeurs "father" jusqu'à la racine de l'arbre (*father = nil*).
 - Reconfiguration dynamique de l'arbre. Le nouveau demandeur devient la nouvelle racine de l'arbre.
 - Les sites dans le chemin compris entre la nouvelle et l'ancienne racine changent leur pointeur "father" vers la nouvelle racine.



18/02/10

AR: Exclusion mutuelle

48



Algorithme de Naimi/Trehel

Local Variables:
 Token : boolean;
 requesting; boolean
 next, father: 1.. N U {nil}

Initialisation de S_i :
 $father = S_i$; $next = nil$;
 $requesting = false$;
 $Token = (father == S_i)$
 if ($father == S_i$)
 $father = nil$;

Request_CS (S_i):
 $requesting = true$;
 if ($father <> nil$) {
 send (Request, S_j) to father;
 $father = nil$;
 }
 attendre ($Token == true$);

Release_CS (S_i):
 $requesting = false$;
 if ($next <> nil$) {
 send (Token) to next;
 $Token = false$;
 $next = nil$;
 }

18/02/10

AR: Exclusion mutuelle

49

Algorithme de Naimi/Trehel (cont)

Receive_Request_CS(S_j):
 if ($father == nil$) {
 if ($requesting$)
 $next = S_j$;
 else { $token = false$;
 send (Token) to S_j ;
 }
 }
 else
 send (Request, S_j) to father;
 $father = S_j$;

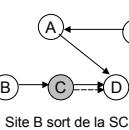
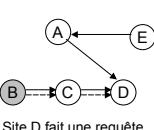
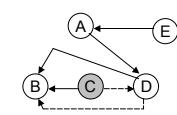
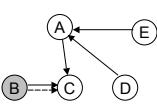
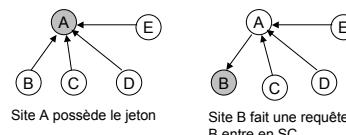
Receive_Token (S_j):
 $Token = true$;

18/02/10

AR: Exclusion mutuelle

50

Algorithme de Naimi/Trehel (Exemple)



18/02/10

AR: Exclusion mutuelle

51

Algorithme de Naimi/Trehel (Evaluation)

■ Nombre de Messages par exécution de SC :

- Entre 0 et N par demande
- Moyenne : $O(\log N)$.

■ Avantages :

- Extensibilité : $O(\log N)$.
- un site qui n'est pas intéressé par la section critique ne sera plus sollicité après quelques transferts de requêtes "adaptativité".

18/02/10

AR: Exclusion mutuelle

52

Algorithme de Susuki/Kasami (diffusion)

- Pour entrer en SC, un processus diffuse une demande de jeton à tous les autres processus.
 - Si le processus qui possède le jeton n'est pas en SC, il renvoie immédiatement le jeton au processus demandeur. Sinon, il attend la sortie de la SC et envoie le jeton au premier processus dont la requête n'a pas été satisfaite.
 - Les requêtes pendantes sont transmises dans le message du jeton en respectant l'ordre FIFO.
- Chaque processus gère un compteur des requêtes qu'il a effectuées et une table des requêtes effectuées par les autres processus.
- Le jeton est un message particulier, unique, contenant la table des requêtes satisfaites et un file d'attente de requêtes pendantes.

18/02/10

AR: Exclusion mutuelle

53

Algorithme de Susuki/Kasami

■ Variables:

- **Etat_i** : requesting, not_requesting, critical_section.
- **Token_i** : indique la présence du jeton sur le site S_i.
- **RN_i**: vecteur de N positions :
 - RN_i[j] est le numéro de la dernière requête reçue de la part du site S_j.
 - RN_i[i] correspond au nombre de requêtes faites par le site S_i.
- **LN_i**: vecteur de N positions des requêtes satisfaites

18/02/10

AR: Exclusion mutuelle

55

Algorithme de Susuki/Kasami

■ Type de Message:

➢ REQUEST (S_j,k):

- k = (1,2,... N). Indique que site S_j est en train de faire sa kème demande d'entrée en section critique.

➢ TOKEN (Q, LN)

- Q : une file d'attente de demandes pour entrer en section critique des différents sites.
- LN : où LN[j] est le numéro de la dernière demande d'entrée en section critique du site S_j qui a été satisfaite.

