Ordre, temps et état dans un système réparti (1)

Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

http://sardes.inrialpes.fr/~krakowia

Objectifs et contraintes

- On veut être capable de raisonner sur un système ou une application
  - Définir des prédicats, ce qui implique d'accéder à un état
  - Coordonner des activités, ce qui implique de définir un ordre
- Pourquoi est-ce différent (et plus difficile) en univers réparti ?
  - Pas de mémoire commune (support habituel de l'état)
  - Pas d'horloge commune (qui définit le séquencement des événements)
  - Asynchronisme des communications
    - Pas de borne supérieure sur le temps de transit d'un message
    - Conséquence du mode de fonctionnement de la plupart des réseaux (dont l'Internet)
  - Asynchronisme des traitements
    - Pas de bornes sur le rapport relatif des vitesses d'exécution sur deux sites
    - Conséquence de la variabilité de la charge et de l'absence (en général) de garanties sur l'allocation des ressources

Sûreté et vivacité

On est souvent amené à spécifier et à vérifier des propriétés d'un système dynamique (évoluant dans le temps)

Ces propriétés sont en général classées sous deux rubriques

Sûreté (*safety*) : un événement indésirable n'arrivera jamais

Exemples : violation de l'exclusion mutuelle
incohérence dans les données

Vivacité (*liveness*) : un événement désirable finira par arriver

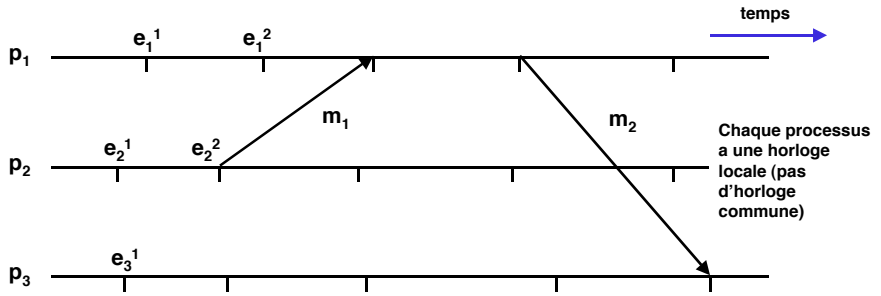
Exemples : un message sera délivré à son destinataire
une ressource demandée sera rendue disponible
un algorithme se termine (indécidable dans le cas général…)

Il est souvent impossible de fixer une borne supérieure à l'attente. Les propriétés de vivacité sont plus difficiles à établir que celles de sûreté

Le modèle asynchrone (fiable)

- C'est une tentative pour modéliser certains aspects du monde réel
  - Asynchronisme des communications et des traitements
- C'est le modèle (fiable) le plus "faible"
  - Les contraintes sont les plus fortes
  - Donc les résultats sont les plus généraux
    - Bornes sur les coûts
    - Résultats d'impossibilité
  - Le modèle peut être renforcé (en levant certaines contraintes
    - Exemple : borne supérieure sur le traitement et/ou la durée de transmission

Le modèle asynchrone



Les processus (sur différents sites) ne communiquent que par messages
Trois types d'événements
- Local (changement de l'état d'un processus)
- Lié à la communication
  - Émission d'un message
  - Réception d'un message

Pas de bornes sur :
- la durée de transmission d'un message
- le rapport des vitesses d'exécution des processus

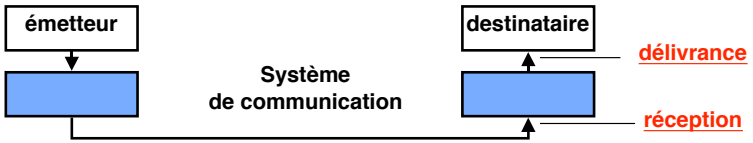
À propos des messages

On suppose disponible un système de communication permettant d'envoyer des messages entre processus

Propriétés :

1. Un message finit par arriver à destination, mais son temps de transmission n'est pas borné ; on modélise ainsi une panne (détectée) suivie d'une ou plusieurs réémission(s)
2. Un message arrive intact (non modifié) ; on suppose que des mécanismes de détection / correction d'erreur sont utilisés
3. Selon le cas, on fera ou non l'hypothèse que le canal est FIFO entre deux processus

Bien distinguer la réception d'un message de sa délivrance à son destinataire !



