

---

# **Exclusion Mutuelle en Réparti**

Master informatique UPMC 2016-2017  
UE AR (4I403)

# Plan

---

- **Exclusion mutuelle en réparti**

- Types d'algorithmes
- Propriétés
- Classes d'algorithmes

- **Algorithmes à permission**

- Lamport
    - Ricart-Agrawala
    - Maekawa (quorum)

- **Algorithmes à jeton**

- Martin
    - Raymonde
    - Naimi-Trehel
    - Susuki-Kasami

# 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.

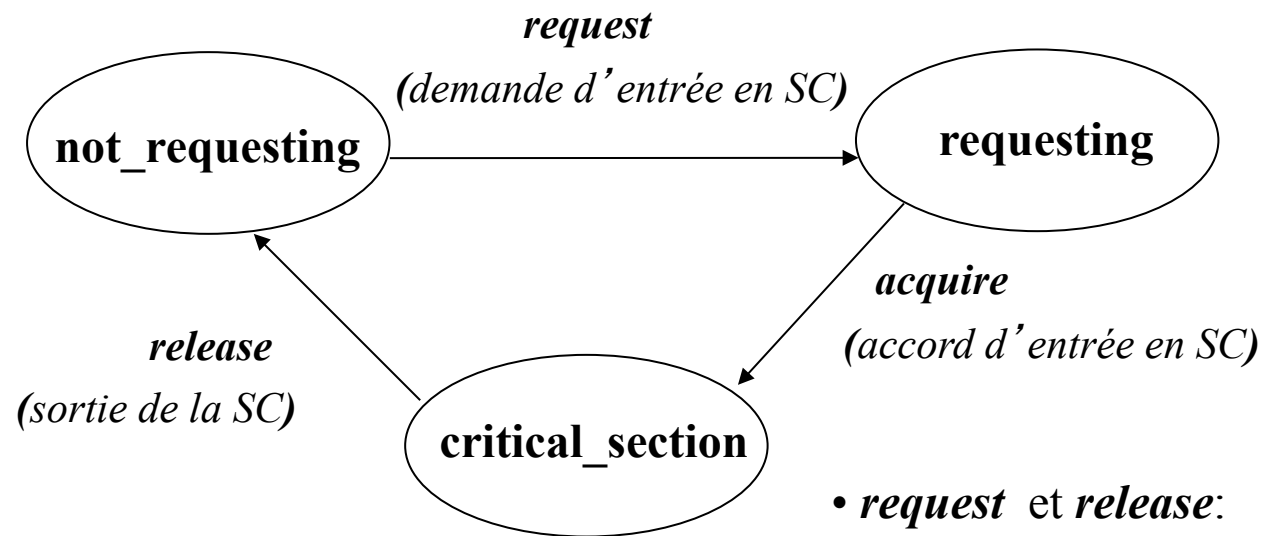
## ■ Exclusion Mutuelle en réparti:

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

# Transitions d'un processus

- **N processus:  $P_1, \dots, P_N$**

- $\text{état}_i : \{requesting, not\_requesting, critical\_section\}$



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

# 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.

# 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  $j \neq i$ ;*
- **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.**

# 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.

# Algorithme centralisé x réparti

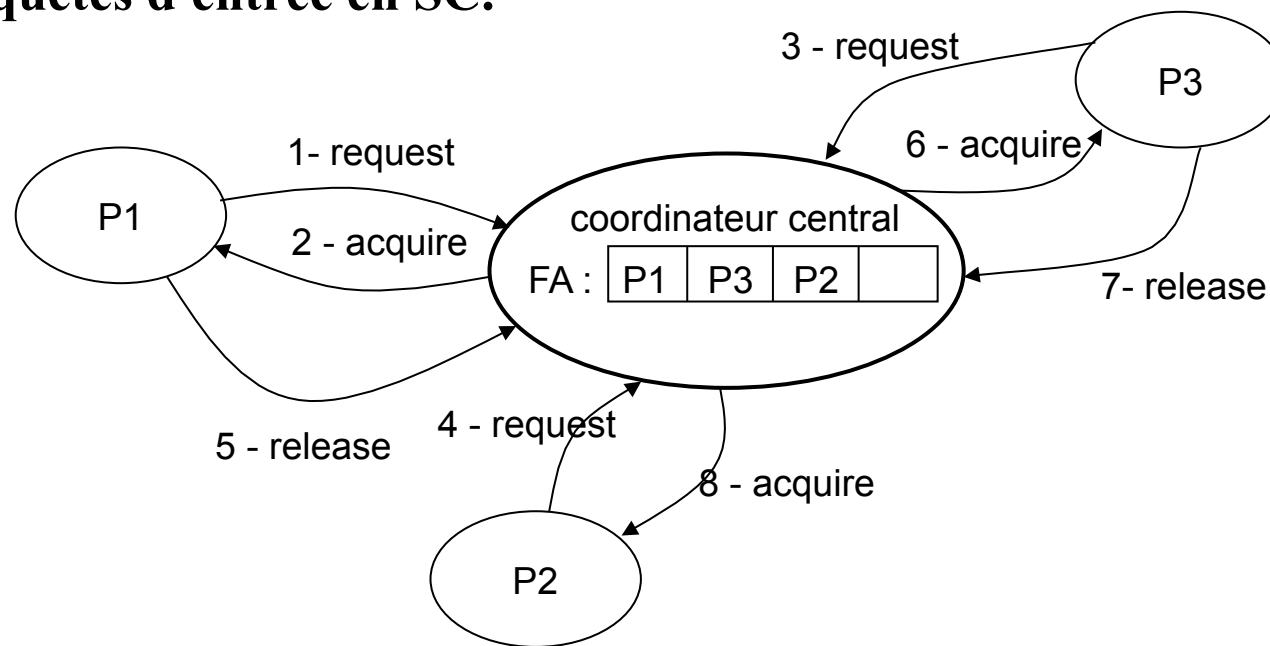
---

- **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.



# 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.



# 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.

# 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.

# Algorithmes à base de Permission

---

- Lamport
- Ricart-Agrawala
- Maekawa (quorum)

# 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  $P_i$ , complétée par son identifiant :  $(H_i, i)$ .
      - $(H_i, i) < (H_j, j) \Leftrightarrow (H_i < H_j \text{ ou } (H_i = H_j \text{ et } i < j))$
    - 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  $At_i$ .

# 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.

# Algorithme de Lamport

## Tous les processus $S_i$ :

### Variables Locales:

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

### Request\_CS( $S_i$ ) :

- $H_i = ++$ ;
- Placer sa requête  $req_i$  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  $req_i = head(FA_i)$ );

