Chapitre 3

Causalité
Horloges de Lamport

Causalité

Problème : on a besoin de savoir si

un événement « arrive avant un autre »

Difficultés :

asynchronie, pas de temps global.

→ **Définir des critères, sans quantifier le temps**

L. LAMPORT (1978)

Ordre partiel sur les événements

ordre partiel ?

= ordre CAUSAL

(un événement est la cause d'un autre)

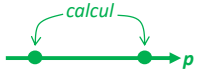
Ordre causal

Trois types d'**événements**

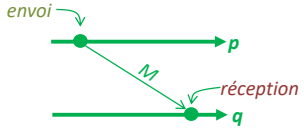
- **un** pas de **calcul interne**
- **envoi** de message
- **réception** de message

Scénarios

deux **événements** ayant lieu sur le même processus en séquence



un événement **envoi** et un événement **réception**



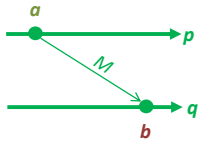
Sur ces scénarios :
on peut dire qu'un événement arrive avant un autre
sans quantifier le temps

Ordre causal : relation sur les événements

- Soit ***a*** et ***b***, deux événements
  - s'exécutant sur le même processus
  - en séquence
  - ***b*** s'exécute directement après ***a*** : ***« a ; b »***

On pose : $a <_C b$

- Soit ***a*** et ***b***, deux événements
  - ***a*** est l'envoi d'un message
  - ***b*** est la réception de ce même message



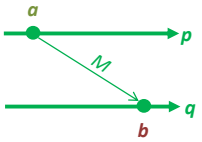
On pose : $a <_C b$

- L'ordre est obtenu par **fermeture transitive** de ces deux règles de base

fermeture transitive ?

Ordre causal : relation sur les événements

On pose : $a <_C b$



On pose : $a <_C b$

+ fermeture transitive

Remarque : on dit que $<_C$ est un *préordre strict ou un ordre partiel strict*
c'est-à-dire :

Preuve ?

- → $<_C$ est une relation **irréflexive** : *on n'a jamais* $a <_C a$
- → $<_C$ est une relation **asymétrique** : *si* $a <_C b$ *alors* ***non***$(b <_C a)$
- → $<_C$ est une relation **transitive** : *si* $a <_C b$ *et* $b <_C c$ *alors* $a <_C c$

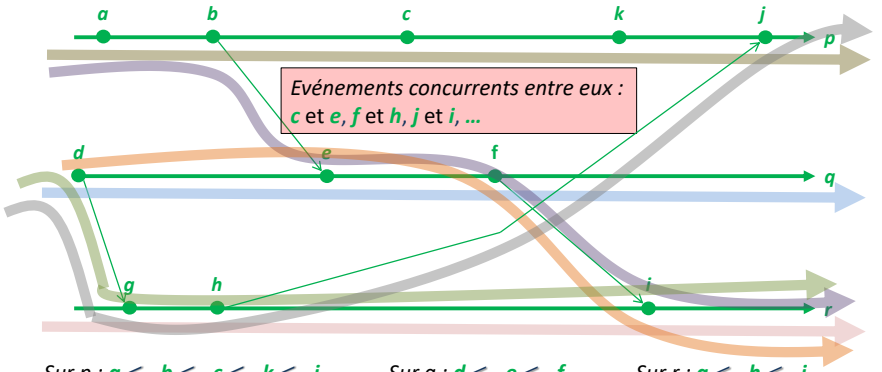
Certains éléments *ne sont pas comparables :*

on peut avoir ***non***$(a <_C b)$ ***et non***$(b <_C a)$

dans ce cas on dit que *a* et *b* sont **concurrents**

Ordre causal : exemple

Comparer les événements 2 à 2 selon l'ordre causal



Sur p : $a <_C b <_C c <_C k <_C j$ *Sur q :* $d <_C e <_C f$ *Sur r :* $g <_C h <_C i$

Envoi / réception : $b <_C e$, $d <_C g$, $f <_C i$, $h <_C j$

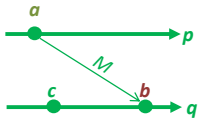
Fermeture transitive : $a <_C b <_C e <_C f <_C i$, $d <_C e <_C f <_C i$, $d <_C g <_C h <_C i$
$d <_C g <_C h <_C j$

Horloges logiques

- → Utiliser l'ordre causal pour construire des algorithmes **corrects**
- → Idée : **numéroter les** événements
- → Chaque processus localement doit pouvoir numéroter les événements qui lui arrivent

Les « numéros » doivent <u>refléter l'ordre causal</u> !!