Événements, histoires, synchronisation

L'exécution d'un processus est une suite d'événements (local, émission, réception) appelée histoire (ou trace) du processus

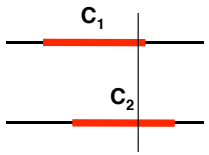
pour p_1 : $e_1^1, e_1^2, e_1^3, \ldots, e_1^k, \ldots$

Cette suite est ordonnée par l'horloge locale du processus

Que veut dire "synchroniser deux processus ?"

Imposer un ordre entre des événements appartenant à ces deux processus

Exemple : l'exclusion mutuelle



fin(C2) précède deb(C1) ou fin(C1) précède deb(C2)

Relation de précédence

Le problème :

1) Définir une relation globale de précédence (donner un sens à l'opérateur précède ci-dessus)
2) Définir un ordre entre deux événements sur la seule base d'informations locales

Solution pour 1) : utiliser le principe de causalité : la cause précède l'effet. Une relation de précédence est dite causale si elle est compatible avec ce principe. Application ici :

- Sur un processus : un événement local ne peut agir localement que sur les événements postérieurs
- Entre deux processus : l'envoi d'un message précède sa réception
- Composition : la relation de causalité est transitive

À propos de causalité

D'ou la définition [Lamport 78] : e précède causalement e' (e → e') si :

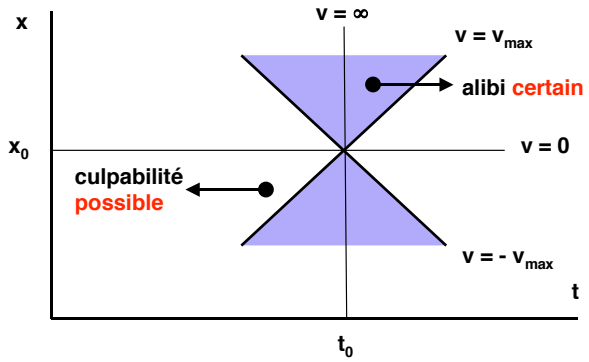
$e \rightarrow e'$ ⎧ e précède localement e' (sur 1 site, dans 1 processus), ou
⎨ ∃ message m tel que e = émission(m), e' = réception(m), ou
⎩ ∃ e" tel que (e précède e") et (e" précède e')

La relation de précédence causale → définit en fait une causalité potentielle (par négation)

Si e → e', on peut simplement dire qu'on ne viole pas le principe de causalité en disant que e est la cause de e'. On peut donc dire que e est une cause potentielle de e', mais pas que e est effectivement une cause de e' (il faudrait pour cela analyser la sémantique de l'application)

En revanche, on peut dire avec certitude que e' ne peut pas être la cause de e (le futur, jusqu'à nouvel ordre, n'agit pas sur le passé)

Illustration de la causalité



Si Z… a été vu au lieu x au temps t par un témoin digne de foi, peut-il avoir été au lieu x_0 au temps t_0 ?

N.B. t peut être antérieur ou postérieur à t_0

Dépendance et indépendance causale

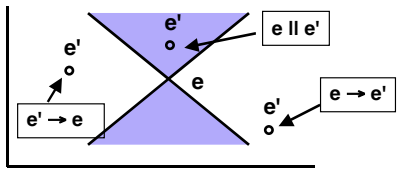
Définition : passé (ou historique) d'un événement e

hist(e) = l'ensemble des e' tels que e' → e ∪ {e}

Seul le passé strict de e peut influencer e

- si e' → e, alors e' peut influencer e
- si ¬ (e' → e), alors il est certain que e' ne peut pas influencer e

Si ¬ (e → e') et ¬ (e' → e), on note e ll e' et on dit que e et e' sont causalement indépendants (aucun des deux n'appartient au passé de l'autre, aucun des deux ne peut influencer l'autre)