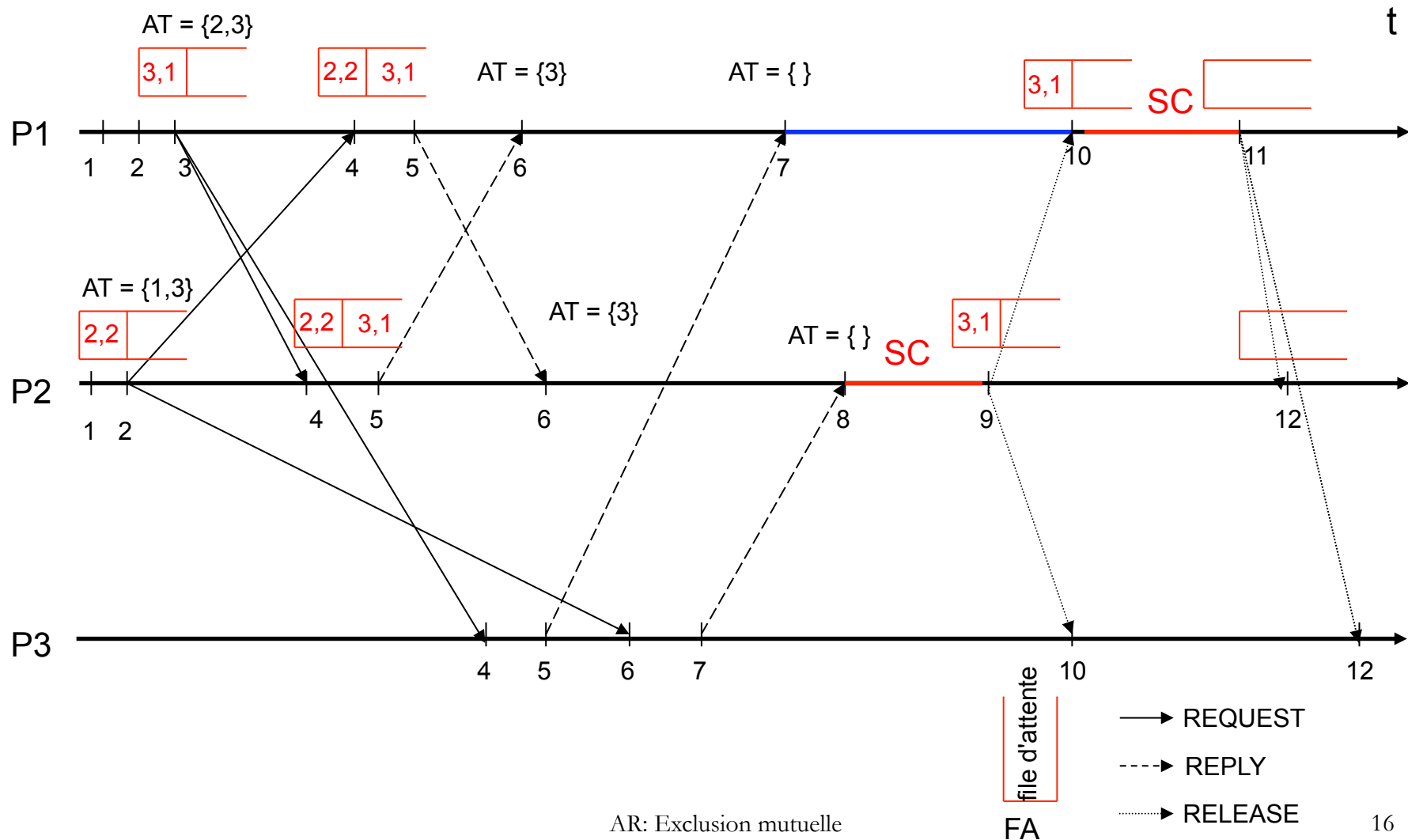
### Release\_SC( $S_i$ ) :

- $H_i ++$ ;
- Diffuser un message *RELEASE* à tous les autres sites ( $R_i - \{S_i\}$ );
- Enlever sa requête  $req_i$  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 :  $(FA_i \cup \{msg\ S_j\})$ ;
  - 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  $(FA_i - \{msg\ S_j\})$ ;}

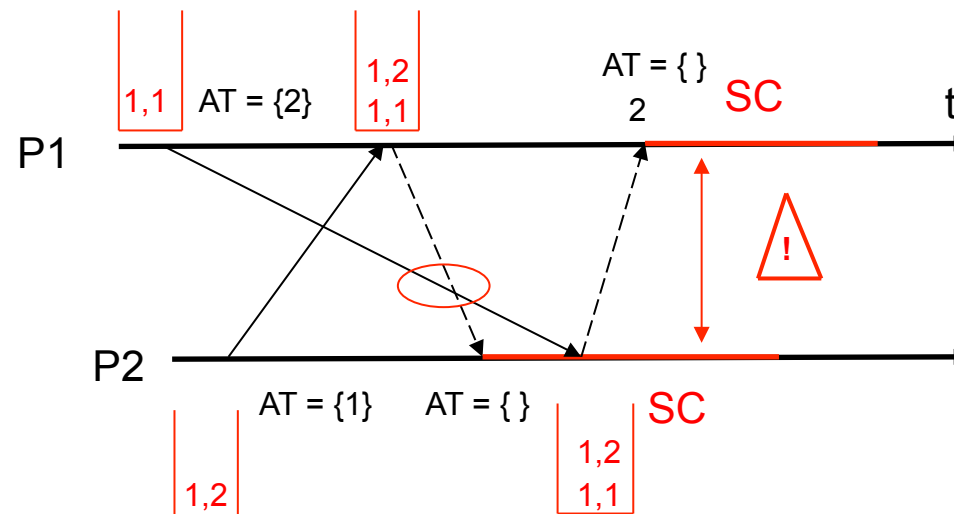
# Algorithme de Lamport (exemple)





# 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.

# 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.

# 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.

# 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'acquiesce une requête que s'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.

# 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 :
  - $(\text{type}, (H_i, S_i))$ ;

## ■ Variables Locales du processus $P_i$ :

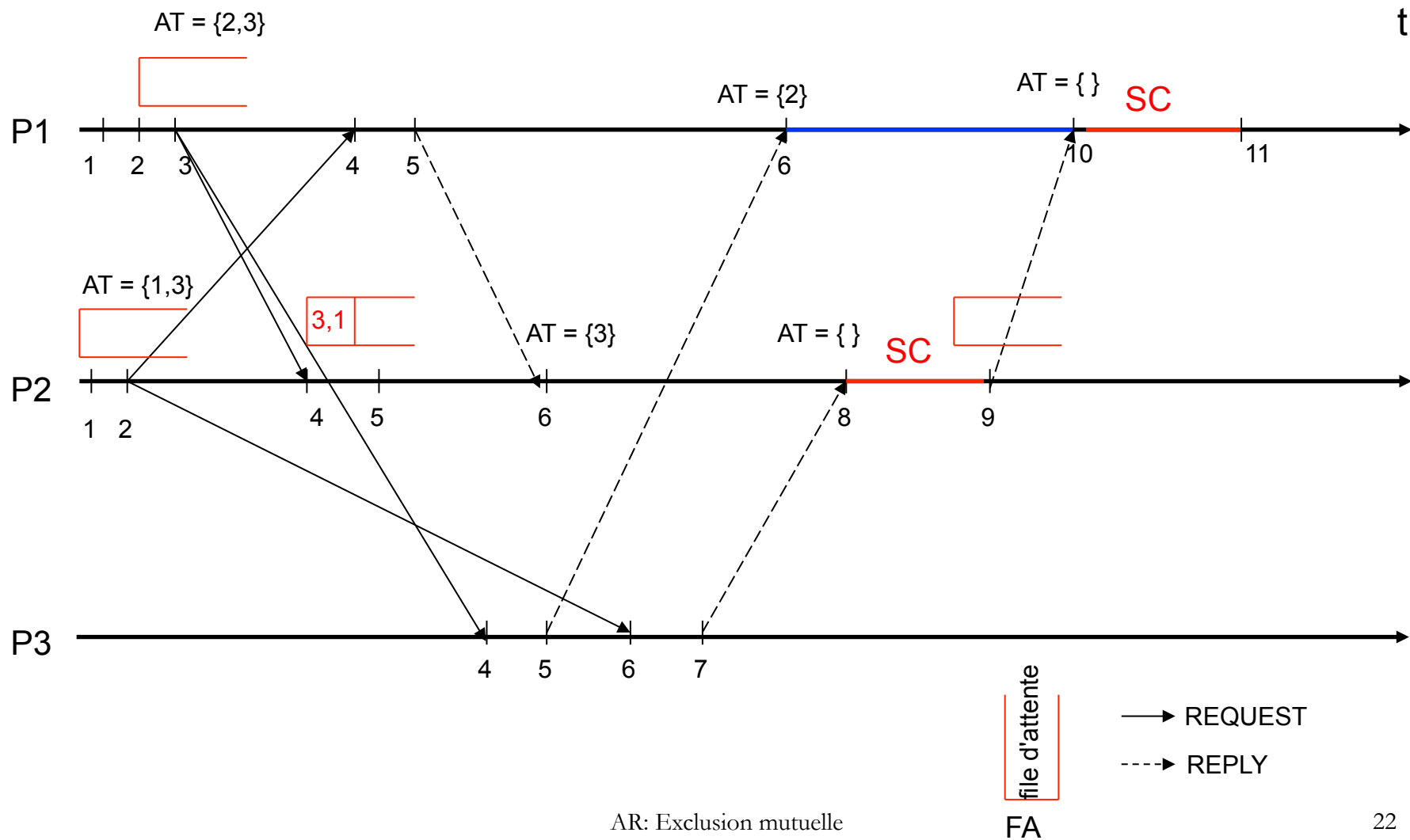
- $H_i$  : Horloge logique scalaire
- $FA_i$  : File d'attente
- $At_i$  : Attente de permission.
- $Etat_i$  : *requesting*, *not\_requesting*, *critical\_section*.

