Le Temps

Université François Rabelais de Tours
Faculté des Sciences et Techniques
Antenne Universitaire de Blois

Master 1 Informatique
Option Systèmes d'Information et Aide à la Décision

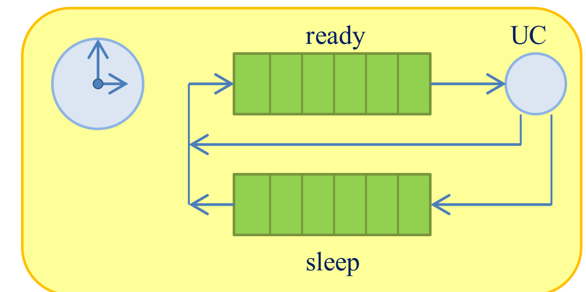
Mohamed Taghelit
taghelit@univ-tours.fr

Différents Aspects du Temps

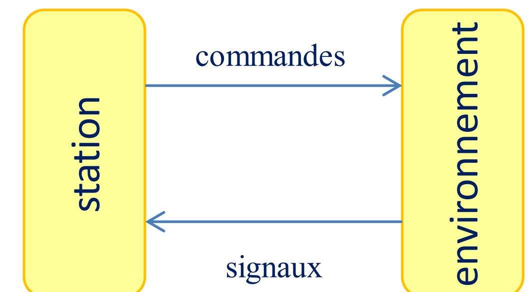
Le temps : concept fondamental des applications et des systèmes informatiques :

- Contexte réparti : autant de référentiels possibles que de sites
- Revêt des aspects différents : chacun nécessitant un traitement particulier

- Le temps interne
  - s'appuie sur l'horloge locale d'un site
    - interruption horloge, qui se déclenche avec une périodicité définie à partir de cette horloge
    - délai de suspension d'un processus,

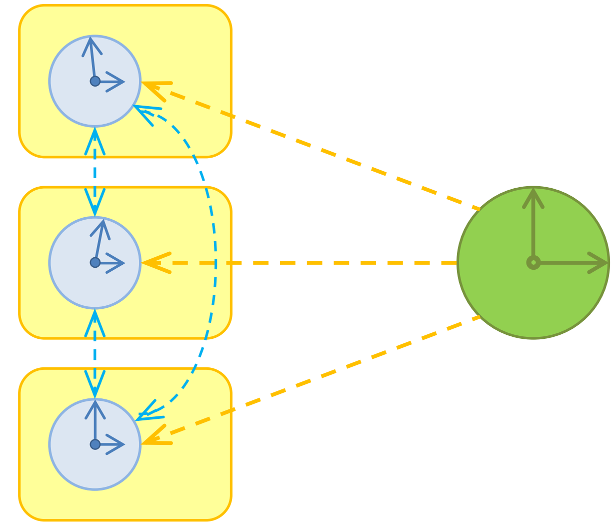


- Le temps de l'environnement (temps réel)
  - désigne le rythme d'évolution de l'environnement
    - prise en compte des signaux provenant de l'environnement
    - émission de commandes permettant d'agir sur l'environnement
  - fonctionnement en une suite de cycles "calcul-commandes-signaux"

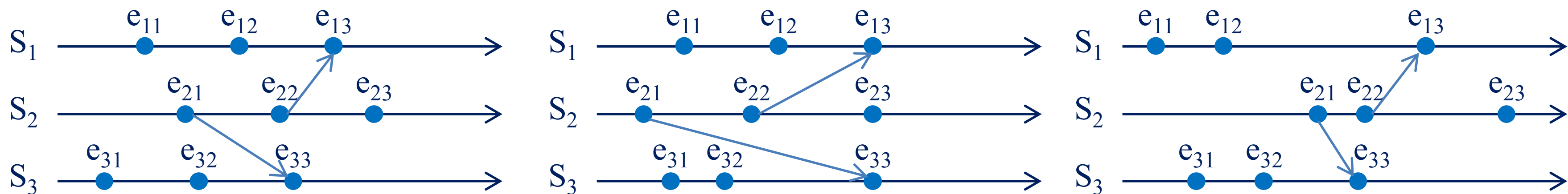


Différents Aspects du Temps

- Le temps universel
  - dater les évènements (comparaison, tri, …)
  - délivré par des organismes officiels (LPTF)
  - synchronisation des horloges physiques



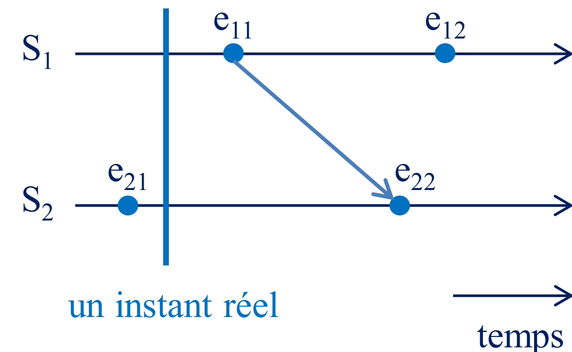
- Le temps logique
  - l'ordre causal est significatif (non la précédence temporelle)
  - plusieurs mécanismes basés sur cet ordre : horloges logiques



Analyse d'une Exécution

- Interprétation correcte d'une exécution répartie
  - difficulté due à l'indéterminisme inhérent à son exécution (les exécutions locales peuvent ou non s'influencer mutuellement)
  - nécessité de connaître les relations de dépendance causale entre les différents événements

- Représentation d'une exécution
  - pour chaque processus, l'écoulement du temps est représenté par une droite orientée
  - les événements sont représentés par des points sur ces droites, dans leur ordre relatif d'occurrence
  - les messages sont décrits par des flèches connectant l'événement "émission" à l'événement "réception correspondante"



Un Exemple d'Application

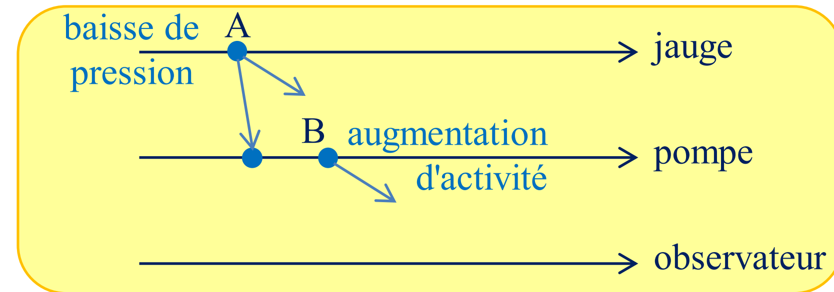
Exemple de la fuite (Friedemann Mattern)

Considérons un système très simple composé :

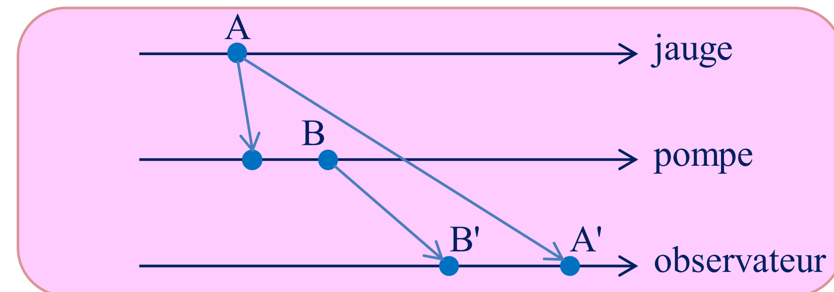
- d'un tuyau,
- d'une jauge de pression et
- d'une pompe chargée de réguler la pression dans le tuyau en fonction de l'information envoyée par la jauge

Les processus gérant la pompe et la jauge peuvent communiquer entre eux et envoyer des messages à un observateur chargé de la surveillance.

Supposons l'exécution suivante : une fuite survient, la pression baisse, la jauge la détecte et envoie un message à la pompe et à l'observateur (événement A). A la réception du message, la pompe augmente son activité pour maintenir le niveau de pression puis envoie un message à l'observateur (événement B).