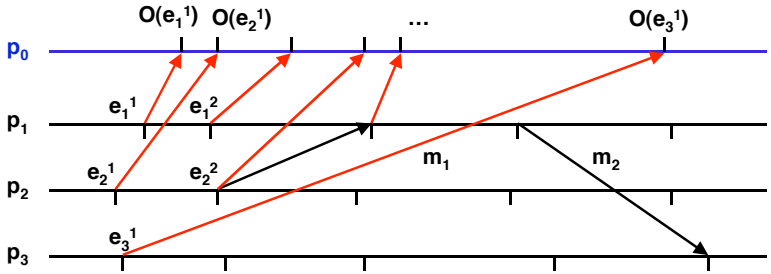


Vers un système de datation

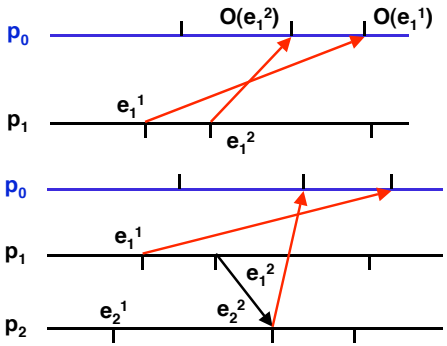
Problème : construire un système de datation des événements compatible avec la causalité

Approche : introduire un processus "observateur" p_0 qui est informé, par message, de tout événement du système

La suite des événements enregistrée par p_0 est une observation globale du système



Validité des observations



Canal non FIFO. L'observation viole la causalité : les événements sont observés dans l'ordre inverse de leur occurrence

La causalité est encore violée : $e_1^1 \rightarrow e_2^2$, mais ces événements sont observés dans l'ordre inverse. Ce phénomène est indépendant de la propriété FIFO

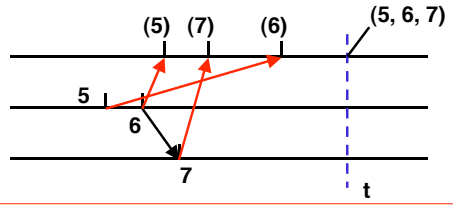
Une observation du système est un ordonnancement particulier des événements du système.

Une observation est dite valide si pour tout couple d'événements (e, e') tels que $e \rightarrow e'$, O(e) précède O(e'). Sinon, elle est dite invalide (elle viole la causalité)

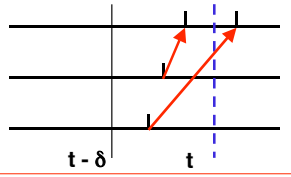
Observation valide : une première solution

On va temporairement lever l'hypothèse d'asynchronisme (temps de transmission borné par δ) et supposer qu'on dispose d'une horloge HR donnant le temps réel
Chaque événement transmis à l'observateur est estampillé par HR ; donc une observation de e est le couple (e, HR(e))

Au temps t, on délivre tous les messages ayant des estampilles < t – δ dans l'ordre des estampilles.
NB : délivrance d'un message ≠ réception !



Nécessité du t – δ : risque de réception après t d'événements antérieurs à ceux déjà reçus avant t



Caractérisation des observations valides

- **Condition de validité**
  - Dans un système où les observations sont ordonnées par des estampilles H, une condition suffisante de validité est :

$$e \rightarrow e' \Rightarrow H(e) < H(e')$$ condition de validité faible de l'horloge (car implication dans un seul sens)

Cette condition est trivialement satisfaite par HR

  - En fait, on ne dispose pas de HR et le temps de propagation des messages n'est pas borné
  - Néanmoins on va construire un système d'horloges assurant une observation valide, en respectant la condition de validité faible
  - L'absence de borne δ aura une conséquence sur la complétude, non sur la validité

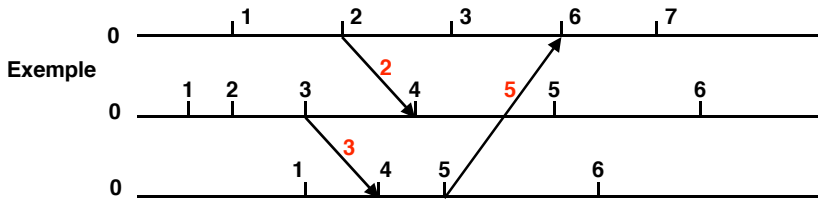
Horloges logiques (Lamport) - 1

On associe à chaque site i un compteur HL_i à valeurs entières
Un événement e se produisant sur le site i est daté par la valeur courante de HL_i , soit $HL_i(e)$
Un message m émis à partir du site i porte une estampille égale à sa date d'émission

Règles d'évolution des horloges (sur un site i)