## ■ Algorithme

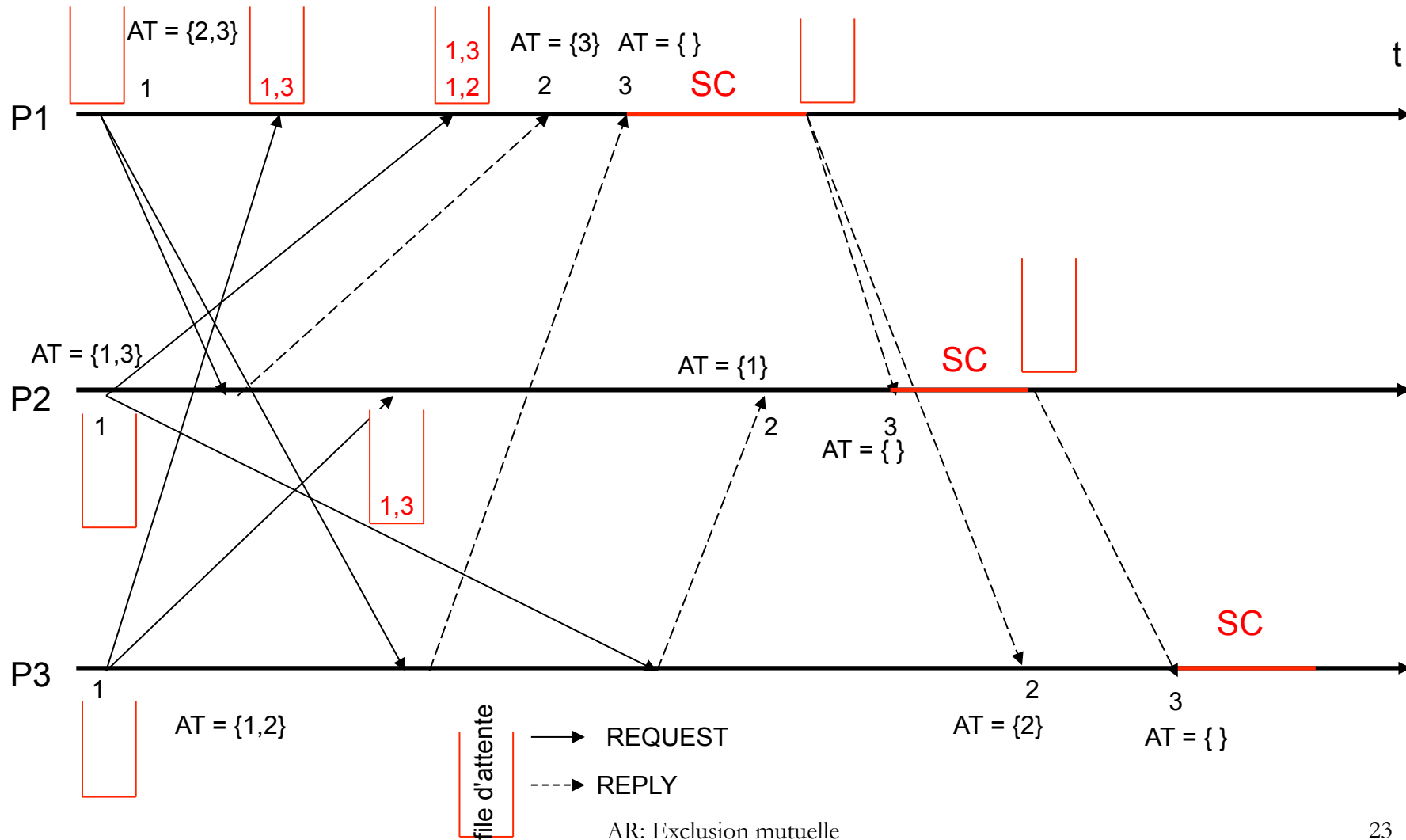


- Sera vu en TD et TME.

# Algorithme de Ricart/Agrawala (exemple 1)



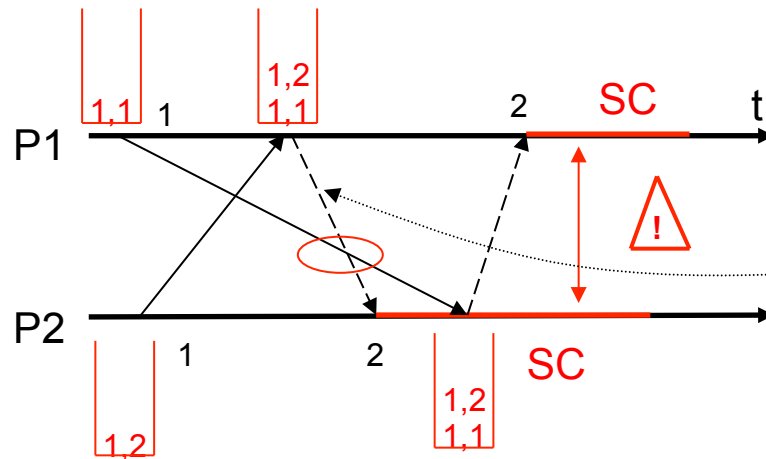
# Algorithme de Ricart/Agrawala (exemple 2)



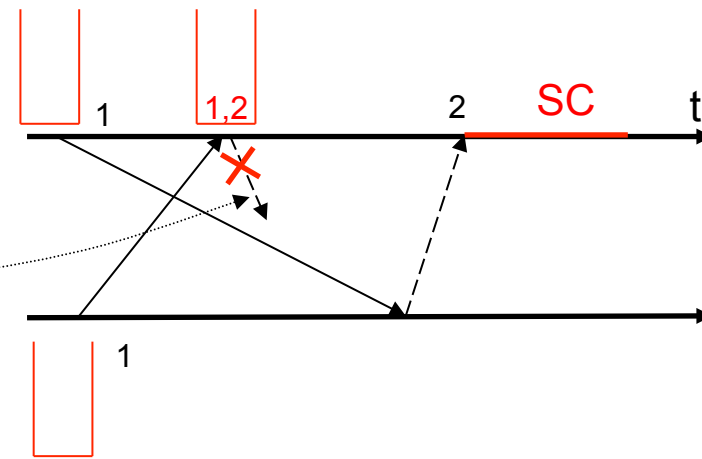
# Algorithme de Ricart/Agrawala

## ■ 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.



Lamport 78



Ricart et Agrawala 81



# 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.

# 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.

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

# 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:**

- $|RS_1| = |RS_2| = |RS_3| \dots = |RS_N| = K$
- $\forall S_i \in \{S_1, \dots, S_N\}, S_i \in RS_i$
- $\forall i, j \in \{1, \dots, N\}$  tels que  $i \neq j$ ,  
 $S_i$  et  $S_j$  appartiennent à  $D$   $RS$   
/\* même nombre d'ensembles \*/
- $D = K$  est une possibilité