18/02/10

AR: Exclusion mutuelle

54

Algorithme de Susuki/Kasami

Initialisation variables locales (S_i):

Token = (S_i == S_i)
Etat = not_requesting;
RN [i] = 0, j = 1, 2, ..., N;
LN [i] = 0, j = 1, 2, ..., N;
Q = Ø;

LN → dernière req satisfaite.

Release_CS (i):
RN[i] = RN[i];
for (site = 1; site <= N; site++) {
if ((site != i) && (site not in Q) &&
(RN[site] > LN[site]))
ajouter site à la fin de Q;
}

Request_CS (i):
Etat = requesting;
if (Token == false) {
RN[i] = RN[i] + 1;
diffuser REQUEST(S_i,RN[i]);
attendre (Token == true)
}
if (Q != Ø) {
Token = false;
site = extraire (Q); /*premier de la file
send TOKEN (Q,LN) to site ;
}
Etat = not_requesting;

18/02/10

AR: Exclusion mutuelle

56

Algorithme de Susuki/Kasami (cont)

Receive_Request_CS(S_j, REQUEST (j,k)):

```
RN[j] = max (RN[j], k);
if ((Token = true) && (Etat == not_requesting) && (RN[j] > LN [j])) {
    Token = false;
    send TOKEN (Q,LN);
}
```

Receive_Token (TOKEN (Q,LN)):

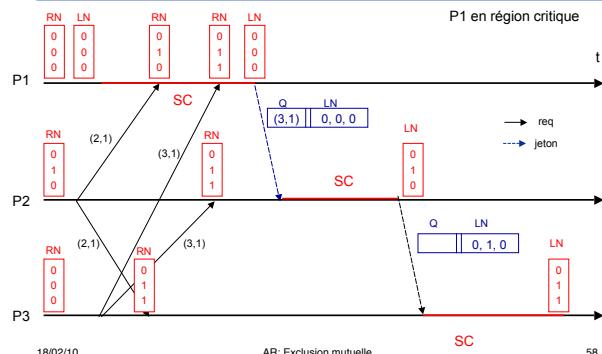
```
Token = true;
LN = TOKEN.LN ;
Q = TOKEN.Q;
```

18/02/10

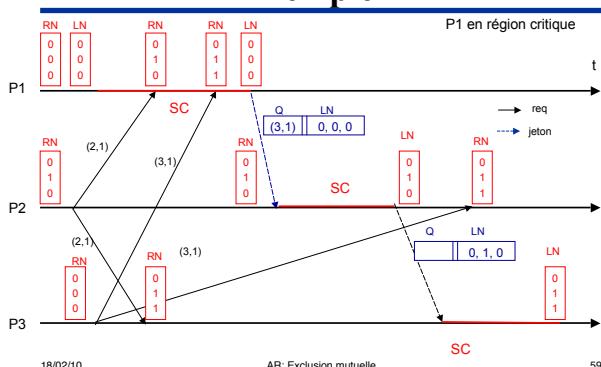
AR: Exclusion mutuelle

57

Algorithme de Susuki/Kasami Exemple 1



Algorithme de Susuki/Kasami Exemple 2



Algorithme de Susuki/Kasami (Evaluation)

■ Nombre de Messages par exécution de SC:

- N, si le processus n'a pas le jeton.
- 0, si le processus a le jeton

■ Vivacité

- Garantie par l'ordre FIFO de la file Q

■ Inconvénient :