Si num de *a* vaut **3**, num de *b* ??



Si num de *a* vaut **3** et num de *c* **5**, num de *b* ??

Ces « numéros » sont des **dates logiques** des événements
= qui servent à ordonner ; ≠ quantité de temps

→ On attribue à chaque processus une *variable* = **Horloge logique**
qui va lui permettre de calculer des dates logiques des événements

Calcul des dates logiques pour un processus **p**

Variable : $C_p \in \mathbb{N}$ *est l'****horloge logique*** *de* ***p***

Initialisation : $C_p \leftarrow 0$;

Règle 1 : pour un événement **interne** ou un **envoi de message**

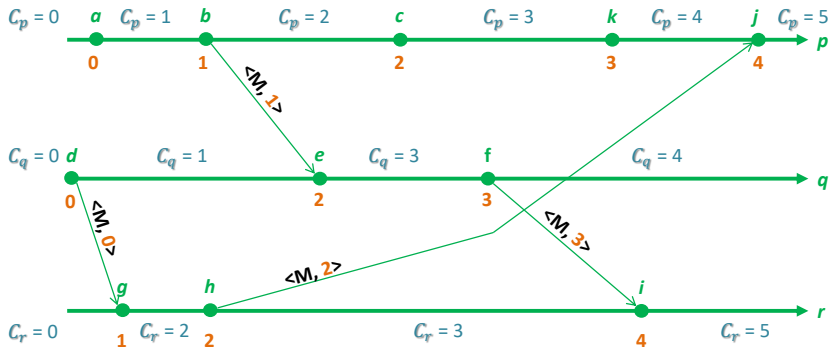
- **Attribuer** la date C_p à l'événement
- Si c'est un événement interne, exécuter l'événement
- Si c'est un envoi de message **M**, exécuter : Envoyer <**M**, C_p>
- $C_p \leftarrow C_p + 1$;

Règle 2 : pour une **réception de message**

- Exécuter : Réceptionner <**M**, **d**>
- $C_p \leftarrow \max(C_p, d + 1)$;
- **Attribuer** la date C_p à l'événement
- $C_p \leftarrow C_p + 1$;

Reprendre l'exemple précédent pour calculer les dates logiques de chaque événement

Ordre causal : exemple



Proposition : pour deux événements ***a*** et ***b***,
si on a $a <_C b$,
alors date logique de ***a*** < date logique de ***b***

Discuter de la preuve

Ordre total sur les événements <<

→ ***Ordonner totalement tous les événements d'une exécution***
Y compris les événements concurrents

- Soit ***a*** et ***b***, deux événements ayant des dates logiques différentes
  - On ordonne alors ***a*** et ***b*** selon leur date

Exemple : si date de ***a*** vaut 2 et date de ***b*** vaut 3 alors ***a << b***

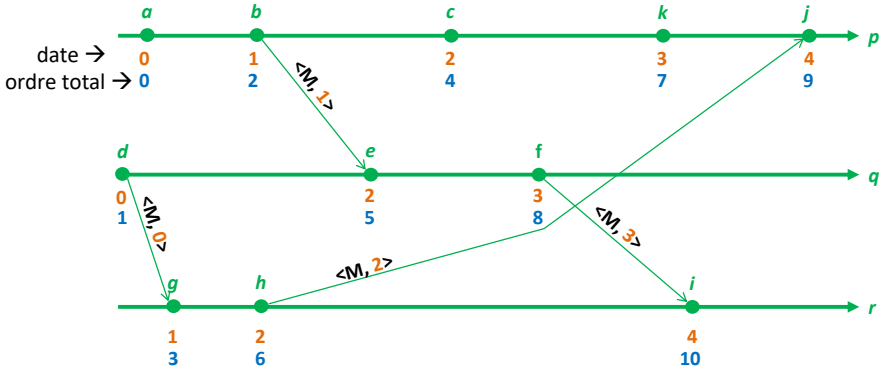
- Soit ***a*** et ***b***, deux événements ayant la même date logique
  - *NB : **a** et **b** ne peuvent s'être exécutés sur le même processus !*
  - On suppose que **les processus sont ordonnés totalement**
  - On ordonne alors ***a*** et ***b*** selon l'ordre de leurs processus