# Algorithme de Maekawa (quorum)

---

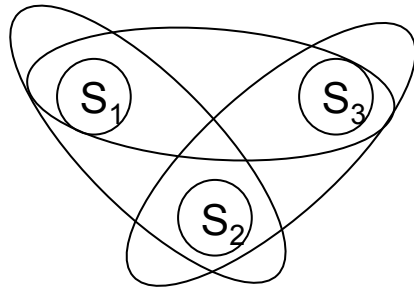
## ■ Exemples de quorum

$$RS_1 = \{S_1, S_2\}$$

$$RS_2 = \{S_2, S_3\}$$

$$RS_3 = \{S_3, S_1\}$$

$N=3, K=2$



$$RS_1 = \{S_1, S_2, S_3\}$$

$$RS_2 = \{S_2, S_4, S_6\}$$

$$RS_3 = \{S_3, S_5, S_6\}$$

$$RS_4 = \{S_4, S_1, S_5\}$$

$$RS_5 = \{S_5, S_2, S_7\}$$

$$RS_6 = \{S_6, S_1, S_7\}$$

$$RS_7 = \{S_7, S_3, S_4\}$$

$N=7, K=3$

# Algorithme de Maekawa

---

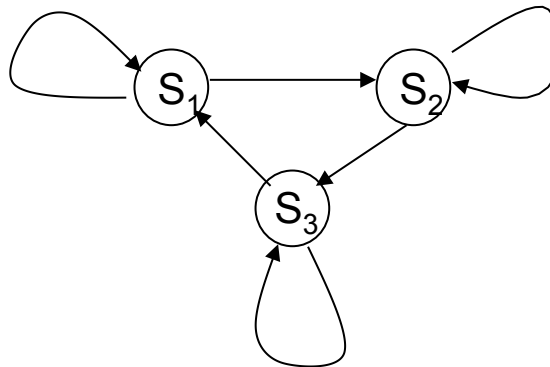
- Pour entrer en SC, le site  $S_i$  doit verrouiller tous les membres de son ensemble  $RS_i$  en leur envoyant un message du type REQUEST.
  - En recevant un msg REQUEST de  $S_j$ , si  $S_i$  ne se trouve pas déjà verrouillé,  $S_i$  envoie son accord (msg *LOCKED*) à  $S_j$  et se verrouille au profit 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  $RS_i$ , 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  $RS_i$ .

# 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.



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

$$N=3, K=2$$

# Algorithme de Maekawa

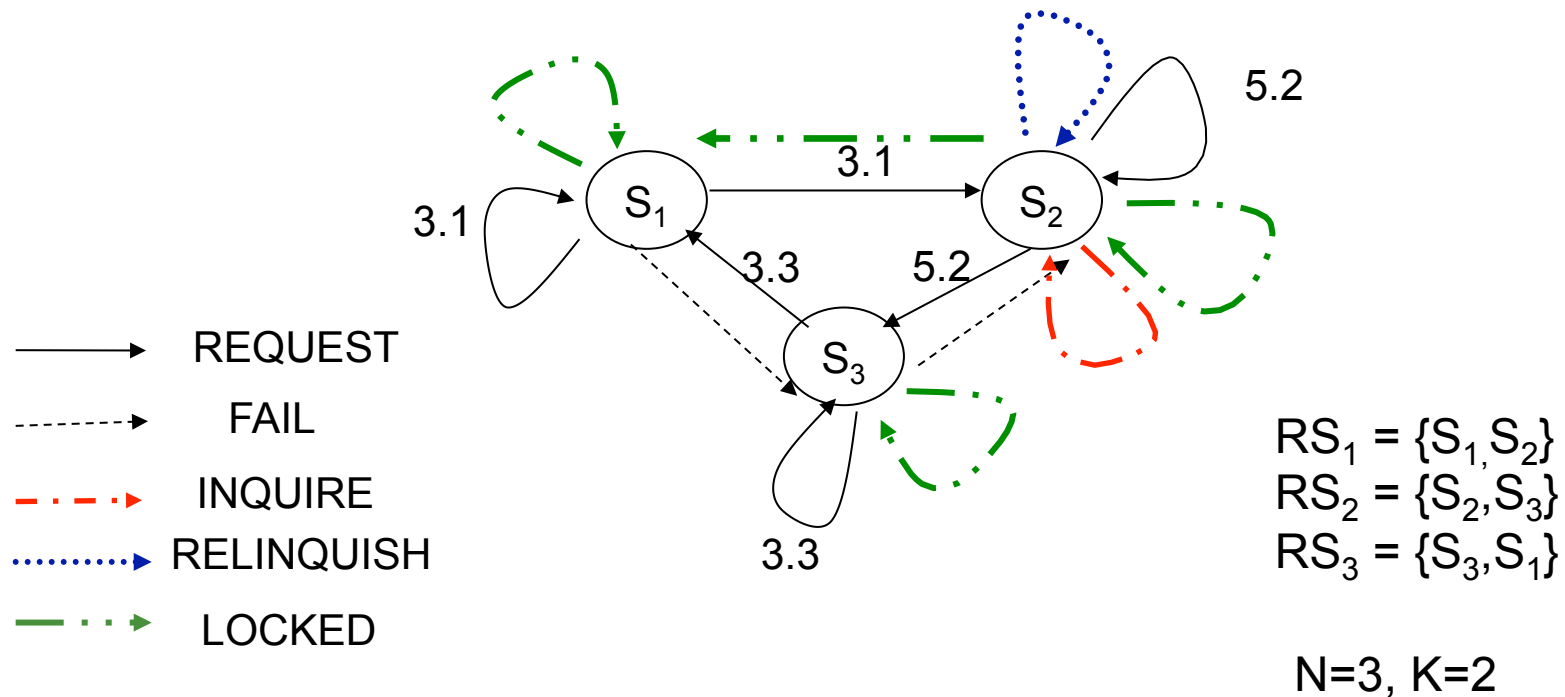
---

## ■ Solution pour le problème d'interblocage

- Dater les messages : ordre total
  - Horloge de Lamport + identifiant
- Reprendre la permission accordée si la nouvelle demande est antérieure à celle déjà satisfaite
  - 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).

# Algorithme de Maekawa

## ■ Solution pour le problème d'interblocage





# Algorithme de Maekawa

---

## ■ Contenu des messages :

- $(\text{type}, (H_i, S_i))$  : 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  $S_j$

# 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  $S_j$  ( $S_i$  était verrouillé au profit de  $S_j$ ).

### ➤ *RELINQUISH*

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

# 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() :**

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

### **Variables Locales:**

$FA_i = \emptyset$ ;

$H_i = 0$ ;

$At_i = \emptyset$ ;

# Algorithme de Maekawa

---

## Reception (msg de $S_j$ ) :

- *REQUEST* :
  - Mettre à jour  $H_i$  ( $H_i = \max(H_i, H_j) + 1$ );
  - 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  $S_k$  \*/ {
    - ( $FA_i \cup \{msg\ S_j\}$ ); /\* insérer 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  $S_j$}
- *LOCKED* :
  - ( $At_i = At_i - \{S_j\}$ ); /\* comptabiliser la réception d'une permission en plus \*/

# Algorithme de Maekawa

---

*INQUIRE* :

Si un message du type *FAIL* a été reçu

Envoyer message *RELINQUISH* à  $S_j$ ; ( $At_i = At_j \cup \{S_j\}$ );

*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;

$FA_i - \{S_k\}$ ; /\*retirer la requête  $S_k$  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