Supposons que l'observateur reçoive le message de la pompe (événement B') avant celui de la jauge (événement A').



Interprétation de l'observateur : la pompe a augmenté (de manière injustifiée) son activité, ce qui a conduit à une augmentation de pression, laquelle a entrainé une fuite ce qui a provoqué la baisse de pression détectée par la jauge.

Cette interprétation erronée est due au fait que l'observateur ne connaît pas la relation de causalité entre les événements.

Causalité et Concurrence

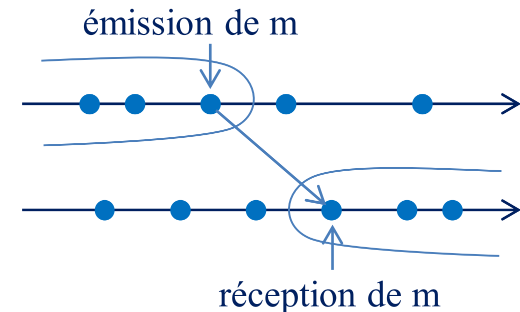
Événements pouvant survenir sur un site

- événement interne — met en jeu un site et modifie son état en conséquence
- envoi de message
- réception de message — mettent en jeu un site et un canal ; l'état du site et celui du canal sont modifiés en conséquence

Contraintes séquentielles fondamentales

C1 les événements qui apparaissent sur un site donné sont totalement ordonnés

C2 si l'on considère un message quelconque **m**, l'événement "émission du message **m**" précède l'événement "réception de **m**"



Relation de causalité (potentielle)

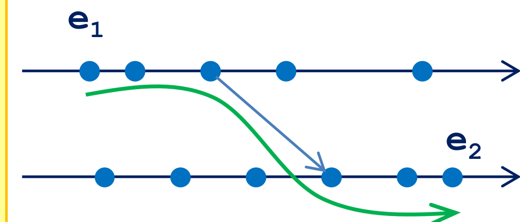
Définition : Soit **E** un ensemble d'événements, e_1 et e_2 deux événements de **E**. La relation de causalité sur **E**, notée "→" est telle que, pour une exécution donnée, $e_1 \rightarrow e_2$ si l'une des 3 conditions suivantes est vérifiée:

(1) e_1 et e_2 sont deux événements du même processus, et e_1 précède e_2,

(2) e_1 est l'événement "envoi d'un message" et e_2 est l'événement "réception de message" correspondant,

(3) il existe un événement e_3 tel que $e_1 \rightarrow e_3$ et $e_3 \rightarrow e_2$ (transitivité)

elle est irréflexible et transitive, donc asymétrique. Elle définit un ordre partiel strict.



traduction graphique de la dépendance causale

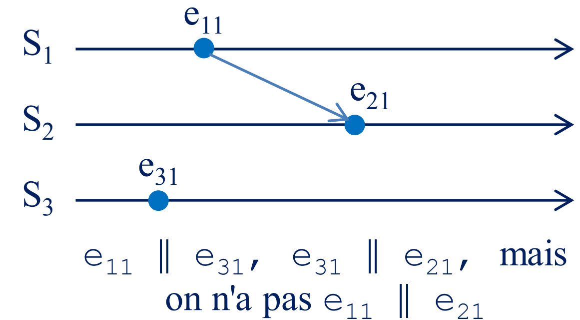
Causalité et Concurrence

La concurrence

Définition : Soit $\mathbf{E}$ un ensemble d'événements, $\mathbf{e_1}$ et $\mathbf{e_2}$ deux événements de $\mathbf{E}$. Si on n'a ni $\mathbf{e_1} \rightarrow \mathbf{e_2}$, ni $\mathbf{e_2} \rightarrow \mathbf{e_1}$, alors aucun des deux ne peut avoir d'influence causale sur l'autre. On définit la relation suivante :

$$e_1 \parallel e_2 \Leftrightarrow \text{NON}(\ e_1 \rightarrow e_2\ )\ \text{et}\ \text{NON}(\ e_2 \rightarrow e_1\ )$$

$\mathbf{e_1}$ et $\mathbf{e_2}$ sont dits concurrents. Cette relation n'est pas transitive.



Utilité de la causalité

La connaissance de la relation de précédence causale est utilisée pour résoudre un grand nombre de problèmes en algorithmique répartie :

la conception d'algorithmes : assurer la vivacité et l'équité d'algorithmes d'exclusion mutuelle, construire des algorithmes de détection d'interblocage,

le debugging distribué : aide à construire des états consistants, des points de reprise en cas de panne,

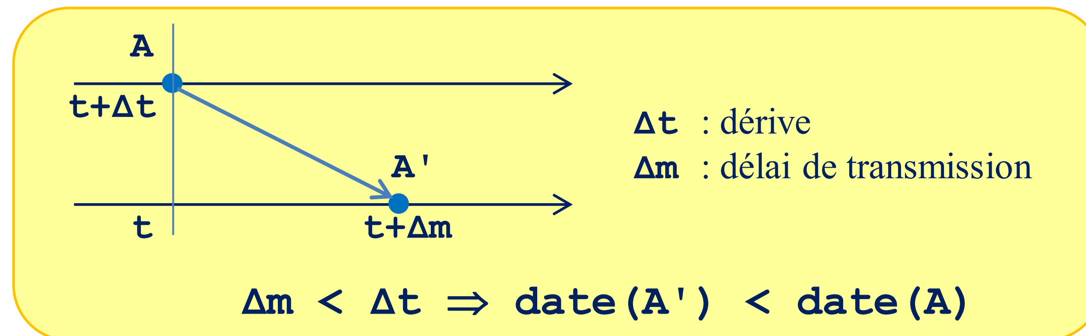
la connaissance de l'état d'avancement : permet à un processus de mesurer l'état d'avancement des autres processus (effacer des informations obsolètes, détecter la terminaison, …)

la mesure de la concurrence : les événements qui ne sont pas reliés causalement peuvent être exécutés de manière concurrente (degré de parallélisme).

Les Horloges Physiques

Modèle considéré :

- chaque site possède sa propre horloge physique, toutes les horloges sont synchronisées au départ,
  - précision suffisante → possibilité de dater tous les événements (semblable à la vie courante)
  - dans une application répartie, il se produit beaucoup plus d'événements en un temps beaucoup plus faible → il est donc fondamental d'avoir des horloges synchronisées de manière très précise pour préserver une relation de causalité correcte.
- les horloges dérivent
  - la dérive dépend fortement de la fréquence de l'horloge, et les fréquences diffèrent d'un site à l'autre
  - la dérive dépend des conditions climatiques : elle est plus importante lorsqu'il fait chaud et humide



- solutions :
  - utilisation d'une horloge unique sur un site maître. Plus de dérive mais un coût de gestion très important (interrogation du maître pour chaque datation). Incertitude sur la date reçue.
  - utilisation d'un algorithme de resynchronisation d'horloge (définition d'une horloge distribuée idéale, définition d'un schéma d'échange de messages, définition des points de resynchronisation, définition de l'algorithme utilisé pour garder l'horloge locale à l'intérieur d'un intervalle spécifié de l'horloge idéale).