Exemple : si date de ***a*** et date de ***b*** valent 3,
si ***a*** s'exécute sur le processus **p** et ***b*** s'exécute sur le processus **q**
et si **p** est prioritaire sur **q** alors ***a << b***

- La relation << est obtenue par **fermeture transitive** de ces deux règles de base

Ordre total << : exemple

Reprendre l'exemple précédent pour calculer l'ordre << sur les événements
(on peut numéroter les événements)
On suppose que p < q < r



Ordre total sur les événements <<

En résumé : ***a*** et ***b***, deux événements
a << b si et seulement si
*date **a** < date **b** OU*
*date **a** = date **b** ET proc **a** < proc **b***

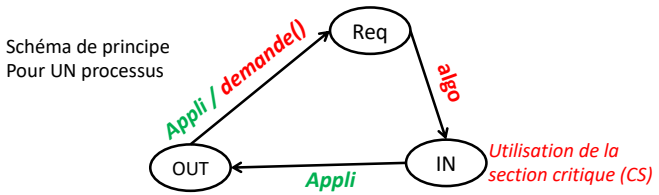
Preuve

Propriété : La relation << est un *ordre strict* **total**
c'est-à-dire : pour toute paire d'événements distincts ***a, b*** ($a \neq b$)
on a soit ***a << b***, soit ***b << a***

Propriété :
Soient ***a*** et ***b***, deux événements.
Si $a <_C b$, alors ***a << b***.

Evident
Mais c'est exactement ce qu'on voulait !!

Algorithme d'exclusion mutuelle de Lamport
Spécification



Spécification

Sûreté : ***« Rien de mal ne peut jamais arriver »***

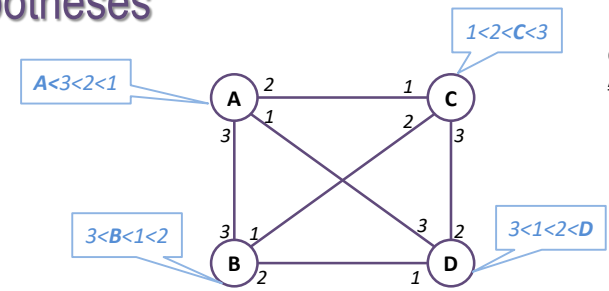
→ *Il y a toujours au plus un processus en CS*

Vivacité : ***« Quelque chose de bien finit par arriver »***

→ Tout processus demandeur (Req) finit par entrer en CS

(= en temps fini)

Algorithme d'exclusion mutuelle de Lamport
Hypothèses



- Topologie = graphe complet bidirectionnel (au moins 2 nœuds)
- Canaux FIFO
- Pas de faute, tout asynchrone
- Ordre total sur les processus
  - → Les processus n'ont pas besoin d'être identifiés
  - → Mais leurs canaux sont ordonnés de façon à représenter le même ordre pour tous les processus