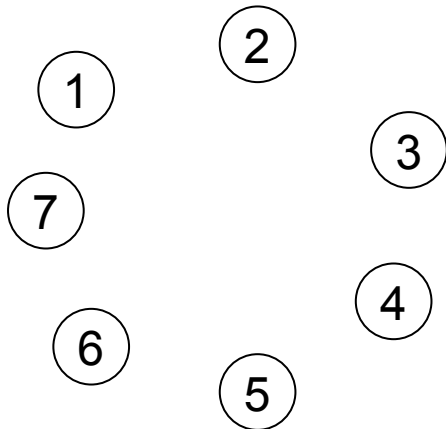
envoyer msg *RELINQUISH* à  $S_k$  ( $At_i = At_j \cup \{S_k\}$ );

}

# Algorithme de Maekawa

---

## ■ Exemple



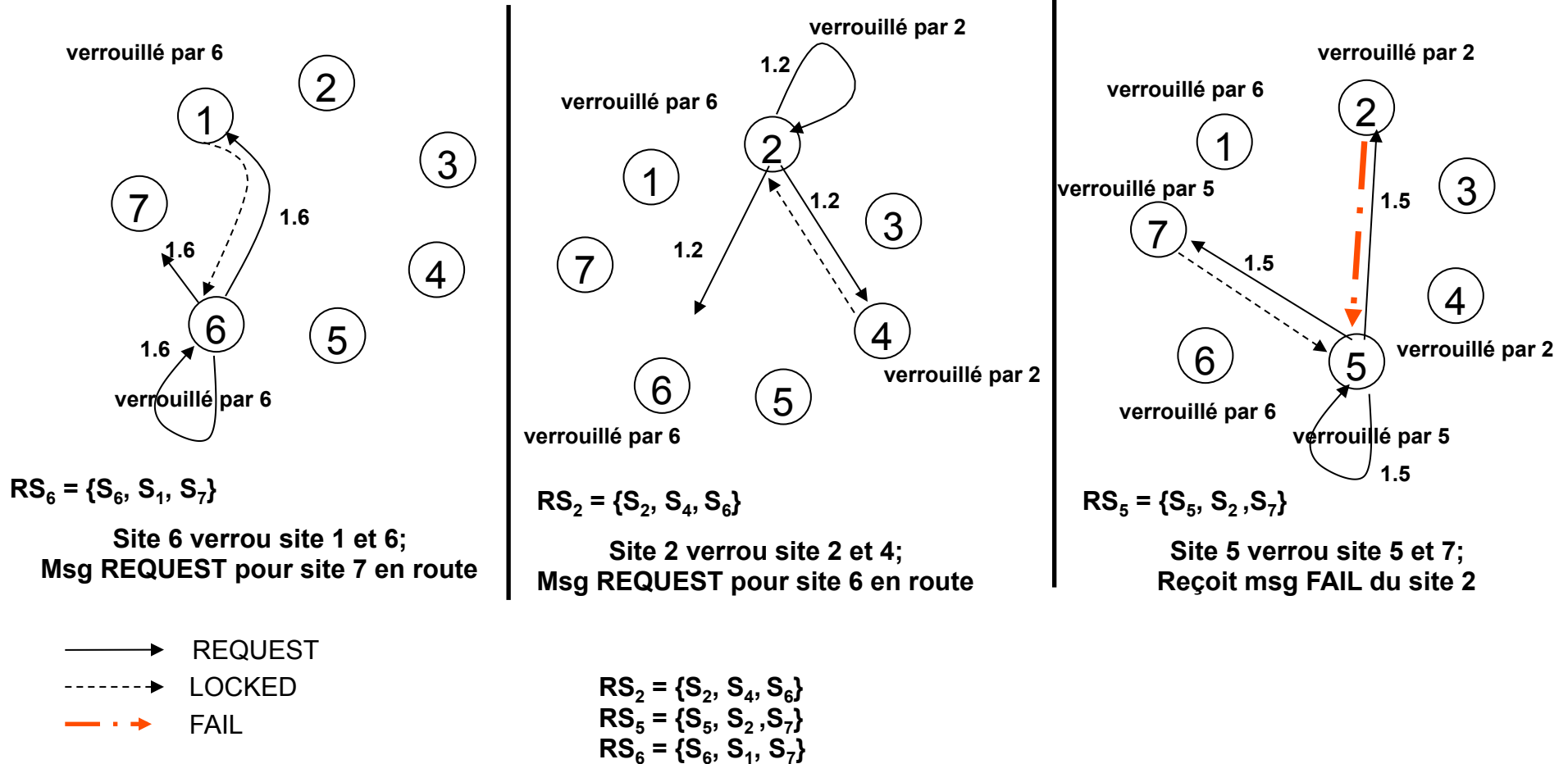
Sites 2, 5 et 6 exécutent Request\_CS  
 $H_2, H_5$  et  $H_6 = 1$

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

$N=7, K=3$

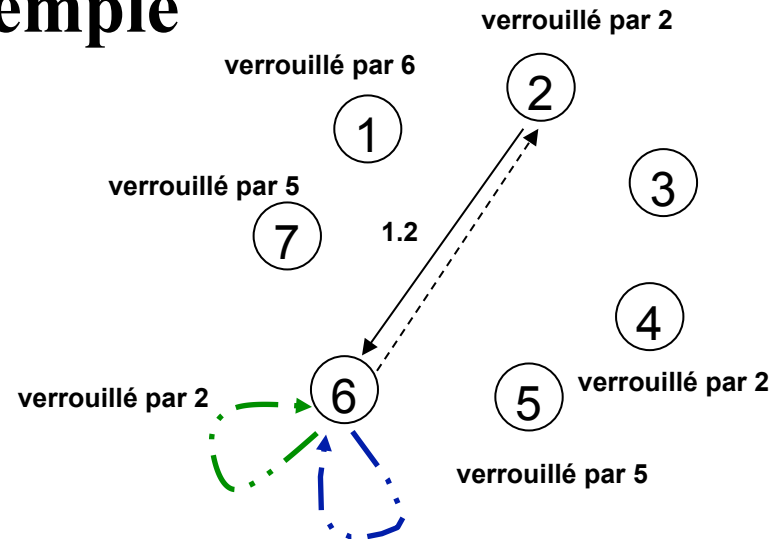
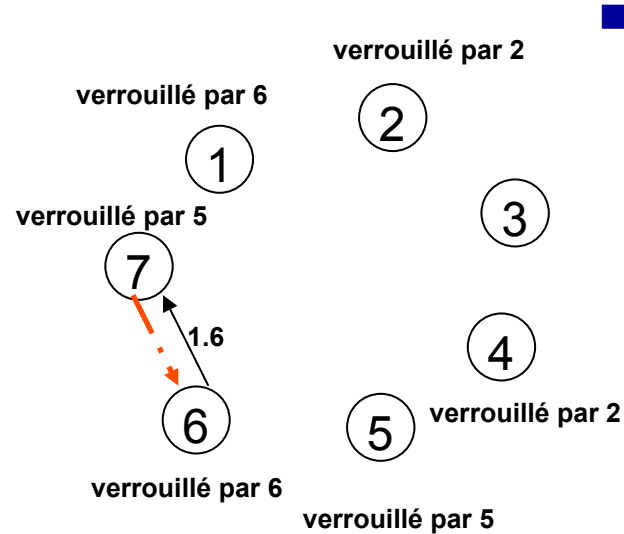
# Algorithme de Maekawa

## ■ Exemple



# Algorithme de Maekawa

## ■ Exemple



Site 7 reçoit msg REQUEST du site 6

- 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



# Algorithme de Maekawa

---

- **Nombre de Messages par exécution de SC :  $O(\sqrt{N})$  )**
  - Faible demande :  $3*(K-1)$ 
    - $(K-1) \text{ msg REQUEST} + (K-1) \text{ msg LOCKED} + (K-1) \text{ msg RELEASE}$
  - Forte demande :  $5*(K-1)$ 
    - $(K-1) \text{ msg REQUEST} + (K-1) \text{ msg LOCKED} + (K-1) \text{ msg RELEASE} + (K-1) \text{ *msg INQUIRE} + (K-1) \text{ *RELINQUISH}$
  - 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