Les Horloges Logiques

Horloges logiques (Leslie Lamport – 1978)

La définition des horloges logiques repose sur une notion de temps virtuel discret :

– vu comme une succession d'événements,

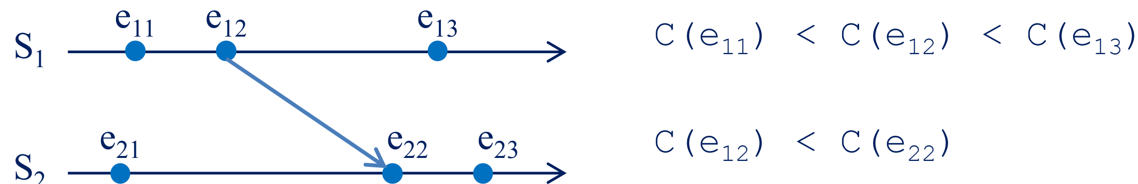
– reste figé tant qu'aucun événement ne se produit

Définition : une horloge logique est une fonction **C : E → T**, où **T** est un ensemble partiellement ordonné, qui vérifie la condition suivante :

$e \rightarrow e' \Rightarrow C(e) < C(e')$ // Condition de consistance des horloges

Propriétés (combinaison de la condition de consistance des horloges et de la relation de causalité) :

- Si un événement **e** a lieu avant un autre événement **e'** dans un même processus, alors la date logique associée à **e** précède la date logique associée à **e'**.
- Pour tout message **m** envoyé d'un processus à un autre, la date logique de l'événement "émission de **m**" précède toujours la date logique de l'événement "réception de **m**".



Les horloges logiques de Lamport utilisent les entiers ℕ comme domaine de temps **T**.

Les Horloges Logiques

Gestion des horloges logiques :

- chaque processus p_i gère un compteur C_i, initialisé à **0**.
- les horloges logiques suivent le protocole suivant (pour garantir la condition de consistance) :

  (1) lorsque le processus p_i exécute un événement interne ou envoie un message, l'horloge C_i est incrémentée :

  $$C_i = C_i + d \qquad (d > 0)$$

  (2) tout message contient une estampille qui est égale à la date (heure) d'émission du message,

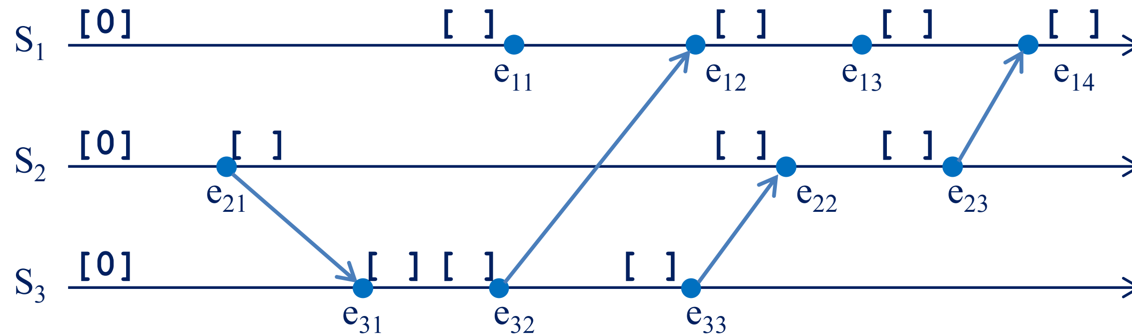
  (3) lorsqu'un processus p_i exécute l'événement "réception d'un message contenant l'estampille **t**", il incrémente son horloge :

  $$C_i = \max(t, C_i) + d \qquad (d > 0)$$

- pour chaque opération, on suppose que l'horloge est mise à jour avant que l'événement ne soit daté.
- on utilise souvent la valeur $d = 1$, mais d peut ne pas être constant (par exemple, valeur approchée du temps physique écoulé).

Les Horloges Logiques

Exemple

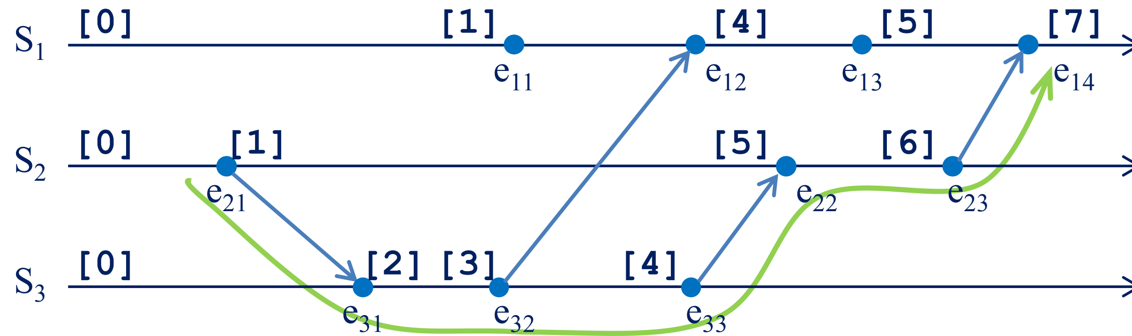


Remarques

- aucune relation entre le temps physique et le temps logique : e_{31} qui précède physiquement e_{11} a une estampille plus élevée,
- les horloges sont croissantes le long d'un chemin causal,
- les horloges de Lamport ne définisse qu'un ordre partiel entre les événements : deux événements sur deux sites différents peuvent avoir la même horloge (pas de lien causal entre eux),

Les Horloges Logiques

Exemple



Remarques

- aucune relation entre le temps physique et le temps logique : $\mathbf{e_{31}}$ qui précède physiquement $\mathbf{e_{11}}$ a une estampille plus élevée,
- les horloges sont croissantes le long d'un chemin causal,
- les horloges de Lamport ne définisse qu'un ordre partiel entre les événements : deux événements sur deux sites différents peuvent avoir la même horloge (pas de lien causal entre eux),

Les Horloges Logiques

La relation établie par les horloges logiques peut être étendue à un ordre total :

- numéroter les sites.
- associer à un événement **e** sur le site **i** une estampille formée du couple **(C(e),i)**

Définition : soient **e** et **e'** deux événements d'estampilles respectives **(h,i)** et **(k,j)**. La relation d'ordre total est définie par :

$$(h,i) < (k,j) \Leftrightarrow (h < k) \textbf{ ou } (h=k \textbf{ et } i<j)$$

- ordre total généralement utilisé pour assurer les propriétés de vivacité d'une application répartie : les requêtes sont estampillées et servies suivant l'ordre total déterminé par leurs estampilles.

- Lamport l'utilise pour construire un algorithme réparti d'exclusion mutuelle.

- les horloges logiques donnent une information incomplète sur les liens de causalité entre les événements :

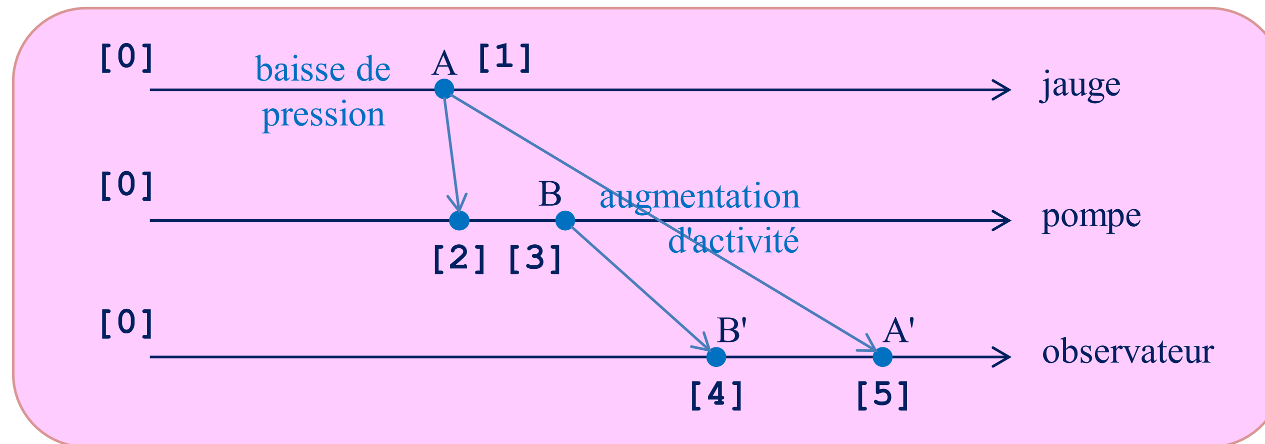
  Soient deux événements **e** et **e'** concurrents, ils peuvent avoir des horloges identiques ou des horloges différentes. Donc, lorsque deux événements ont des horloges différentes, on ne peut pas savoir s'ils ont ou non un lien causal :

$$e \rightarrow e' \Rightarrow C(e) < C(e')$$

<u>mais</u> $C(e) < C(e') \Rightarrow (e \rightarrow e' \textbf{ ou } e \parallel e')$