Algorithme d'exclusion mutuelle de Lamport

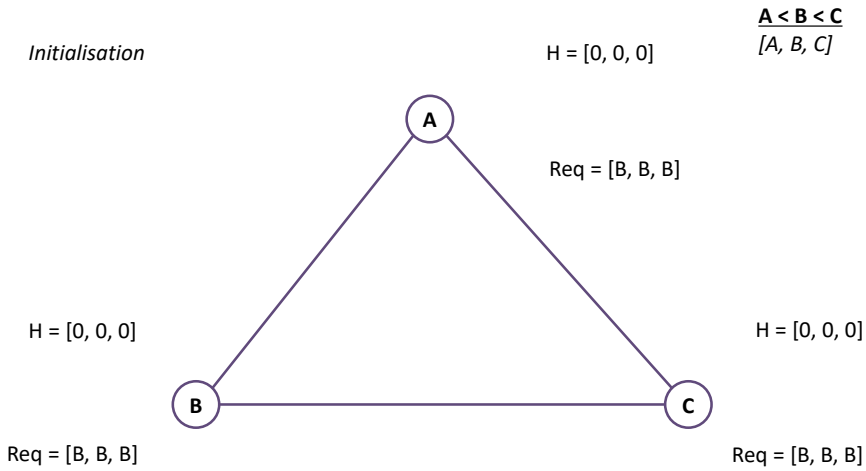
Principe

- **Tableau des horloges locales** des autres processus
- Les valeurs de ces horloges sont envoyées en même temps que les messages
- Le processus qui obtient la CS est le minimum selon <<

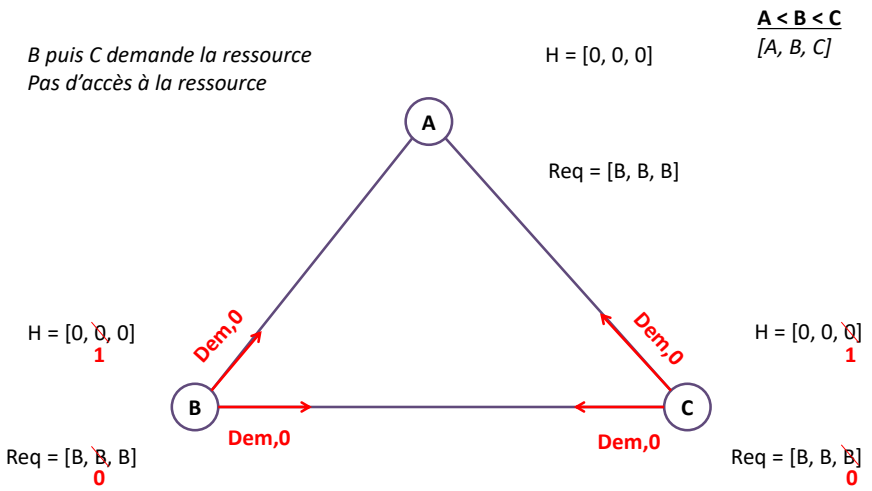
(le processus prioritaire qui attend depuis le plus longtemps)