# Algorithmes à base de Jeton

---

- **Anneau**
  - Martin
- **graphe complet**
  - Susuki/Kasami
- **Arbre**
  - Raymond (statique)
  - Naimi-Trehel (dynamique)

# Algorithme à base de jeton

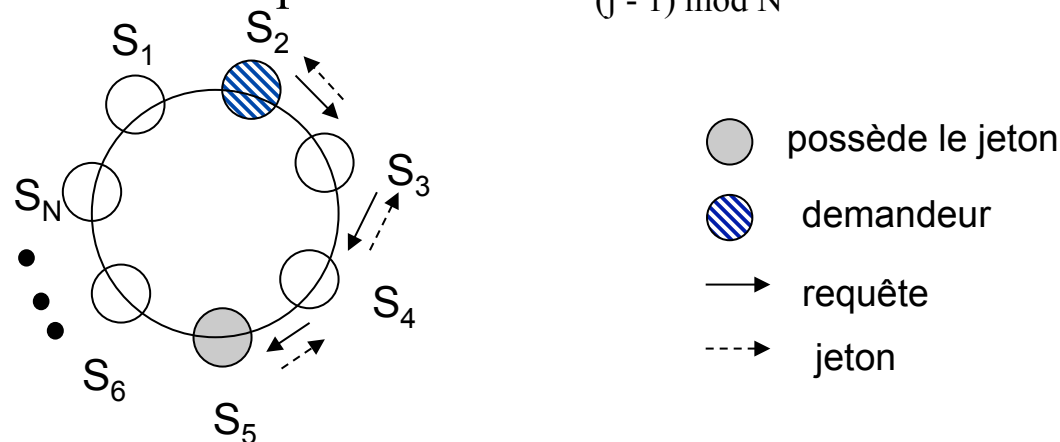
---

- **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).
    - **Envoie de requêtes**
      - Anneau : **Martin**
      - Arborescence: **Naimi/Trehel**
      - Diffusion : **Suzuki/Kasami**

# 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  $S_{(j-1) \bmod N}$ .



# Algorithme de Martin

## Evaluation

---

- **Nombre de Messages par exécution de SC :**
  - Si  $K$  = nombre de sites entre  $S_i$  (site qui demande la SC) et le site  $S_p$  (site qui possède le jeton), alors :
    - Nb messages =  $2 \cdot (K+1)$ ;
- **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.

# Algorithme de Raymonde

## arborescence statique

---

- **Les processus sont organisés en arbre ayant pour racine le site qui possède le jeton.**
  - Les arrêts sont orientés vers la racine
- **Les demandes du jeton**
  - sont propagées vers la racine
  - sont enregistrées dans une file locale sur chaque site du trajet

# Algorithme de Raymonde

## arborescence statique

---

- Un nœud ne communique qu'avec ses voisins
- Chaque nœud possède:
  - variable *holder* qui pointe en direction du nœud racine
- *file FIFO* pour sauvegarder les requêtes pendantes de ses voisins
- Arbre modifié (inversion du pointeur) à chaque transmission du jeton

# Algorithme de Raymonde

## arborescence statique

---

### ■ Algorithme

- Lorsqu' un nœud demande lui-même le jeton ou reçoit une requête pour le jeton de ses voisins, le noeud ajoute la requête dans sa file locale.
  - Si la file était vide il renvoie un requête à son *holder*
- En recevant une requête, le nœud qui possède le jeton le libère lorsqu' il ne l' utilise plus.
  - A chaque libération du jeton un nœud inverse la direction de *holder*



# Algorithme de Raymonde arborescence statique

---

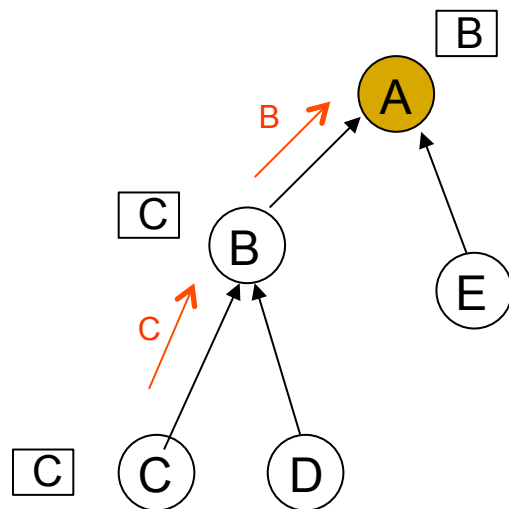
## ■ Algorithme (cont.)

- Lorsqu'un nœud reçoit le jeton, il enlève le premier élément *first* de sa file.
  - Si *first* est le propre nœud, il rentre en section critique
  - Sinon le jeton est renvoyé à *first*
- Si la file n'est pas vide, une requête pour le jeton est renvoyé au voisin.

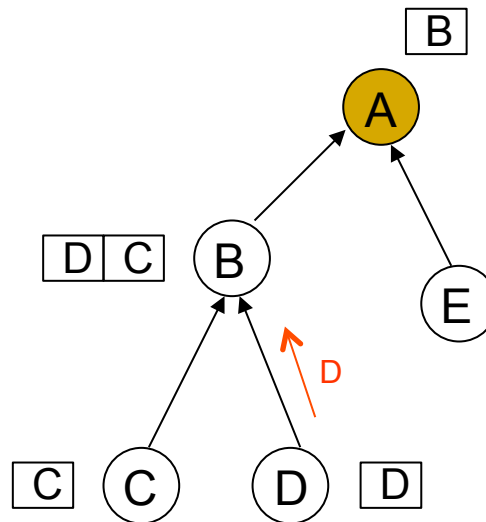
# Algorithme de Raymonde

## arborescence statique

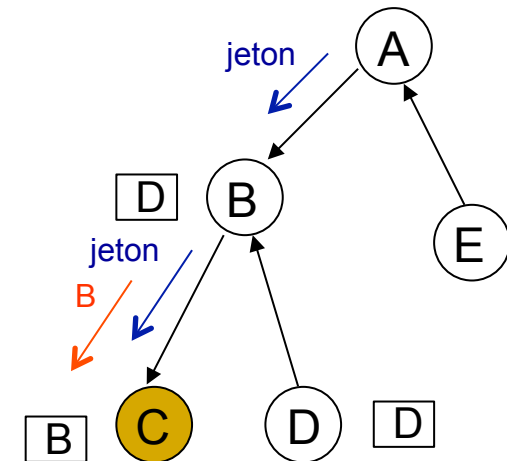
---



C demande le jeton



D demande le jeton



A libère le jeton

# Algorithme de Naimi/Trehel

## arborescence dynamique

---

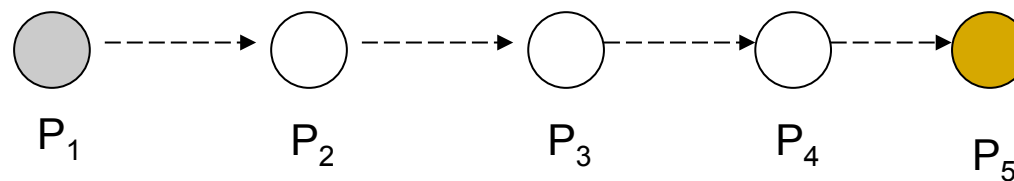
- **Deux structures de données:**
  - **File de requêtes : "*next*"**
  - **Arbre de chemins vers le dernier demandeur : "*father*"**