Les Horloges Logiques

Analyse de l'exemple de la fuite :



Conclusions :

- l'interprétation précédente de l'observateur est erronée : on peut affirmer que l'augmentation d'activité de la pompe n'est pas à l'origine de la baisse de pression car

  C(B) > C(A)

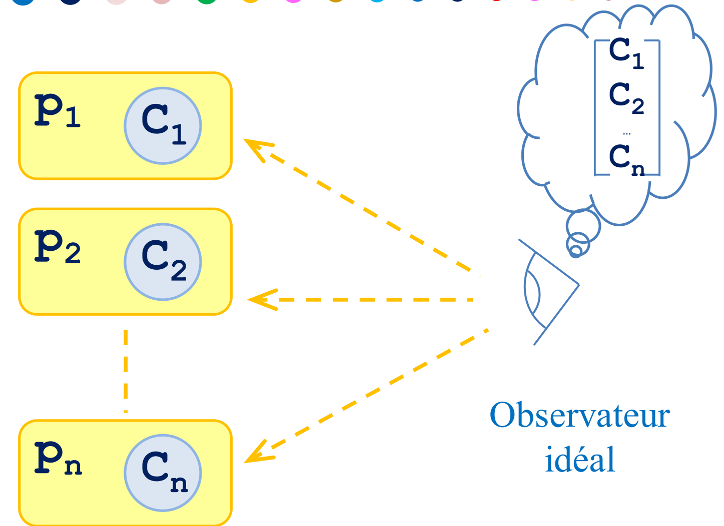
- mais on n'est pas sûr pour autant que la baisse de pression soit à l'origine de l'augmentation d'activité : rien ne prouve qu'il y ait eu échange de messages entre la jauge et la pompe.
- les horloges logiques apporte à un site une connaissance globale du système

Les Horloges Vectorielles

Horloges vectorielles (Friedemann Mattern– 1988)

La définition des horloges vectorielles repose aussi sur une notion de temps virtuel discret :

- obtention de la meilleur approximation du "temps global"
- le domaine de temps **T** est maintenant un ensemble de vecteurs de dimension **n** (où **n** est le nombre de processus)



Chaque processus $\mathbf{p_i}$ gère un vecteur $\mathbf{vt_i}$

- $\mathbf{vt_i[i]}$ est l'horloge logique locale de $\mathbf{p_i}$ et donne l'heure locale de $\mathbf{p_i}$
- $\mathbf{vt_i[j]}$ est l'information la plus récente que $\mathbf{p_i}$ a sur l'heure locale de $\mathbf{p_j}$.
  Si $\mathbf{vt_i[j]=x}$, cela signifie que $\mathbf{p_i}$ sait que l'heure de $\mathbf{p_j}$ a progressée au moins jusqu'à $\mathbf{x}$.

Le vecteur $\mathbf{vt_i}$ constitue donc la vue que $\mathbf{p_i}$ a du temps logique global et il est utilisé pour estampiller les événement en $\mathbf{p_i}$. Ce vecteur est initialisé à zéro.

Les Horloges Vectorielles

Gestion des horloges vectorielles :

- La mise à jour de l'horloge vectorielle vt_i se fait de la manière suivante :

  (1) à chaque événement, le processus p_i incrémente son horloge logique locale vt_i :

  $$vt_i[i] = vt_i[i] + d \qquad (d > 0)$$

  (2) tout message contient une estampille qui est le vecteur local au moment de l'émission du message,

  (3) lorsqu'un processus p_i exécute l'événement "réception d'un message contenant l'estampille vt", il exécute les opérations suivantes :

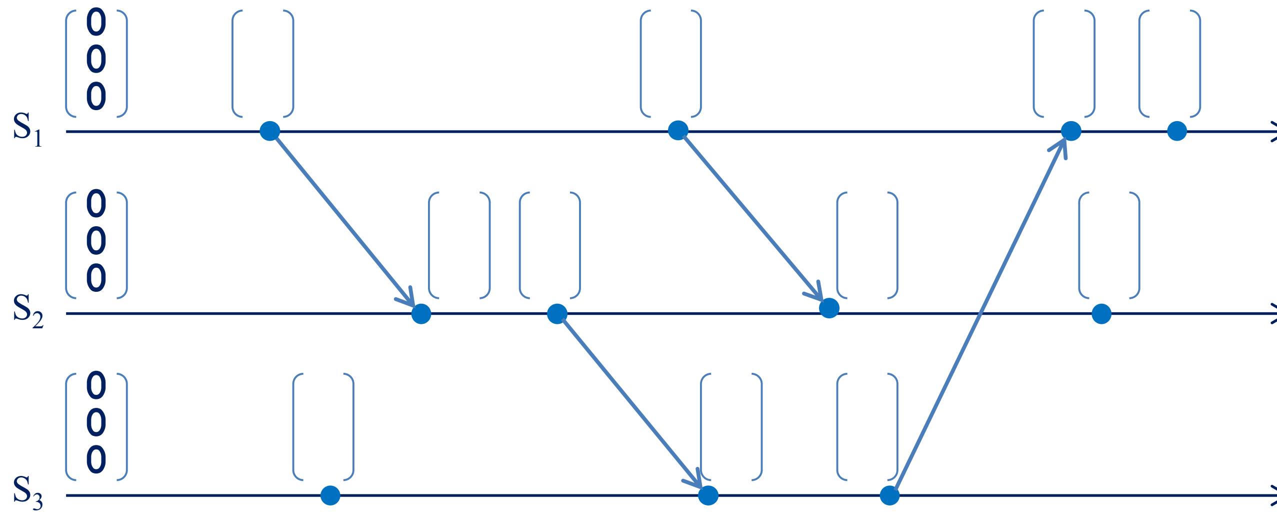
  $$\forall\ k,\ 1 \leq k \leq n,\ vt_i[k] = \max(vt[k],\ vt_i[k])$$

- comme pour les horloges de Lamport, on suppose qu'à chaque opération, l'horloge est mise à jour avant que l'événement ne soit estampillé.

  Note : dans le cas d'une réception de message, il y a successivement incrémentation de l'horloge locale et mise à jour par comparaison.

Les Horloges Vectorielles

Exemple



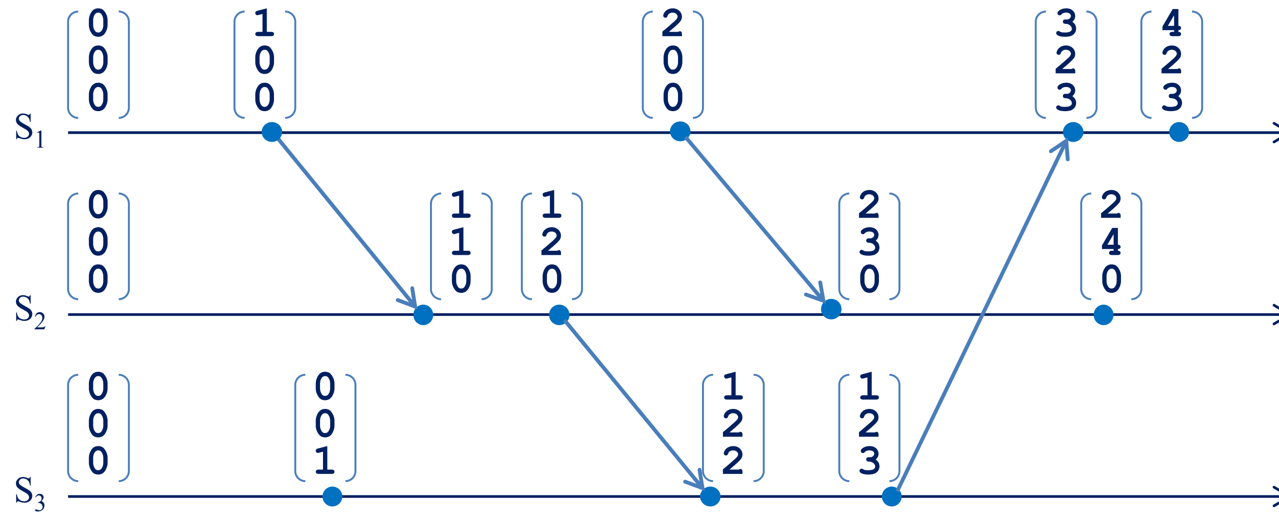
Remarque

- comme seul le processus p_i peut incrémenter la $i^{ème}$ composante du temps global, c'est toujours lui qui a la vision la plus précise de son heure locale. En fait, on a la propriété suivante :