Initialisation :	$HL_i = 0$
Événement local :	$HL_i = HL_i + 1$
Envoi d'un message m :	on envoie (m, E_m), avec $E_m = HL_i$ (après incrémentation)
Réception d'un message (m, E_m) :	$HL_i = \max(HL_i, E_m) + 1$

Horloges logiques - 2



1) **HL satisfait la condition de validité faible : e → e' ⇒ HL(e) < HL(e')**
Mais HL(e) < HL(e' ⇒ ¬ (e' → e). Donc ou bien e' → e, ou bien e' Il e

2) **L'ordre n'est pas strict. Deux événements peuvent avoir la même estampille : ils sont alors causalement indépendants. Si on veut un ordre strict, il faut ajouter un critère (en général on prend le numéro du processus)**

Si e ∈ p_i, e' ∈ p_j, alors

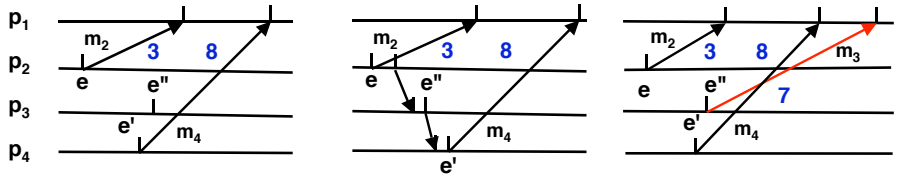
(HL(e), i) < (HL(e'), j) ssi HL(e) < HL(e') ou ((HL(e) = HL(e') et i<j)

Horloges logiques - 3

Conséquence de l'asynchronisme : non-détection des événements manquants

p_1 reçoit des messages des autres processus. On souhaite qu'ils lui soient délivrés dans l'ordre de leurs estampilles.

On a délivré m_2 (3) et on reçoit m_4 (8). Peut-on le délivrer ?



Les trois situations sont indistinguables avec les seules HL

Plus généralement, si HL(e) < HL(e'), existe-t-il e" tel que e → e" → e' ? Question insoluble. Il s'agit d'une propriété de **vivacité** (un événement va-t-il arriver, et quand ?)

Comment vivre avec l'asynchronisme

Dans la pratique, comment résoudre le problème de la délivrance des messages dans l'ordre (et les problèmes similaires) ?

On dit qu'un message reçu est stable si aucun message portant une estampille inférieure ne parviendra au destinataire
Règle : on ne délivre un message que s'il est stable

Pour s'en assurer, on attend d'avoir reçu un message de tous les émetteurs potentiels. On peut alors les délivrer dans l'ordre des estampilles. Ce principe est appliqué dans la suite. Il faut des canaux FIFO (si non FIFO, c'est possible mais plus complexe)

Problèmes
1. **Il faut connaître tous les émetteurs potentiels (dépend de la situation)**
2. **Que faire si on n'a rien reçu d'un processus ? Lui envoyer un message avec demande de réponse (*ping*)**
3. **On ne garantit toujours pas la terminaison en temps borné. Dans la pratique, on utilise un délai de garde, avec le risque d'une arrivée tardive.**

Applications des horloges logiques

- **Mécanismes utilisant une file d'attente répartie**
  - **Exclusion mutuelle**
  - **Mise à jour de copies multiples**
  - **Diffusion cohérente**
- **Détermination de l'accès "le plus récent"**
  - **Invalidation de cache**
  - **Mémoire virtuelle répartie**
- **Génération de noms uniques**
  - **typiquement : <adresse de site, estampille locale>**
  - **utilisation de l'heure physique pour unicité sur le long terme**
- **Aide à la synchronisation physique**

Plusieurs de ces exemples sont développés dans la suite

Exemple 1 : exclusion mutuelle répartie

- **Rappel du problème**
  - **On définit des sections critiques dans plusieurs processus. On veut garantir qu'un seul processus au plus est dans sa section critique à un instant donné**
  - **En réparti, que veut dire "à un instant donné" ? On préfère une spécification utilisant l'ordre (cf plus haut) ; pour deux processus quelconques : fin(C_i) → deb(C_j) ou fin(C_j) → deb(C_i)**
  - **Conditions supplémentaires**
    - **équité entre les processus**
    - **vivacité (ni interblocage, ni privation)**
    - **solution symétrique (pas d'ordonnanceur centralisé)**