→ ***Voir algorithme***

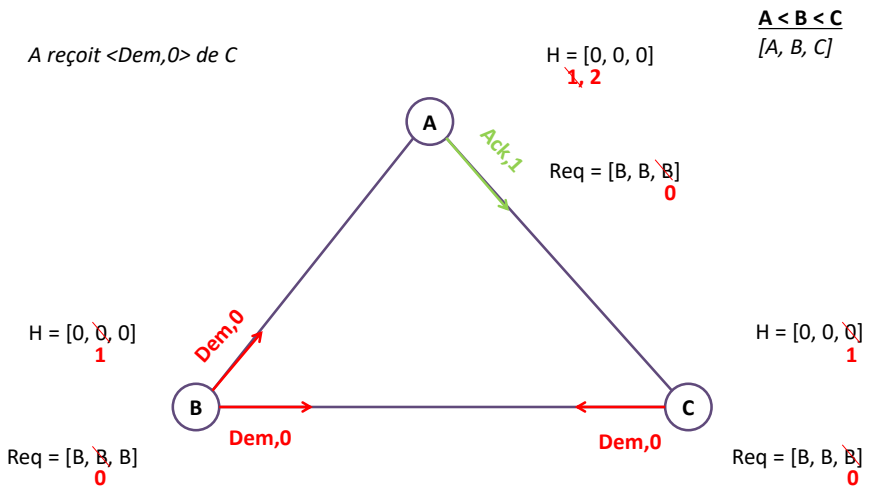
Exclusion mutuelle de Lamport - Exemple



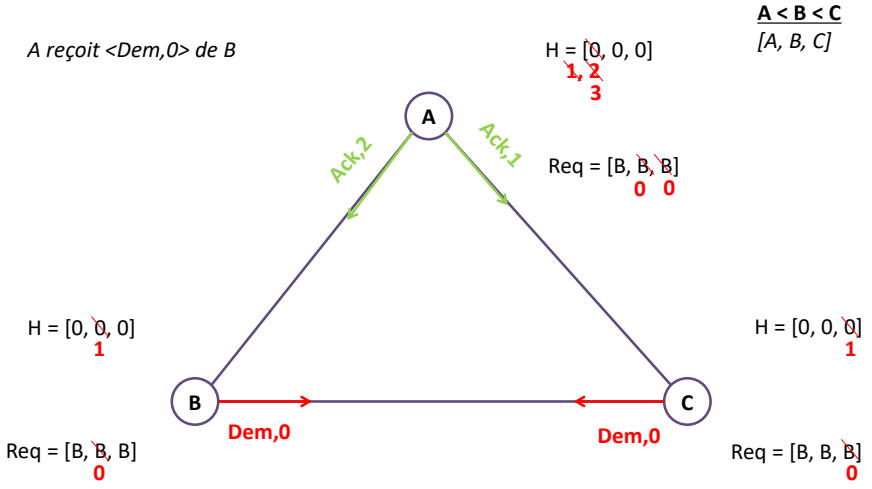
Exclusion mutuelle de Lamport - Exemple



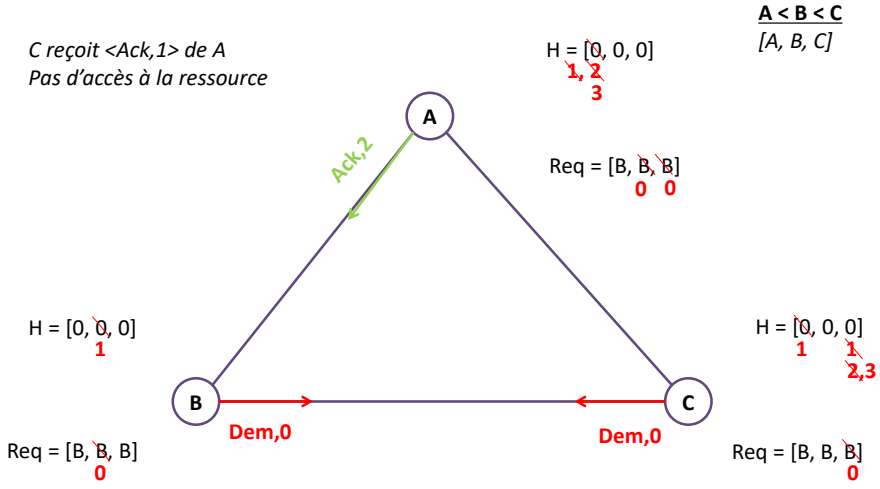
Exclusion mutuelle de Lamport - Exemple



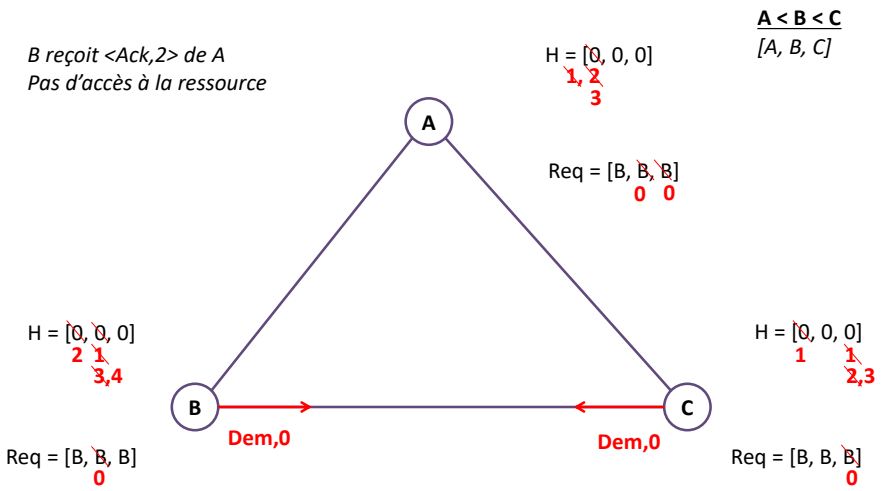
Exclusion mutuelle de Lamport - Exemple



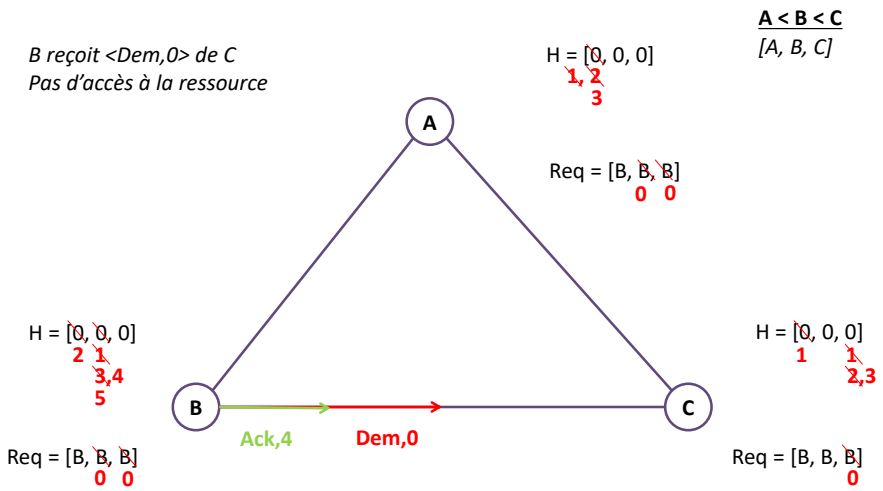
Exclusion mutuelle de Lamport - Exemple



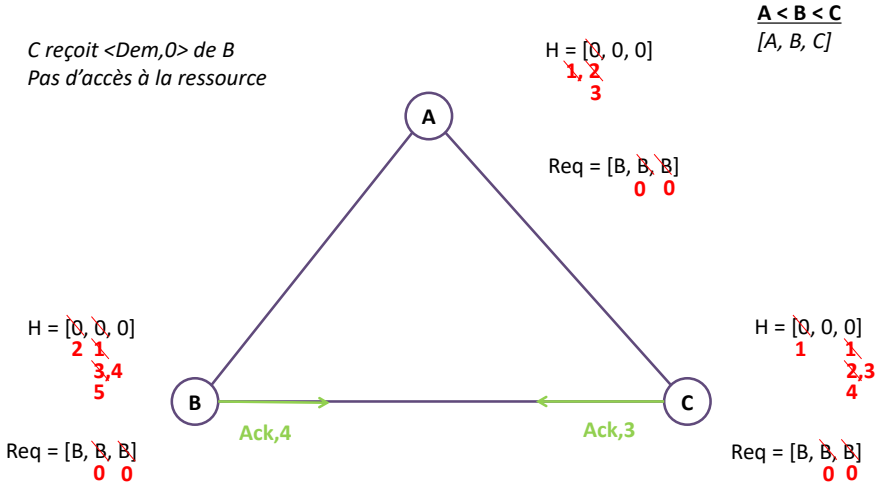
Exclusion mutuelle de Lamport - Exemple