  A tout instant physique, $\forall\ i, j,\quad vt_i[i] \geq vt_j[i]$

Les Horloges Vectorielles

Exemple



Remarque

- comme seul le processus $\mathbf{p_i}$ peut incrémenter la i[ème] composante du temps global, c'est toujours lui qui a la vision la plus précise de son heure locale. En fait, on a la propriété suivante :

  A tout instant physique, $\forall$ **i,j,** $\mathbf{vt_i[i]} \geq \mathbf{vt_j[i]}$

Les Horloges Vectorielles

Relations permettant de comparer deux horloges vectorielles **vh** et **vk** :

(1) $\mathbf{vh} \leq \mathbf{vk} \Leftrightarrow \forall\ \mathbf{i},\ \mathbf{vh[i]} \leq \mathbf{vk[i]}$

(2) $\mathbf{vh} < \mathbf{vk} \Leftrightarrow \mathbf{vh} \leq \mathbf{vk}\ \mathbf{ET}\ \exists\ \mathbf{i},\ \mathbf{vh[i]} < \mathbf{vk[i]}$

(3) $\mathbf{vh} \parallel \mathbf{vk} \Leftrightarrow \mathbf{NON(vh < vk)}\ \mathbf{ET}\ \mathbf{NON(vk < vh)}$

Soient **e** et **e'** deux événements de l'application répartie, **vt(e)** et **vt(e')** les horloges vectorielles associées. On a maintenant la relation d'équivalence suivante :

$$\mathbf{e} \rightarrow \mathbf{e'} \Leftrightarrow \mathbf{vt(e)} < \mathbf{vt(e')}$$

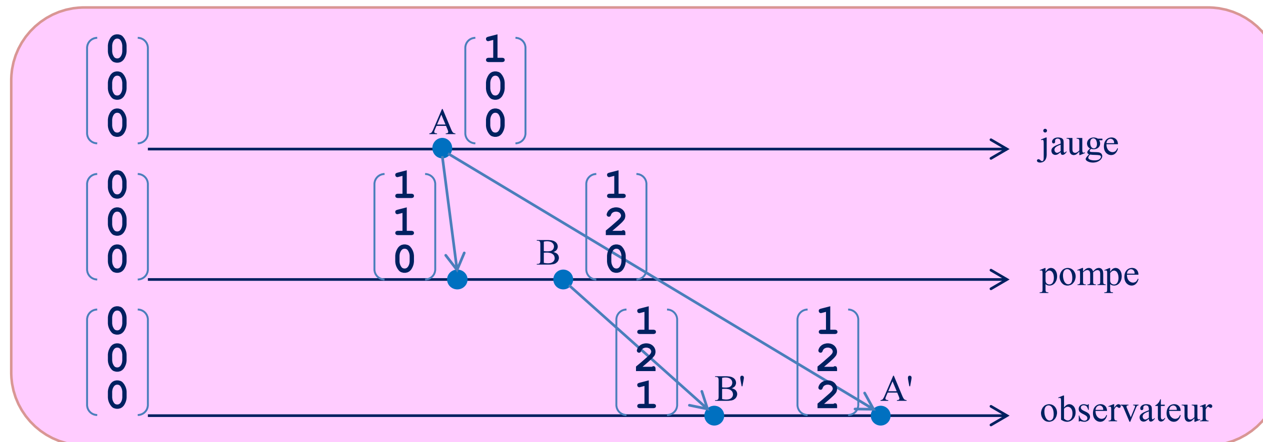
$$\mathbf{e} \parallel \mathbf{e'} \Leftrightarrow \mathbf{vt(e)} \parallel \mathbf{vt(e')}$$

Soient **e** et **e'** deux événements de l'application répartie, **vt(e)** et **vt(e')** les horloges vectorielles associées.

- Si $\mathbf{vt(e)} < \mathbf{vt(e')}$ alors $\mathbf{e} \rightarrow \mathbf{e'}$
- Si $\mathbf{vt(e')} < \mathbf{vt(e)}$ alors $\mathbf{e'} \rightarrow \mathbf{e}$
- Sinon, $\mathbf{e} \parallel \mathbf{e'}$

Les Horloges Vectorielles

Analyse de l'exemple de la fuite :



Conclusions :

- (1, 0, 0) < (1, 2, 0)

  [baisse de pression] → [augmentation d'activité]

- L'observateur peut affirmer qu'il y a une relation entre ces deux événements : le premier ayant entrainer le deuxième.

Limitations :

- stockage de l'information : un vecteur est une structure statique et se prête donc mal au stockage de l'information en cas de création dynamique de processus,
- transmission de l'information : si le nombre de processus impliqués dans l'application est important, la quantité d'information à ajouter à un message pour transmettre les vecteurs d'horloge est loin d'être négligeable.

Environnement Synchrone

Environnement synchrone pour une application répartie :

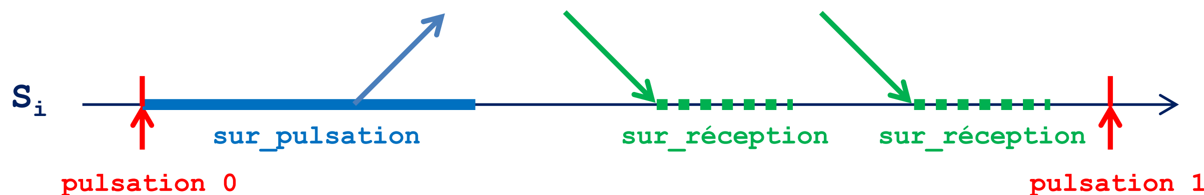
Hypothèse 1 : L'application de chaque site travaille sous forme de cycles numérotés. Chaque cycle est initié par le battement d'une pulsation.

Hypothèse 2 : La primitive exécutée lors du battement de la pulsation est notée **`sur_pulsation(numéro)`**. Son unique paramètre correspond au numéro de la pulsation courante. L'exécution de ce code est non bloquant et doit se terminer avant le battement de la pulsation suivante.

Hypothèse 3 : Durant l'exécution de **`sur_pulsation`**, un site émet au plus un seul message vers chaque autre site.

Hypothèse 4 : Un message émis lors de l'appel à **`sur_pulsation(p)`** est reçu et traité par le site récepteur après l'exécution de **`sur_pulsation(p)`** et avant le battement de la pulsation **`p+1`**.

Hypothèse 5 : Il n'y a pas d'émission de message dans les primitives **`sur_réception_de`**.



L'application travaille de manière synchrone.

Remarque : du moment que l'application se comporte globalement comme indiqué, il est inutile que la même pulsation soit simultanément battue sur deux sites différents.