Exclusion mutuelle répartie (1)

Le système comporte n sites (n connu) reliés par des canaux FIFO

Principe de la solution (Ricart et Agrawala, améliorant un algorithme de Lamport)
- utiliser une file d'attente répartie, ordonnée par les estampilles
- garantir la stabilité des messages

Intérêt historique : un des premiers algorithmes répartis utilisant les horloges logiques

- demande d'entrée en section critique par p_i (on note ⇒ l'envoi de message)

  p_i ⇒ tous (y compris p_i) : (REQ, i, HL_i)
- réception par p_j de (REQ, i, HL_i)

  p_j non demandeur : p_j ⇒ p_i : (AUT, j, HL_j)

  p_j demandeur : (ou s.c. en cours) **if** (j, HL_j) < (i, HL_i) {put (i, Q_j)} **else** {p_j ⇒ p_i : (AUT, j, HL_j)}

La file Q_i sur le site i collecte toutes les requêtes postérieures (au sens des estampilles) à celles de p_i

La file globale virtuelle est représentée par l'ensemble des Q_i

Exclusion mutuelle répartie (2)

p_i entre en section critique ssi il a reçu un message AUT de tous les autres

quand p_i sort de section critique, il exécute :

for all $p_j \in Q_i$: {p_i ⇒ p_j : (AUT, i, HL_i) ; out (p_j, Q_j)}

Les processus exécutent leur section critique dans l'ordre des estampilles de leur requête d'entrée (HL avec discrimination par numéro de processus)

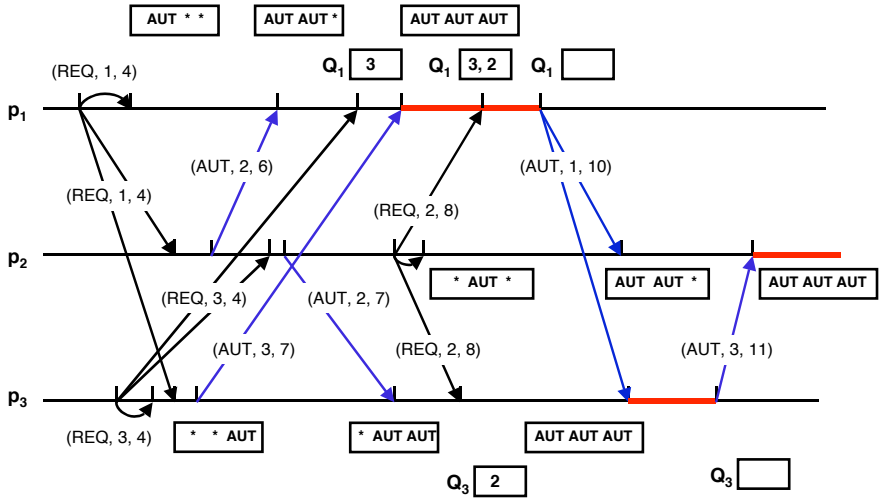
Nombre de messages échangés : (n—1) REQ, (n—1) AUT : 2 (n —1) messages

Quelques améliorations ponctuelles possibles

Exemple (Carvalho - Roucairol) : l'accord donné par un site est considéré comme valable pour les demandes suivantes, sauf avis contraire (i.e. le site dépose lui-même une demande). L'accord n'est demandé qu'aux sites qui ne l'ont pas donné depuis l'exécution d'une section critique

Un algorithme plus efficace (O(log n) messages) est présenté plus loin

Exclusion mutuelle répartie (3)

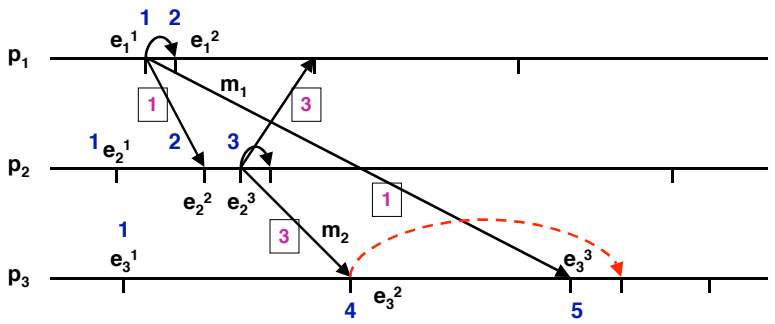


Exemple 2 : diffusion causale (1)