- Pas extensible

18/02/10

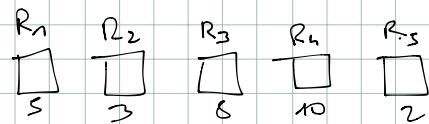
AR: Exclusion mutuelle

60

Bibliographie

- Lamport, L. *Time, clocks and the ordering of events in a desitributed system*, Communications of the ACM, vol. 21, no. 7, july 1978, pages 558-565.
- Ricart, G. and Agrawala, A. *An optimal algorithm for mutual exclusion en computer networks*, Communications of the ACM, vol. 24, no. 1, jan 1981, pages 9-17.
- Suzuki, I. and Kasami, T. *A distributed mutual exclusion algorithm*, ACM Transactions on Computer Systems, vol. 3, no. 4, nov. 1985, pages 344-349.
- Naimi, M. and Trebel, M. *A Log (N) distributed mutual algorithm based on the Path Reversal*, Journal of Parallel and Distributed Computing vol. 34 no. 1, avril 1996, pages 1-13.
- Raynal, M. *Synchronisation et Etat Global dans les Systèmes Répartis*, 1992.
- MARTIN, A.J. Distributed Mutual Exclusion on a Ring of Processes, *Science of Computer Programming*, vol5, pp-265-276, Feb. 1985.
- MAEKAWA, M. *A sgrtn(n) algorithm for mutual exclusion in descentralized systems*, ACM Transaction on Computer Systems, vol. 3, no.2, mai 1985, pages 145-159
- ALBERT,A. and SANDLER, R. *An Introduction to Finite Projective Planes*, Holt, Rinehart and Winston, NY, 1968.

Algorithm de Ra蒙de



$$P_1 \rightarrow R_1(2) \\ R_2(3)$$

$$P_2 \rightarrow R_2(5) \\ R_4(3) \\ R_5(7)$$

Transpositions dans
cours papier.

Algorithmes par vague

18/02/10

AR: Algorithmes par Vague

1

Plan

■ Définition Algorithme Total

■ Exemple d'algorithme

- Algorithme de l'anneau
- Algorithme de l'arbre
- Algorithme de l'Echo
- Algorithme de la Phase

Algorithme Total

■ Algorithme par Vague

- Les algorithmes par vague sont utilisés pour diffuser une information sur le réseau, découvrir la topologie du réseau, rassembler des informations du réseau.
- N nœuds
 - Un nœud ne se communique qu'avec ses voisins

Tous les nœuds du réseau participent avant qu'une décision soit prise

18/02/10

AR: Algorithmes par Vague

3

Algorithme Total

■ Un algorithme à Vague doit satisfaire les trois propriétés:

- *terminaison* : toute exécution est finie
- *décision* : une décision doit être prise à terme par au moins un processus.
- *dépendance* : Une décision est précédée causalement par un événement de chaque processus.

18/02/10

AR: Algorithmes par Vague

4

Algorithme Total

- Types de nœuds :
 - **Initiateur :**
 - nœud qui spontanément décide de démarrer l'algorithme
 - **Non-initiateur :**
 - Ne commence à exécuter qu'après avoir reçu un message.
- e_p = premier événement qui a lieu en p lors d'un exécution de l'algorithme :
 - **Nœud initiateur** : e_p est un événement interne ou l'envoi de message
 - **Nœud non-initiateur** : e_p est l'événement réception de message.
- **Observation :**
 - Pour un même algorithme, pour des exécutions différentes, le nœud initiateur peut être différent.

18/02/10

AR: Algorithmes par Vague

5

Algorithme Total

- Une décision est prise au plus une fois par un processus p .
 d_p = événement correspondant à la décision prise par p
- Définition (Tel) :** une exécution d'un algorithme est totale si au moins un processus p décide et pour tout $q \in N$ et pour tout p qui prend une décision $e_q \rightarrow d_p$.

Un algorithme est **total** ssi toutes ses exécutions possibles sont totales
- Dans un algorithme total sur un réseau de N nœuds, il y a au moins $N-1$ message échangés.

18/02/10

AR: Algorithmes par Vague

7

Algorithme Total

- **Caractéristiques :**
 - **Symétrie**
 - **Algorithme symétrique** :
 - L'algorithme est le même dans tous les nœuds
 - **Algorithme asymétrique**
 - L'algorithme du nœud *initiateur* n'est pas le même que celui d'un nœud non *initiateur*.
 - **Initialisation du calcul**
 - **Algorithme centralisé**
 - Il existe un unique processus qui est *initiateur* du calcul
 - **Algorithme décentralisé**
 - Pour tout sous-ensemble $\Pi_0 \subseteq \Pi$, \exists un calcul de l'algorithme dont Π_0 est l'ensemble des initiateurs.