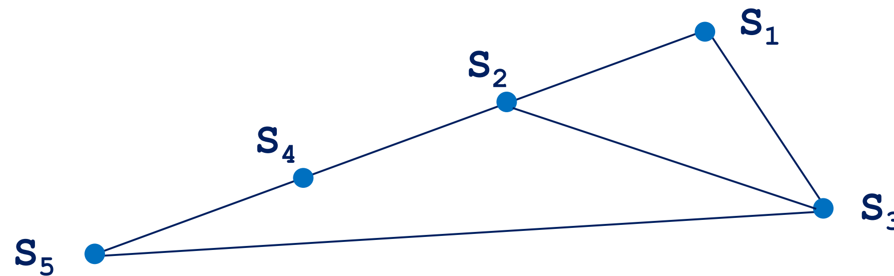
Environnement Synchrone

Apports du synchronisme :

- simplicité du code (de l'écriture des programmes),
- gain en complexité (diminution du nombre de messages échangés).

Exemple : Construction d'un arbre de plus courts chemins sur un graphe de communication (quelconque)

- Construction initiée par un site qui deviendra la racine de l'arbre
- le chemin qui conduit de l'initiateur à un site quelconque doit être l'un des plus courts chemins possibles parmi ceux qui les relient dans le graphe de communication



Algorithme en Environnement Asynchrone

Principe de l'algorithme :

- lorsqu'un site est rattaché à l'arbre (cad qu'il possède un père), il propose à ses voisins de devenir ses fils,
- on adjoint à la proposition la nouvelle distance à la racine qu'obtiendra le nœud sollicité (puisque recherche du plus court).

Variables du site **i** :

- $\mathbf{voisins_i}$: sous-ensemble des sites voisins de **i**. Cette constante définit le graphe de communication.
- $\mathbf{père_i}$: identité du site "père" dans l'arbre. Initialisée pour l'initiateur à sa propre identité (facilité de programmation), non initialisée pour les autres sites.
- $\mathbf{distance_i}$: variable contenant la longueur du chemin allant du site jusqu'à la racine. Initialisée à **0** pour l'initiateur et à l'infini (**+∞**) pour les autres sites (ou une constante supérieure ou égale au nombres de sites).

```
construire(){
Pour tout j ∈ voisins_i\{père_i}
    envoyer_à(j, (cons, distance_i+1));
Fin pour
}
```

```
sur_réception_de(j,(cons,distance)){
Si distance_i > distance Alors
    père_i = j;
    distance_i = distance;
    construire();
}
```

Complexité en Environnement Asynchrone

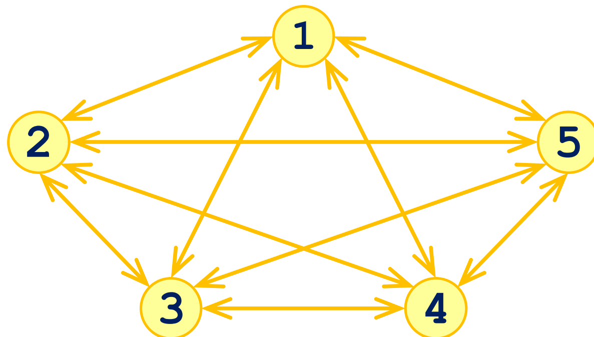
Mesure de la complexité calculée en nombre de messages échangés dans le pire cas (cas de **n** sites) :

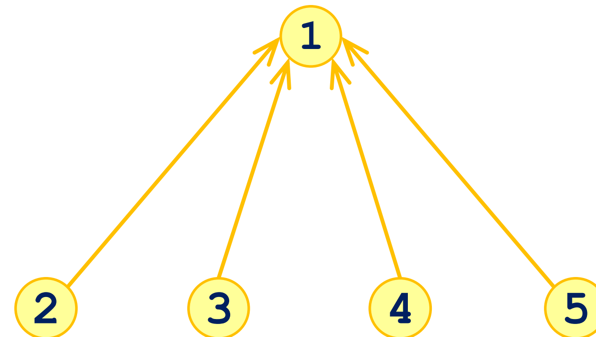
- l'initiateur appelle exactement une fois **construire** ⇒ envoi de **n-1** messages
- à chaque appel à **construire**, un site envoie au plus **n-2** messages (le père est exclu de l'envoi)
- à chaque fois qu'un site appelle **construire**, sa distance décroît. La valeur maximale (différente de l'infini) que celle-ci peut prendre est **n-1**. Donc, un site peut changer au plus **n-1** fois de valeurs. Par conséquent, il appellera au plus **n-1** fois **construire**.

nombre de messages échangés $\leq$ **(n-1)+(n-1).(n-1).(n-2) = Θ(n³)**

Exhibition d'un scénario d'exécution dont le nombre de messages échangés est de l'ordre de **n³**.

- Exemple de graphe de communication totalement maillé (une clique) :
  Dans une clique, le résultat de l'algorithme est nécessairement un arbre de hauteur **1**.





Complexité en Environnement Asynchrone

Scénario d'exécution (cas d'une clique) :

- le site **1** est l'initiateur et envoie **n-1** messages. Nous supposons qu'à l'exception du message au site **2**, tous les autres messages sont retardés,
- à la réception, le site **2** prend comme père le site **1** et envoie **n-2** messages. Nous supposons de même que tous les messages sont retardés excepté celui envoyé au site **3**.
- en itérant ce procédé, on construit un arbre (non stabilisé) qui n'est autre qu'un chemin qui parcourt les sites de **1** à **n**.

	Site	**Distance**				
n-1	**1**	+∞	0	0	…	0
n-2	**2**	+∞	1	1	…	1
n-2	**3**	+∞	2	1	…	1
n-2	**4**	+∞	3	2	…	1
	⋮	⋮	⋮	⋮	⋮	1
n-2	**i**	+∞	i-1	i-2	…	1
	⋮	⋮	⋮	⋮	⋮	1
n-2	**n**	+∞	n-1	n-2	…	1

Messages échangés					
n-1					
n-2					
n-2	n-2				
…	…				
⋮	⋮	⋮			
n-2	n-2	…	n-2		
⋮	⋮	⋮			
n-2	n-2	…	…	…	n-2

Un site **i** (autre que **1**) appelle exactement **i-1** fois **construire**, provoquant l'envoi de **(i-1).(n-2)** messages.

Messages échangés de ce scénario = $(n-1)+(n-2)\sum_{i=1}^{n-1} i = (n-1)+(n-2)\frac{n(n-1)}{2} = \Theta(n^3)$

Algorithme en Environnement Synchrone

Principe de l'algorithme :

- remarquer que, puisque l'environnement est synchrone, la 1ère proposition de rattachement est nécessairement la meilleur,
- seule difficulté : la détection au début d'une pulsation qu'un site vient d'être rattaché à l'arbre. On remarque : le site racine est rattaché à la pulsation **0**, les sites à distance **1** le sont à la pulsation **1**, …, les sites à distance **d** le sont à la pulsation **d**.
- il suffira alors de comparer la distance du site à la valeur de la pulsation pour savoir quand proposer à ses voisins le rattachement à l'arbre.

Variables du site **i** :

- $\mathbf{voisins}_i$: sous-ensemble des sites voisins de **i**. Cette constante définit le graphe de communication.
- $\mathbf{père}_i$: identité du site "père" dans l'arbre. Initialisée pour l'initiateur à sa propre identité (facilité de programmation), non initialisée pour les autres sites.
- $\mathbf{distance}_i$: variable contenant la longueur du chemin allant du site jusqu'à la racine. Initialisée à **0** pour l'initiateur et à l'infini (**+∞**) pour les autres sites .
- $\mathbf{pulsation}_i$: indice de la pulsation courante, initialisée à **0**.