# 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.



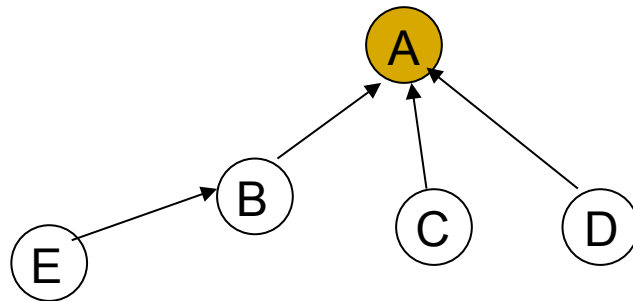
● possède le jeton

● dernier demandeur

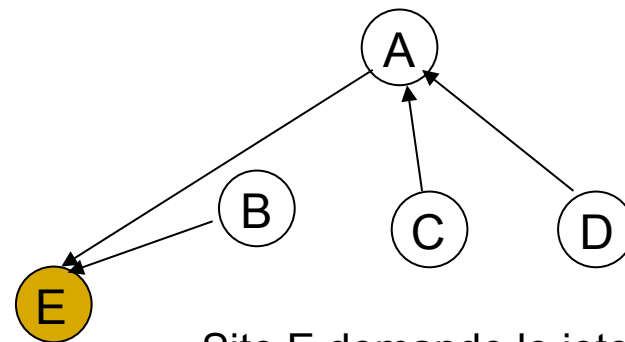
# Algorithme de Naimi/Trehel

## ■ 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.



Site A dernier demandeur



Site E demande le jeton

# Algorithme de Naimi/Trehel

---

## Local Variables:

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

## Initialisation de $S_i$ :

father =  $S_1$ ; 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,  $S_i$ ) to father;  
    father = nil;  
}  
attendre (Token == true);
```

## Release\_CS ( $S_i$ ):

```
requesting = false;  
if (next <> nil) {  
    send (Token) to next;  
    Token = false;  
    next = nil;  
}
```

# Algorithme de Naimi/Trehel (cont)

---

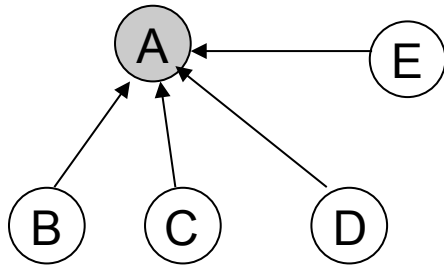
## **Receive\_Request\_CS( $S_j$ ):**

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

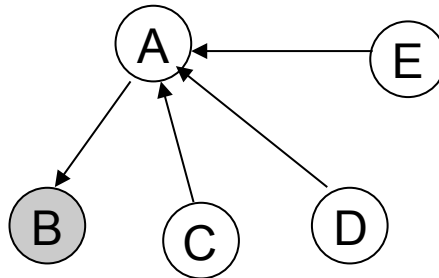
## **Receive\_Token ( $S_j$ ):**

```
Token = true;
```

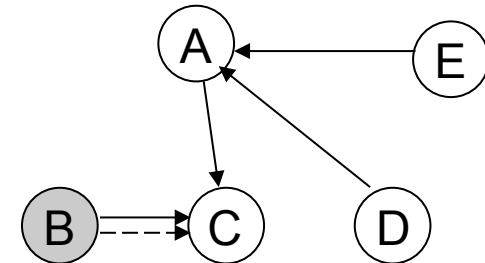
# Algorithme de Naimi/Trehel (Exemple)



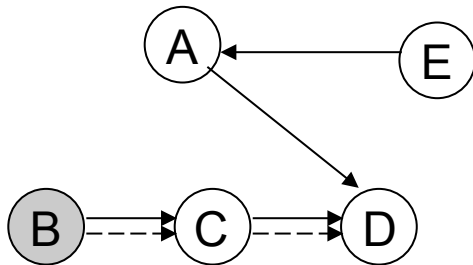
Site A possède le jeton



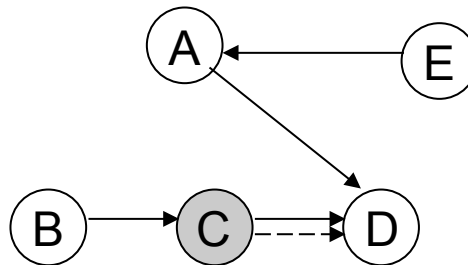
Site B fait une requête  
B entre en SC



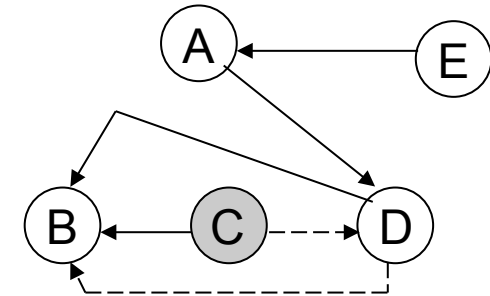
Site C fait une requête



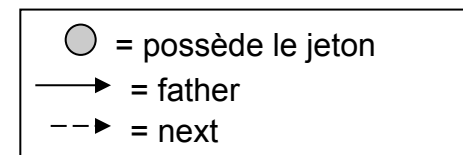
Site D fait une requête



Site B sort de la SC  
C entre en SC



Site B fait une requête





# Algorithmes de Raymonde et Naimi/Trehel (Evaluation)

---

- **Nombre de Messages par exécution de SC :**
  - Entre 0 et N par demande
  - Moyenne :  $O(\log N)$ .
  
- **Avantages :**
  - Extensibilité :  $O(\log N)$ .
  - Naimi-Tréhel
    - un site qui n'est pas intéressé par la section critique ne sera plus sollicité après quelques transferts de requêtes "adaptativité".

# 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.

# Algorithme de Susuki/Kasami

---

## ■ Type de Message:

### ➤ REQUEST ( $S_j, k$ ):

- $k = (1, 2, \dots, N)$ . 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.

# Algorithme de Susuki/Kasami

---

## ■ Variables:

- **Etat<sub>i</sub>** : *requesting, not\_requesting, critical\_section*.
- **Token<sub>i</sub>** : indique la présence du jeton sur le site  $S_i$ .
- **RN<sub>i</sub>**: vecteur de N positions :
  - RN<sub>i</sub> [j] est le numéro de la dernière requête reçue de la part du site  $S_j$ .
  - RN<sub>i</sub> [i] correspond au nombre de requêtes faites par le site  $S_i$ .
- **LN<sub>i</sub>**: vecteur de N positions des requêtes satisfaites

# Algorithme de Susuki/Kasami

---

## Initialisation variables locales ( $S_i$ ):

```
Token = ( $S_i == S_1$ );  
Etat = not_requesting;  
RN[j] = 0, j = 1, 2, ... N;  
LN[j] = 0, j = 1, 2, ... N;  
Q =  $\emptyset$ ;
```

## Request\_CS (i):

```
Etat = requesting;  
if (Token == false) {  
    RN[i] = RN[i] + 1;  
    diffuser REQUEST( $S_i$ , RN[i]);  
    attendre (Token == true)  
}  
Etat = critical_section;
```

## Release\_CS (i):

```
LN[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;  
}  
  
if (Q !=  $\emptyset$ ) {  
    Token = false;  
    site = extraire (Q); /*premier de la file  
    send TOKEN (Q, LN) to site ;  
}  
  