18/02/10

AR: Algorithmes par Vague

6

Algorithmes par Vague

■ Notation (Tel) :

- **Événements :**
 - S_p : envoie message
 - R_p : réception message
 - D_p : décision
- $E_p : \{condition\}$
 - L'événement E_p est exécuté si la $\{condition\}$ est vraie.

18/02/10

AR: Algorithmes par Vague

8

1. L'algorithme de l'anneau

- Anneau unidirectionnel
- Algorithme asymétrique et centralisé
 - N noeuds
 - Les communications sont fiables, pas forcément FIFO, et tout message émis est reçu dans un temps fini mais arbitraire (modèle temporel asynchrone)
 - Un nœud ne connaît que l'identifiant de son successeur.
- Principe de l'algorithme :
 - Un seul initiateur à chaque exécution.
 - *Initiateur* envoie un jeton dans l'anneau.
 - Jeton doit être reçu par tous les nœuds.
 - *L'initiateur* décide lorsque le jeton lui est renvoyé.

18/02/10

AR: Algorithmes par Vague

9

1.L'algorithme de l'anneau

- Supposons :
 - $(j+1)\%N$ est le successeur de j
 - s_j : événement envoi de message du site j
 - r_j : événement réception de message du site j
 - e_j : premier événement de j
 - i initiateur
 - d_i : événement décision.
 - $e_i = s_i$ et $e_j = r_j$ pour tout $j \neq i$.
 - $r_j \rightarrow s_j$ pour tout $j \neq i$
 - $s_j \rightarrow r_{j+1}$ pour tout j
 - $r_i \rightarrow d_i$ (seule initiateur décide)

$$e_i = s_i \rightarrow r_{i+1} = e_{i+1} \rightarrow s_{i+1} \rightarrow r_{i+2} \dots \rightarrow r_j = e_j \rightarrow s_j \dots \rightarrow r_i = d_i$$

- Complexité en nombre de messages et temps : N

18/02/10

AR: Algorithmes par Vague

11

1.L'algorithme de l'anneau

Variable :
booléen Rec_p = false;
/*contrôle de la
réception du message*/

p initiateur :

S_p : { Spontanément, une fois}
envoie \leftrightarrow au successeur

R_p : { Un message \leftrightarrow arrive}
réception de \leftrightarrow ;
 Rec_p = true;

D_p : { Rec_p }
Décision

p non_initiateur :

R_p : { Un message \leftrightarrow arrive}
réception de \leftrightarrow ;
 Rec_p = true;

S_p : { **Recp**}
envoie \leftrightarrow au successeur;
 Rec_p = false;

\leftrightarrow : message vide.

18/02/10

AR: Algorithmes par Vague

10

2. L'algorithme de l'arbre

- N noeuds
 - Un nœud ne connaît que l'identifiant de ses voisins
- Liens bidirectionnels
- Algorithme symétrique et (**pas centralisé ni décentralisé**)
- Principe de l'algorithme
 - Un nœud qui a reçu un message de tous ses voisins sauf un envoi un message à celui-ci.
 - En ne possédant qu'un voisin, les **feuilles** de l'arbre sont des nœuds **initiateurs**
 - Toutes les feuilles, (possibilité de sauf une) doivent être des initiateurs.
 - Un nœud qui a reçu un message de tous ses voisins décide.

18/02/10

AR: Algorithmes par Vague

12

2. L'algorithme de l'arbre

Variables :

set $Vois_p$; /* ensemble de voisins de p^* /
 $boo Rec_p[q] = \text{false}$; $\forall q \in Vois_p$ /* contrôle réception message*//
 $boo Sent_p = \text{false}$; /* contrôle envoi d'un message*/

$R_p : \{ \text{Un message } \leftrightarrow \text{arrive de } q\}$
réception de \leftrightarrow ;
 $Rec_p[q] = \text{true}$;

$S_p : \{ \exists q \in Vois_p : \forall r \in Vois_p, r \neq q : Rec_p[r] \text{ et } !Sent_p\}$
envoie \leftrightarrow à q
 $Sent_p = \text{true}$;

$D_p : \{\forall q \in Vois_p : Rec_p[q]\}$
Décision

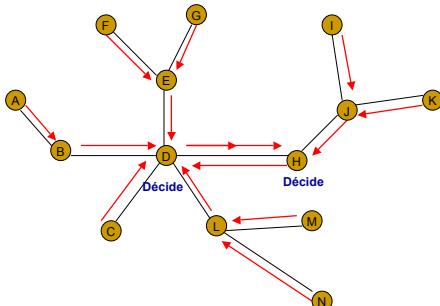
\leftrightarrow : message vide.