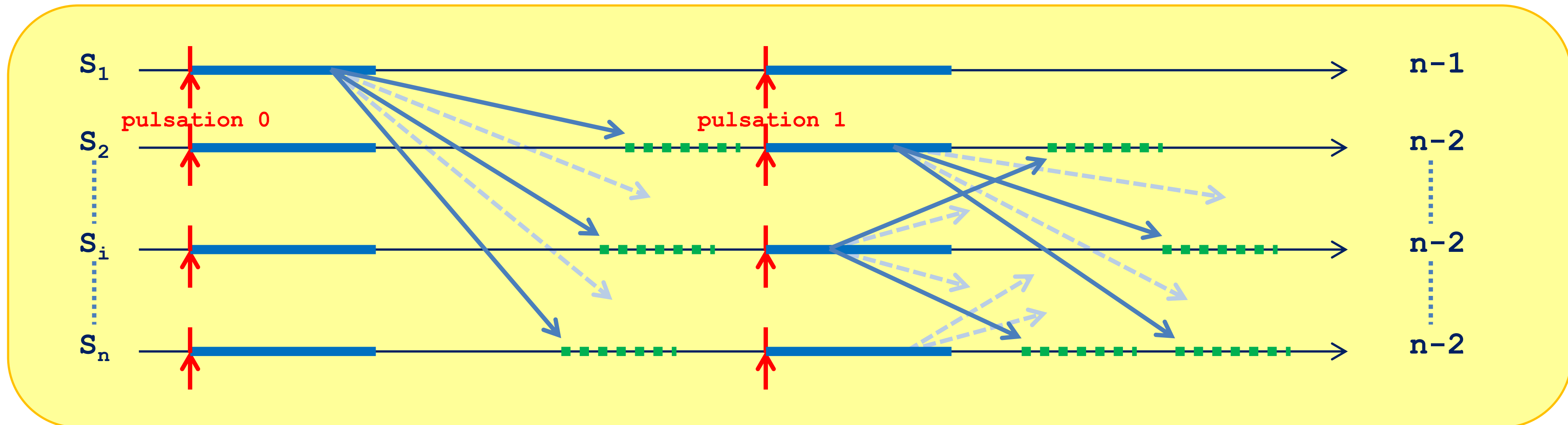
```
sur_pulsation(pulsation_i){
Si pulsation == distance_i Alors
    Pour tout j ∈ voisins_i\{père_i}
        envoyer_à(j, (cons, distance_i+1));
    Fin pour
Fsi
}
```

```
sur_réception_de(j,(cons,distance)){
Si distance_i == +∞ Alors
    père_i = j;
    distance_i = distance;
Fsi
}
```

Complexité en Environnement Synchrone

Scénario d'exécution (cas d'une clique) :

- un site n'envoie qu'une proposition de construction à tous ses voisins excepté son père,
  - le site **1** est l'initiateur et envoie **n-1** messages,
  - tout autre site envoie **n-2** messages.



Messages échangés de ce scénario = $(n-1)+(n-1).(n-2) = (n-1)^2 = \Theta(n^2)$

Gain d'un ordre de grandeur par rapport à l'algorithme asynchrone.

Émulation d'un Environnement Synchrone

Apports du synchronisme mais les environnements asynchrones sont plus répandus :

- émulation d'un environnement synchrone au dessus d'un environnement asynchrone

Difficultés de mise en œuvre

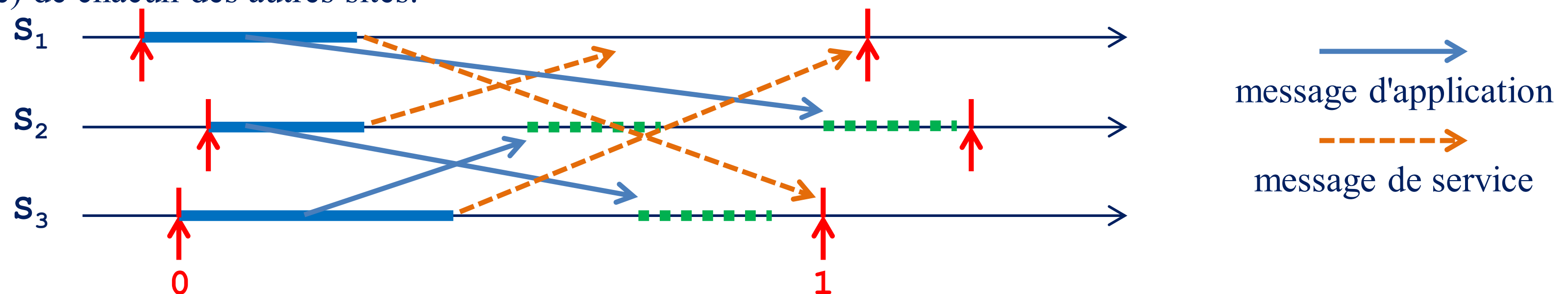
Détermination de l'instant où le service pourra battre la pulsation suivante. Avant de battre la pulsation, deux conditions doivent être remplies :

- le code associé à la pulsation doit avoir été exécuté

  → fournir à l'application une primitive de l'interface qui lui permet d'indiquer que le traitement de la pulsation courante est terminé. On la notera **`fin_traitement()`**.

- tous les messages envoyés par les autres sites à destination de ce site doivent être reçus et traités. Impossibilité de savoir, dans le cas d'un réseau asynchrone, lorsqu'un message n'est pas reçu d'un site donné si le message n'a pas été envoyé ou s'il est encore en transit.

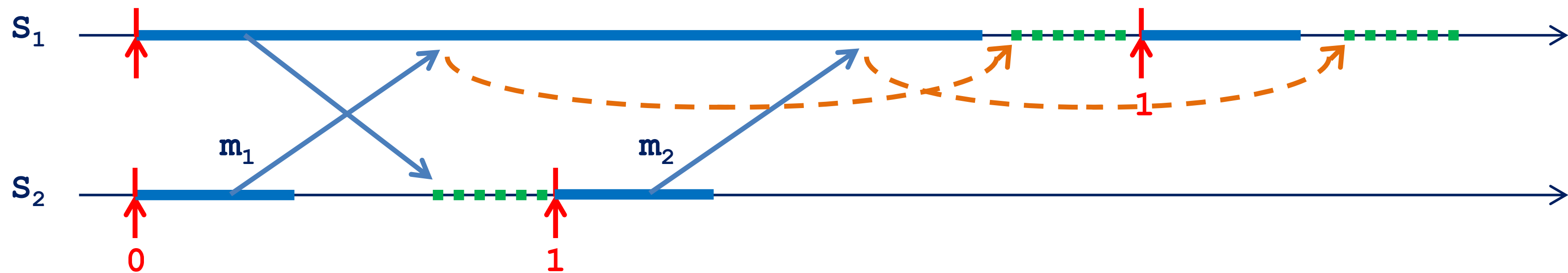
  → coordonner les services : chaque service observe les messages émis par son application lors du traitement de la pulsation. A la fin du traitement, il détermine les sites qui ne recevront pas de messages de son application et envoie à leur service des messages de contrôle. Ainsi, chaque site est assuré de recevoir un message (d'application ou de service) de chacun des autres sites.



Émulation d'un Environnement Synchrone

Difficultés de mise en œuvre

Conservation des messages pouvant arriver trop tôt :

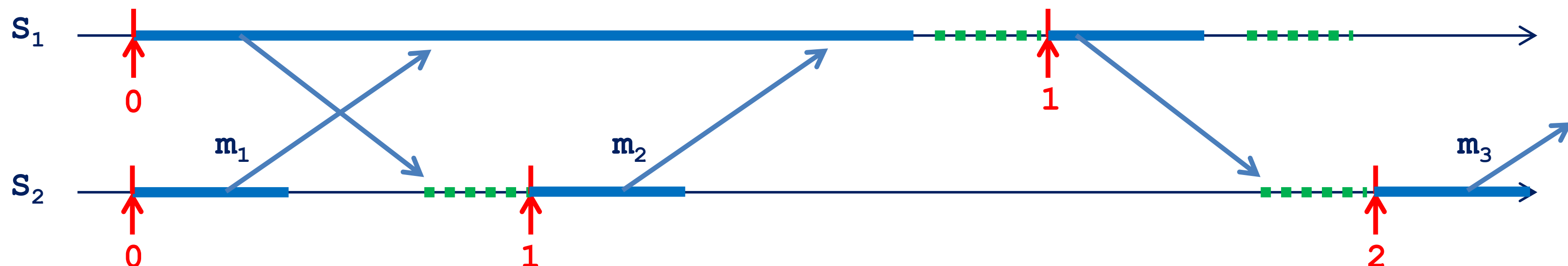


– aucun des deux messages **m1** et **m2** ne peut être délivré immédiatement :

m1 doit attendre la fin du traitement de la pulsation courante,

m2 doit attendre la fin du traitement de la pulsation suivante.