- **Diffusion**
  - diffusion d'un message *m* = envoi de *m* à un ensemble de processus destinataires
    - diffusion générale (*broadcast*)
    - diffusion sélective (*multicast*)
  - mécanisme très important en programmation répartie
    - pour partager un état
    - pour maintenir la cohérence de données réparties
- **Problèmes de la diffusion**
  - Spécifier précisément un mécanisme de diffusion est une tâche délicate
    - Spécifier les invariants à maintenir (différentes formes de diffusion)
  - Garantir les propriétés spécifiées en cas de défaillances est un problème très difficile (cf suite du cours)
- **Une forme particulière : la diffusion causale**
  - On suppose dans un premier temps la communication fiable
  - La diffusion causale doit garantir la propriété suivante

$$\forall m, m', \forall i : send(m) \rightarrow send(m') \Rightarrow deliver_i(m) \rightarrow deliver_i(m')$$

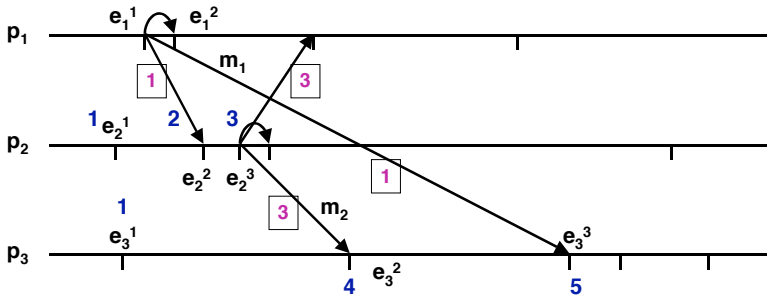
Exemple 2 : diffusion causale (2)



L'exécution ci-dessus viole la causalité : l'émission de m_2 dépend causalement de la réception de m_1, mais m_2 est reçu avant m_1 sur p_3
Le message m_2 reçu sur p_3 n'est pas stable

Les horloges logiques ne permettent pas de détecter l'incohérence.

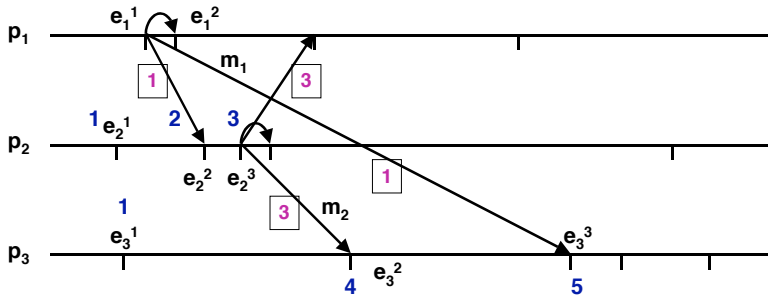
Exemple 3 : mise à jour de copies multiples (1)



Une base de données est maintenue en 3 exemplaires (un par site). Chaque site peut consulter la base ou la mettre à jour (principe *read one, write all*).

Si m_1 et m_2 sont des mises à jour, l'exécution ci-dessus conduit à une incohérence si ces mises à jour ne commutent pas (ordre m_1 ; m_2 sur p_2, ordre m_2 ; m_1 sur p_3)

Exemple 3 : mise à jour de copies multiples (2)



Principe de la mise à jour cohérente : ne pas appliquer une mise à jour tant que l'on n'est pas sûr qu'elle est stable (même principe que l'exclusion mutuelle)
Pour cela, attendre d'avoir reçu un message de tous les sites ; s'il en manque, solliciter les sites correspondants. Ensuite, appliquer les mises à jour dans l'ordre des estampilles
Donc on n'appliquera pas m_2 sur p_3 tant qu'on n'aura pas reçu un message de p_1 (ici m_1). On utilise l'hypothèse FIFO
Pour la consultation, on lit la valeur locale si on peut accepter une valeur périmée. Sinon, on assure d'abord (comme plus haut) la mise à jour de la copie

État d'un système réparti

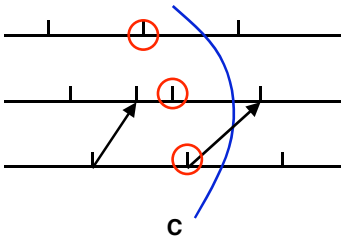
La notion même d'état n'est pas évidente a priori pour un système réparti.