18/02/10

AR: Algorithmes par Vague

13

2. L'algorithme de l'arbre



18/02/10

AR: Algorithmes par Vague

14

2. L'algorithme de l'arbre

Théorème 1: Tant qu'un état permettant la décision n'est pas atteint, il y a toujours une émission ou une réception possible.

Preuve:

- A chaque liaison bidirectionnelle, on associe 2 bits r correspondant aux 2 sens d'émission, initialisés à 0. Le bit est mis à 1 quand le site à l'extrémité a reçu un message.
- Soit :
 - K - le nombre de sites ayant émis
 - M - le nombre de messages en transit
 - F - le nombre total de bits r à 0.

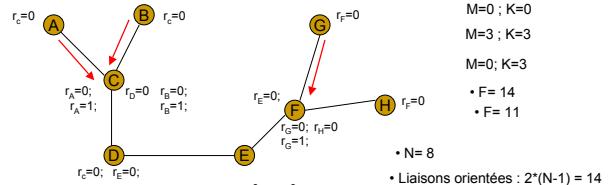
18/02/10

AR: Algorithmes par Vague

15

2. L'algorithme de l'arbre

- La topologie est un arbre, il y a donc $2(N - 1)$ liaisons (orientées).
- Le nombre de messages reçus est $(K - M)$.
- On a donc à tout instant $F = 2(N - 1) - (K - M)$



Note : Un site i peut émettre lorsqu'il n'a qu'un r_j à 0

18/02/10

AR: Algorithmes par Vague

16

2. L'algorithme de l'arbre

- A tout instant : $F = 2(N - 1) - (K - M)$:
 - $M > 0$: il y a une réception possible.
 - $M = 0$: (*preuve par contradiction*)
 - Puisqu'on suppose qu'on n'est pas dans un état où la décision n'a pas eu lieu, tous les sites ont (au moins) un bit $r \neq 0$. Donc $F >= N$.
 - Supposons qu'il n'y a pas d'émission possible, les $(N - K)$ sites n'ayant pas émis ont (au moins) un deuxième bit $r \neq 0$. Donc, $F >= N + (N - K) = 2N - K$. Mais comme $M = 0$, la formule générale de F donne $F = 2N - K - 2$. On arrive donc à une *contradiction*. Donc, il y a des émissions possibles.

Note : Un site i peut émettre lorsqu'il n'a qu'un $r_i \neq 0$

18/02/10

AR: Algorithmes par Vague

17

2. L'algorithme de l'arbre

- Montrer que dans un état terminal (plus d'émission, de réception ou de décision possible), tous les sites ont émis une fois.
 - Preuve par contradiction
 - Supposons qu'un site p_0 n'a pas encore émis.
 - Soit tous les voisins de p_0 ont émis. Comme ils n'ont pas reçu de message de p_0 , c'est donc à lui qu'ils ont envoyé. Donc, p_0 a reçu un message de tous ses voisins et il peut décider (il peut aussi émettre). Le fait qu'il peut émettre *contredit* le fait que l'état est terminal.
 - Soit au moins un de ses voisins n'a pas émis : soit p_1 ce voisin. Si p_1 ne peut pas émettre, c'est qu'au moins un autre de ses voisins (p_2) ne lui a pas envoyé de message, donc n'a pas émis. Le site p_2 doit lui aussi avoir un deuxième voisin qui n'a rien envoyé. Comme le graphe est acyclique, il ne peut pas s'agir de p_0 mais forcément d'un autre site p_3 . En itérant le raisonnement, on conclut qu'aucun site n'a émis. Or comme les feuilles peuvent émettre initialement, on aboutit à une *contradiction*.