Etat = not_requesting;
```

# 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) to  $S_j$ ;

    }

**Recieve\_Token (TOKEN (Q, LN)):**

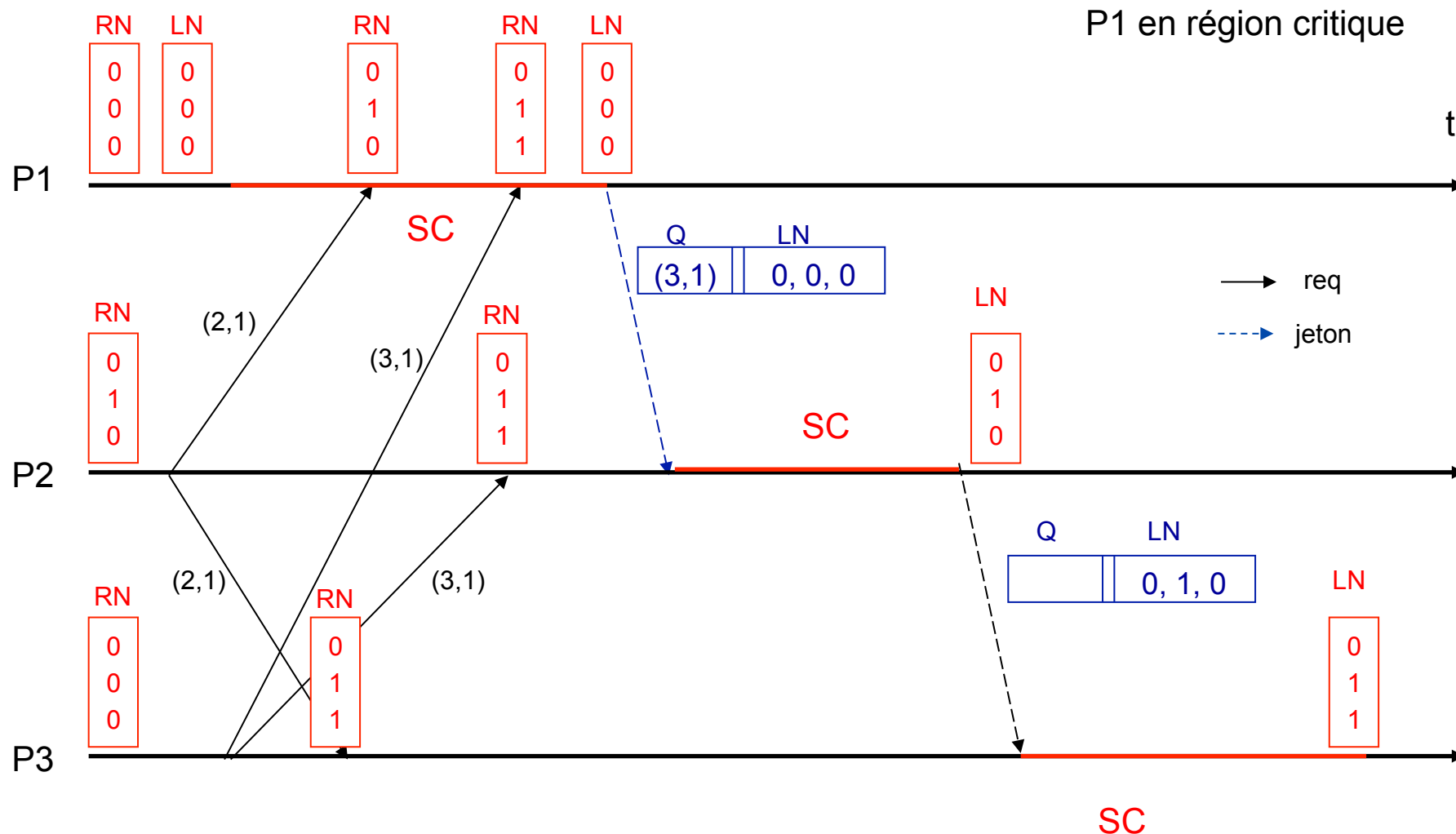
    Token = true;

    LN = TOKEN.LN ;

    Q = TOKEN.Q;

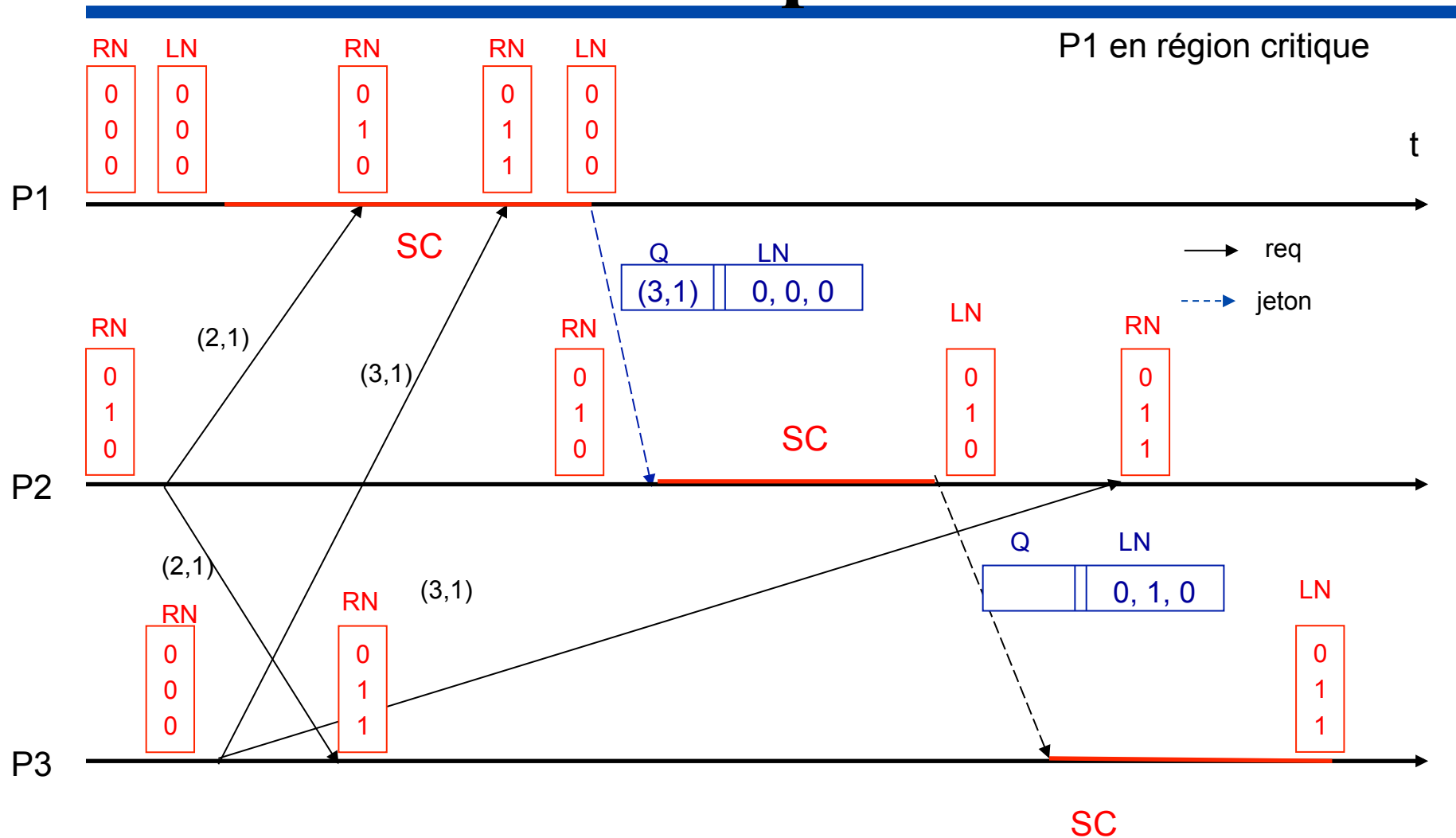
# Algorithme de Suzuki/Kasami

## Exemple 1



# Algorithme de Suzuki/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

# 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 Trehel, 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.
- K. Raymond 1989. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems* 7, 61–77.
- MARTIN, A.J. Distributed Mutual Exclusion on a Ring of Processes, *Science of Computer Programming*, vol5. pp-265-276, Feb. 1985.
- MAEKAWA, M. *A sqrt(n) algorithm for mutual exclusion in decentralized 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.