Elle suppose un observateur universel ayant un accès instantané à toutes les parties du système. Un tel observateur n'est pas physiquement réalisable

Problème : comment donner un contenu concret et pratiquement utilisable à la notion d'état ?

Accessoirement : de quoi a-t-on réellement besoin?

- **Notion de coupure (*cut*)**
  - Notion intuitive (photo instantanée)
  - Capture l'état résultant du "dernier" événement "avant" la coupure pour chaque processus

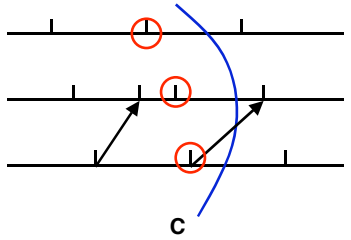


Coupures

Définition plus précise : une coupure est un ensemble d'événements (contenant les événements "à gauche" de la coupure)

e ∈ C et (e' précède localement e) ⇒ e' ∈ C

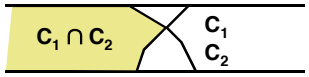
C ⊆ E (ensemble de tous les événements

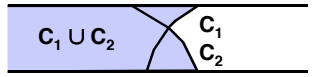


On peut définir un ordre sur les coupures par inclusion : $C_1 < C_2$ si $C_1 \subset C_2$

On note que $C_1 < C_1 \cup C_2$, $C_2 < C_1 \cup C_2$
$C_1 > C_1 \cap C_2$, $C_2 > C_1 \cap C_2$

Tout couple(C_1 C_2) a un élément inférieur $C_1 \cap C_2$ et un élément supérieur $C_1 \cup C_2$.
Cela définit une structure de treillis sur les coupures





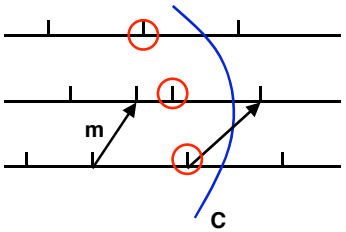
Coupures cohérentes

Une coupure est dite cohérente si :

(e ∈ C et e' → e) ⇒ e' ∈ C

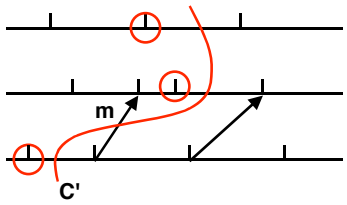
Une coupure cohérente est fermée par la relation de précédence causale (exemple : C)

En particulier tout message reçu dans la coupure a été envoyé dans la coupure (mais l'inverse n'est pas nécessairement vrai)



La coupure C' n'est pas cohérente : elle contient l'événement "réception de m" mais non l'événement "émission de m". Le message m "vient du futur" !

Une coupure incohérente donne une vue du système qui viole la causalité



Propriétés des coupures cohérentes

Le sous-ensemble des coupures cohérentes d'un système forme aussi un treillis (sous-treillis de celui de toutes les coupures)

Si C_1 et C_2 sont des coupures cohérentes, alors $C_1 \cup C_2$ et $C_1 \cap C_2$ le sont aussi. En effet :

si e ∈ $C_1 \cap C_2$ et e' → e, alors :

e' ∈ C_1, car C_1 cohérente et e' ∈ C_2, car C_2 cohérente

donc e' ∈ $C_1 \cap C_2$

si e ∈ $C_1 \cup C_2$ et e' → e, alors :

ou bien e ∈ C_1 et alors e' ∈ C_1, car C_1 cohérente

ou bien e ∈ C_2 et alors e' ∈ C_2, car C_2 cohérente

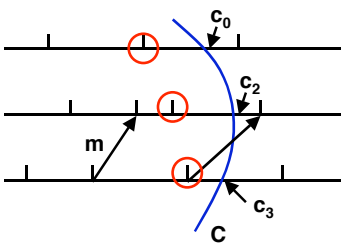
donc e' ∈ $C_1 \cup C_2$

Caractérisation des coupures cohérentes

Considérons les c_i, événements "fictifs" (car non liés à un changement d'état) qui définissent la coupure.