18/02/10

AR: Algorithmes par Vague

19

2. L'algorithme de l'arbre

- En prenant la contraposée du Théorème 1: s'il n'y a ni émission ni réception possible, alors on est dans un état permettant la décision.

➤ Conséquence : comme le nombre d'émissions (donc de réceptions) est borné par le nombre de sites N (chaque processus n'envoie au plus qu'un message), on parvient toujours à une décision dans un temps finit.

18/02/10

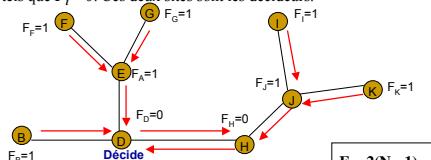
AR: Algorithmes par Vague

18

2.L'algorithme de l'arbre

- Montrer qu'il y a exactement deux décideurs.

➤ Lorsque tous les sites ont émis, et que leurs messages sont arrivés, on a ($K = N, M = 0$) : $F = N - 2$.
➤ Soit F_i le nombre de bits à 0 pour le site i . Comme i a émis, $F_i <= I$. F est la somme des F_i . On a donc $(N - 2)$ sites tels que $F_i = I$. Donc il n'y a que 2 sites tels que $F_i = 0$. Ces deux sites sont les décideurs.



Note : Un site i peut émettre lorsqu'il n'a qu'un $r_i \neq 0$

18/02/10

AR: Algorithmes par Vague

20

2. L'algorithme de l'arbre

Conclusions :

- Il y a exactement **deux décideurs**
 - Les deux décideurs sont voisins
 - Le dernier nœud dont un décideur a reçu un message est aussi un décideur.
- Dans un état terminal, tous les nœuds ont émis une fois
 - Il n'y a qu'un message qui circule dans un lien sauf celui entre les 2 décideurs où circule deux messages
- **Complexité en terme de messages :**
 - Nb_{lien} d'un arbre = $N-1 \Rightarrow$ Nb message = $(N-1)+1 = N$
- **Complexité en temps :**
 - $O(D)$ où D = diamètre de l'arbre.
- Algorithme **n'est pas décentralisé** parce que si Π_0 contient d'autres nœuds que les feuilles alors Π_0 n'est pas un ensemble d'initiateurs.

18/02/10

AR: Algorithmes par Vague

21

3. Algorithme de l'Echo

Variables :

```
set Voisp; /* ensemble de voisins de p */  
boo Recp[q] = false; ∀q ∈ Voisp /* contrôle réception message */  
int pèrep = nil;
```

p initiateur :

```
Sp : { spontanément une fois}  
    ∀q ∈ Voisp, envoie <> à q
```

```
Rp : { Un message <> arrive de q }  
    réception de <>;  
    Recp[q] = true;  
    if (pèrep = nil)  
        pèrep = q;  
        ∀r ∈ Voisp - {q}, envoie <> à r
```

```
Dp : { ∀q ∈ Voisp : Recp[q] }  
    Décision
```

<> : message vide.

p non_initiateur :

```
Rp : { Un message <> arrive de q }  
    réception de <>;  
    Recp[q] = true;  
    if (pèrep = nil)  
        pèrep = q;  
        ∀r ∈ Voisp - {q}, envoie <> à r  
  
Sp : { ∀q ∈ Voisp : Recp[q] }  
    envoie <> à pèrep
```

18/02/10

AR: Algorithmes par Vague

23

3. Algorithme de l'Echo

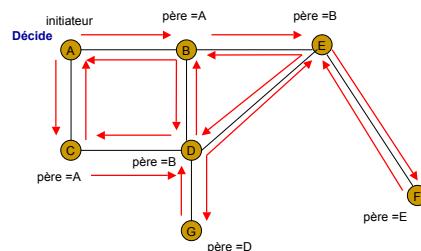
- Proposé par Chang [1982]
- Topologie arbitraire
 - Graphe connexe bidirectionnel
- Algorithme centralisé – un seul initiateur
- Principe de l'algorithme
 - **initiateur** :
 - envoie un message à tous ses voisins
 - lorsqu'il a reçu un message de tous ses voisins, il décide
 - **Un nœuds non-initiateur** :
 - sauvegarde le lien par où le premier message a été reçu ("père")
 - émet à tous les voisins sauf le père
 - lorsqu'il a reçu un message de tous ses voisins, il envoie un message à son "père"