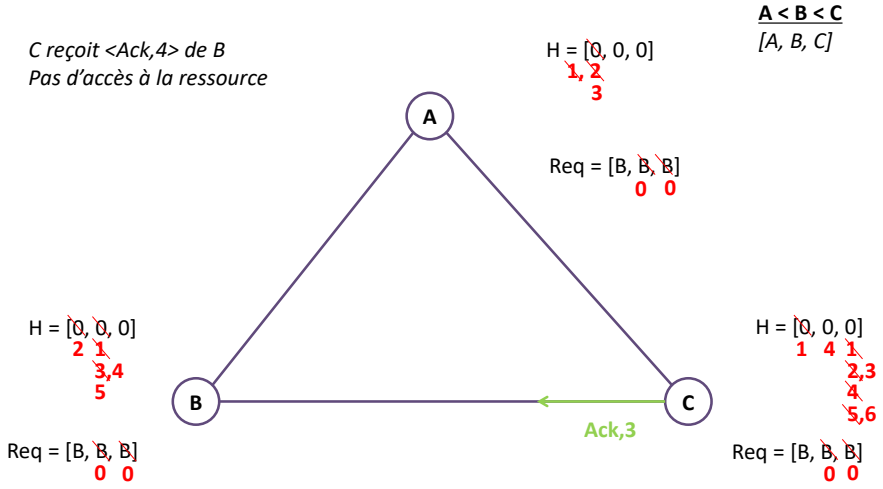
Exclusion mutuelle de Lamport - Exemple



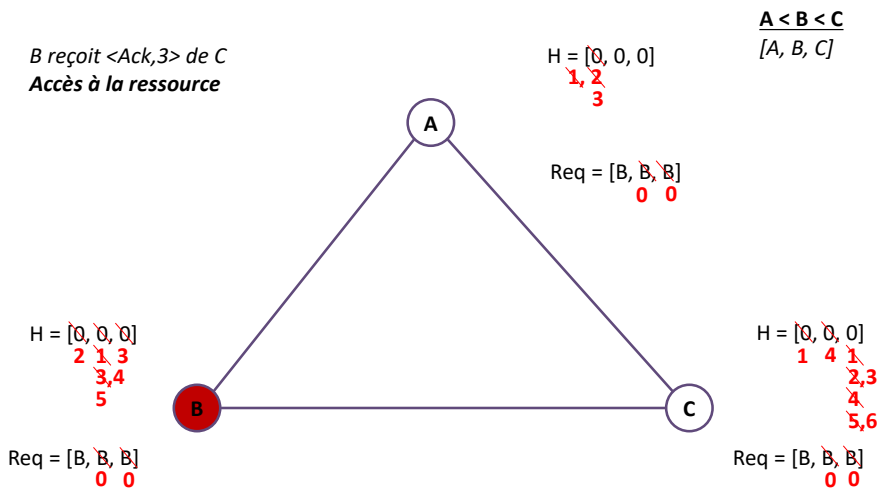
Exclusion mutuelle de Lamport - Exemple



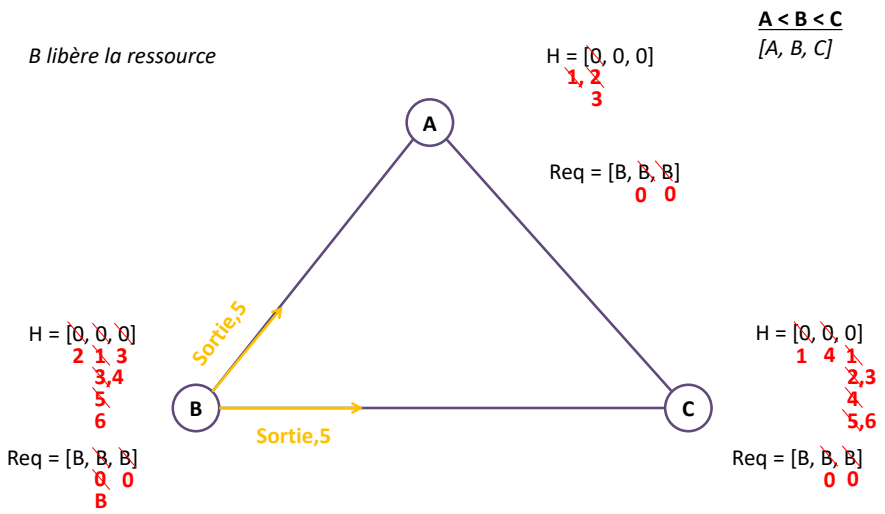
Exclusion mutuelle de Lamport - Exemple



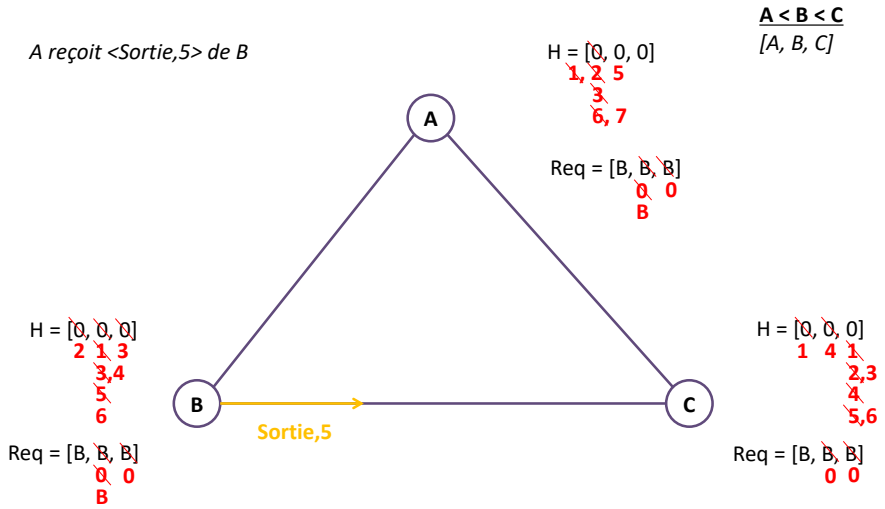
Exclusion mutuelle de Lamport - Exemple



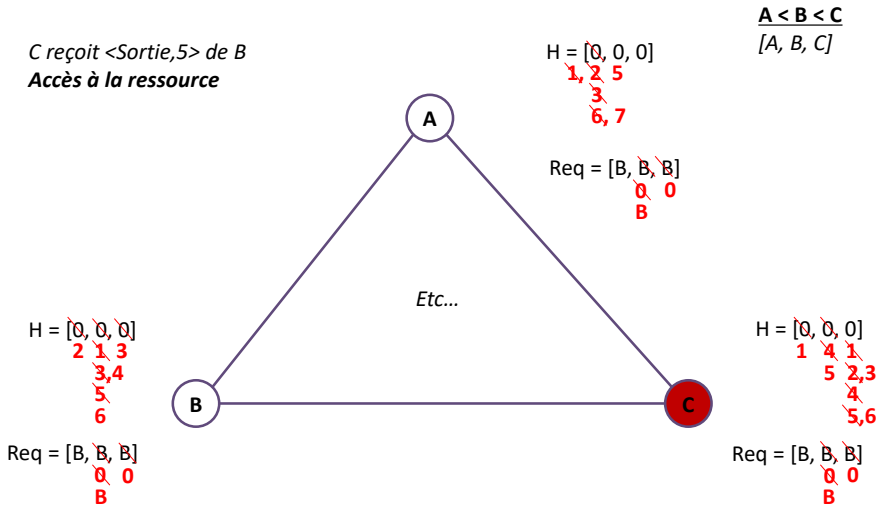
Exclusion mutuelle de Lamport - Exemple



Exclusion mutuelle de Lamport - Exemple



Exclusion mutuelle de Lamport - Exemple



Exclusion mutuelle de Lamport
Preuve de correction

Lemme 1 : quand une demande est servie, elle est minimale selon <<, à cet instant, parmi les demandes non encore servies.

Preuve au tableau

Lemme 2 : seul un nombre fini de demandes peut être servi avant qu'une demande ne soit servie effectivement.

Preuve au tableau

Corolaire (Vivacité) : tout processus demandeur est servi en temps fini.

Lemme (Sûreté) : jamais plus d'un processus n'est en section critique en même temps.

Preuve au tableau

Théorème : l'algorithme résout le problème de l'exclusion mutuelle dans un réseau complet dont les processus sont totalement ordonnés.

Exclusion mutuelle de Lamport - Complexité

On compte : le nombre de messages envoyés

On note : *n le nombre de nœuds dans le réseau*

Nombre de messages par demande : *3(n – 1)*

Temps de service = *nombre de processus pouvant exécuter la CS avant qu'un processus ne l'obtienne :* *(n – 1)*

Ratio nombre de messages / nombre de demandes : *3(n – 1)*

Solution « réactive », algorithmes à permission