On note $C = (c_1\ c_2\ \dots\ c_n)$

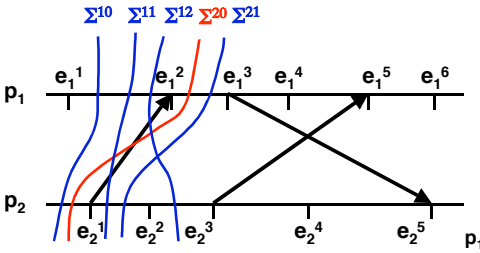
Alors **$(\forall\ i, j,\ \ c_i \text{ ll } c_j) \Leftrightarrow$ C est cohérente**

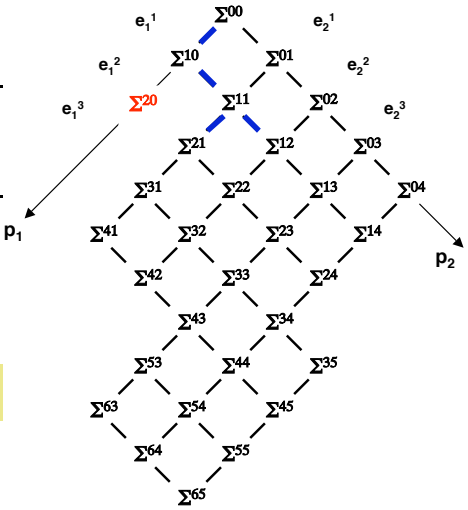


1) Soit C une coupure cohérente. Supposons $\exists\ i \neq j,\ c_i \rightarrow c_j$. On a : $c_i \rightarrow c_j \Rightarrow c_i \rightarrow a_i \rightarrow a_j \rightarrow c_j$ (en effet si $i \neq j$, il faut un message entre les deux). Mais alors $a_i \rightarrow c_j \Rightarrow a_i \in C$, contradictoire avec $c_i \rightarrow a_i$

2) Soit $C = (c_1\ c_2\ \dots\ c_n)$ avec $\forall\ i, j,\ \ c_i \text{ ll } c_j$. Alors C est cohérente. En effet : soit $a_i \rightarrow c_i$ et $a_j \rightarrow a_i$. Alors $a_j \rightarrow c_j$, i.e. $a_j \in C$. En effet, si $c_j \rightarrow a_j$ ($c_j \parallel a_j$ impossible), on aurait $c_j \rightarrow a_j \rightarrow a_i \rightarrow c_i$, contradictoire avec c_i ll c_j.

Le treillis des coupures cohérentes (1)





Avancer d'un pas : exécuter un événement permis (conduisant à un état cohérent)

Une observation valide = un chemin dans le treillis des coupures cohérentes

Une même exécution réelle peut conduire à plusieurs observations valides différentes

Le treillis des coupures cohérentes (2)

Accessibilité

S et S' étant deux états cohérents, on dit que S' est **accessible** depuis S (on note S ~> S') s'il existe un chemin menant de S à S' dans le treillis des coupures cohérentes.

Exemples :

$\Sigma^{21} \rightsquigarrow \Sigma^{55}$

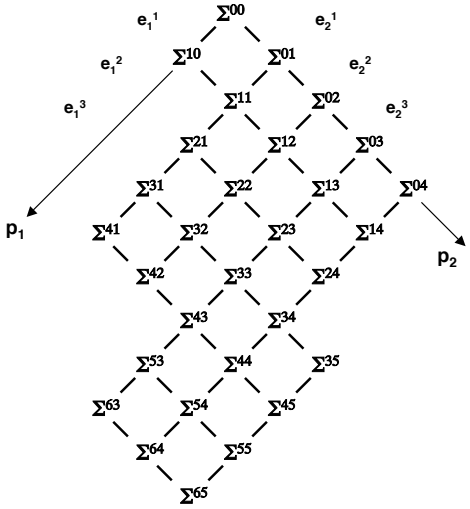
$\neg(\Sigma^{31} \rightsquigarrow \Sigma^{24})$

Pour que $\Sigma^{ij} \rightsquigarrow \Sigma^{kl}$, il faut que $i \leq k$ et $j \leq l$

La relation d'accessibilité est transitive

Si on raisonne en termes de coupures :

$S \rightsquigarrow S' \Leftrightarrow S \subset S'$



Autre exemple