18/02/10

AR: Algorithmes par Vague

22

3. Algorithme de l'Echo



18/02/10

AR: Algorithmes par Vague

24

3. Algorithme de l'Echo

■ Conclusions :

- L'initiateur est le décideur
- Complexité en terme de messages
 - $2 * \text{Nb}_{\text{lien}}$ ($\text{N}_{\text{blien}} = \text{nombre de liens}$)
- Complexité en terme de temps
 - $O(D)$ où $D = \text{diamètre du réseau}$.
- Possibilité de construire un arbre de recouvrement (voir TD)

18/02/10

AR: Algorithmes par Vague

25

4. Algorithme de la Phase

➢ Topologie arbitraire

- Graphe orienté fortement connexe
- Algorithme décentralisé et symétrique
- Diamètre D du réseau est connu de tous les sites
- Principe de l'algorithme :
 - Chaque processus envoie D fois un message à tous ses voisins sortants.
 - Un processus n'a le droit d'émettre un message à tous ses voisins sortants pour la $(i+1)^{\text{ème}}$ fois qu'après avoir reçu le $i^{\text{ème}}$ message de tous ses voisins entrants.
 - Un processus décide lorsqu'il a reçu D messages de tous ses voisins entrants.

18/02/10

AR: Algorithmes par Vague

26

4. Algorithme de la Phase

Variables :

```
set Inp; /* ensemble de voisins de p entrant */  
set Outp; /* ensemble de voisins de p sortant */  
int RCountp[q] = 0; ∀ q ∈ Inp /* contrôle réception message */  
int SCountp=0 /* contrôle envoi message */
```

R_p : { Un message <=> arrive de q }
réception de <=>;
RCount_p[q]++;

S_p : { ∀ q ∈ In_p : RCount_p[q] ≥ SCount_p et SCount_p < D }
forall r ∈ Out_p envoi <=> à r;
SCount_p++;

D_p : {∀ q ∈ In_p : RCount_p[q] ≥ D }
Décision

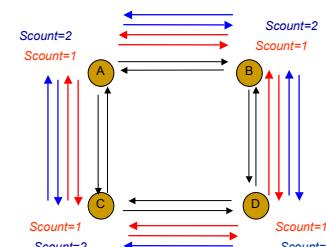
Observation: La primitive S_p doit être exécutée initialement par (au moins) un processus, l'initiateur.

18/02/10

AR: Algorithmes par Vague

27

4. Algorithme de la Phase



Diamètre = 2

18/02/10

AR: Algorithmes par Vague

28

4. Algorithme de la Phase

- Plusieurs messages peuvent être envoyés sur un lien
- Tous les processus peuvent décider
 - Plus d'émission ni de réception possible
- Complexité :
 - Messages = Nb_{lien}*D Nb_{lien} = nombre de lien
 - Temps= O(D) D= diamètre

18/02/10

AR: Algorithmes par Vague

29

Résumé Algorithmes par Vague

Algorithme	Topologie	Centr./ Décen.	Déci-deur	Symétrise	Nb. Mess.	Temps
Anneau	Anneau unidirec.	Centralisé	1 initiateur	Non	N	N
Arbre	Arbre bidirect.	Pas centr. Pas décen.	2	Oui	N	O(D)
Echo	arbitraire bidirect.	centralisé	1 initiateur	Non	2*Nb _{lien}	O(D)
Phase	arbitraire	décentralisé	Tous	Oui	Nb _{lien} *D	O(D)

18/02/10

AR: Algorithmes par Vague

30

Bibliography

- Gerard Tel, **Introduction to Distributed Algorithms**, Cambridge University Press, 1994, 2000 (2ème édition).
- Gerard Tel, **Total Algorithms**, ALCOM: *Algorithms Review, Newsletter of the ESPRIT II Basic Research Actions Program Project no. 3075*
- Ernest J.H. Chang, **Echo Algorithms: Depth Parallel Operations on General Graphs**, IEEE Transactions on Software Engineering, Vol. 8, No. 4, July 1982

18/02/10

AR: Algorithmes par Vague

31