– combien de messages au maximum le service d'un site doit conserver ?



Réalisation de l'Émulation

Variables du site **i** :

- **pulsation**$_i$: indice de la pulsation courante, initialisée à **0**.
- **traitement_en_cours**$_i$: booléen qui indique si le site exécute le code associé à la pulsation, initialisé à **vrai**.
- **nbmess_reçus**$_i$: nombre de messages reçus par **i** pour la pulsation courante, initialisé à **0**.
- **nbmess_avance**$_i$: nombre de messages reçus par **i** pour la pulsation suivante, initialisé à **0**.
- **à_envoyer**$_i$**[1..N]** : tableau de booléens initialisés à **vrai**. **à_envoyer**$_i$**[j]** est vrai si le service doit envoyer un message de service à **j** à la fin du traitement de la pulsation. **N** est le nombre de sites de l'application.
- **courant**$_i$**[1..N]** : tampon de stockage des messages reçus pour la pulsation courante en attente d'être délivrés à l'application. **courant**$_i$**[j]** contient éventuellement le message provenant de **j**. Toute cellule de ce tableau est composée des deux champs **<présent, contenu>**. **présent** est un booléen initialisé à **faux** indiquant si un message est contenu dans la cellule. **contenu** représente les données du message.
- **avance**$_i$**[1..N]** : tampon de stockage des messages reçus en avance. La structure et l'initialisation de ce tableau sont identiques à celles du tableau précédent.

Réalisation de l'Émulation

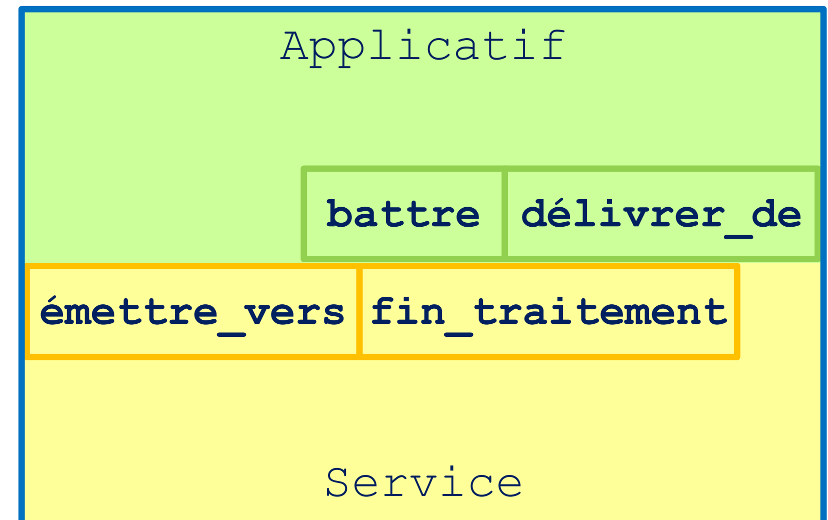
Interface application/service :

Primitives de services appelées par l'application

- **émettre_vers(j,m)** : envoie un message vers **j** dans l'environnement émulé.
- **fin_traitement()** : indique au service que le traitement **sur_pulsation** est terminé.

Primitives d'application appelées par le service

- **délivrer_de(j,m)** : délivre à l'application un message provenant de **j**.
- **battre(pulsation$_i$)** : relance le processus applicatif pour traitement de la pulsation courante.



Réalisation de l'Émulation

Algorithme du site **i** :

```
émettre_vers(j,m){
envoyer_à(j,(app,pulsation_i,m));
à_envoyer_i[j]=faux;
}
```

Ajout d'informations de contrôle au message émis.

```
test_and_set_pulsation(){
Si (traitement_en_cours_i == faux) ET (nbmess_reçus_i == n-1) Alors
    pulsation_i++;
    traitement_en_cours_i = vrai;
    nbmess_reçus_i = nbmess_avance_i;
    nbmess_avance_i = 0;
    Pour j de 1 à N Faire
        Si avance_i[j].présent == vrai Alors
            courant_i[j] = avance_i[j];
            avance_i[j].présent = faux;
        Fsi
    Fin pour
    battre(pulsation_i);
Fsi
}
```

test_and_set_pulsation est une procédure interne appelée par **sur_réception** et **fin_traitement** pour vérifier les conditions de battement d'une pulsation :

- l'application a fini et le service reçoit le dernier message,
- le service a reçu tous les messages et l'application finit son traitement.

Tâches accomplies :

- réinitialisation et maj des variables pour le début du cycle,
- transfert des messages de $\textbf{avance}_i$ vers $\textbf{courant}_i$,
- battre la pulsation.

Réalisation de l'Émulation

```
sur_réception_de(j,(tp,pulse,cont)){
Si pulse > pulsation_i Alors
    nbmess_avance_i++;
    Si tp == app Alors
        avance_i[j].présent = vrai;
        avance_i[j].contenu = cont;
    Fsi
Sinon
    nbmess_reçus_i++;
    Si tp == app Alors
        Si traitement_en_cours_i == faux Alors
            délivrer_de(j,cont);
        Sinon
            courant_i[j].présent = vrai;
            courant_i[j].contenu = cont;
        Fsi
    Fsi
    test_and_set_pulsation();
Fsi
}
```

Trois cas sont possibles lorsqu'un message **m** est reçu par le service :

- **m** arrive en avance et il est stocké dans **avance$_i$**,
- **m** arrive dans la bonne pulsation mais le traitement est en cours, il est alors stocké dans **courant$_i$**,
- **m** arrive à la bonne pulsation et le traitement est fini, il est alors délivré à l'application.

La réception d'un message de service n'entraine que sa comptabilisation.

Comme la réception d'un message de la pulsation courante peut entrainer la fin de la pulsation, cette primitive appelle **test_and_set_pulsation**.

Réalisation de l'Émulation

```
fin_traitement(){
Pour j de 1 à N Faire
    Si (à_envoyeri[j] == vrai) ET (i != j) Alors
        envoyer_à(j,(serv,pulsationi,Ø));
    Sinon
        à_envoyeri[j] = vrai;
    Fsi
    Si couranti[j].présent == vrai Alors
        délivrer_de(j, couranti[j].contenu);
        couranti[j].présent = faux;
    Fsi
Fin pour
traitement_en_coursi = faux;
test_and_set_pulsation();
}
```

A l'appel de la primitive **fin_traitement**, le service délivre les messages stockés dans **courant**$_i$.

Il envoie aussi les messages de service aux sites pour lesquels l'application n'a pas envoyé de message.

Comme il peut s'agir de la fin de la pulsation, il appelle **test_and_set_pulsation**.

Références

Cours d'Algorithmique Répartie, Joyce El Haddad et Serge Haddad, Chapitres I à VIII, Université Paris-Dauphine.

Le Temps dans un Système Réparti, C. Dutheillet, Support de cours, Laboratoire MASI, Université Paris 6

[Mat89] F. Mattern "Virtual time and global states of distributed systems", In : Parallel and Distributed Algorithms, M. Cosnard et al. eds, North-Holland, 1989, pp 215-226

[Lam78] L. Lamport "Time, clocks and the ordering of events in a distributed system", Communications of the ACM 21 (1978) pp 558-564

[Awe85] B. Awerbuch "Complexity of network synchronization" JACM 32 (1985), 804-823

[Tou80] S. Toueg "An all-pairs shortest-path distributed algorithm" Technical Report RC 8327, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, 1980

[Tel00] G. Tel "Introduction to Distributed Algorithms" Second Edition Cambridge University Press. ISBN 0-521-79483-8. 2000