Systèmes Distribués
S6P

LICENCE 3 INFO

CM (16h): Pascal Mérindol - merindol@unistra.fr
TD/TP (10h+4h): Wafaa Ait-Cheik-Bihi - aitcheikbihi@unistra.fr

http://www-r2.u-strasbg.fr/~merindol/

http://robinet.u-strasbg.fr/enseignement/sd/sd

NETWORK RESEARCH TEAM

Plan du cours

I. Introduction aux Systèmes Distribués

II. Partage des données et Diffusion

III. Exclusion mutuelle & Inter-blocages

IV. Terminaison & Etat Global

V. Ordonnancement & Election

VI. Tolérance aux pannes & Sécurité

NETWORK RESEARCH TEAM

Terminaison

- Définition du problème
  - ✓ Soit un ensemble de tâches inter-dépendantes réparties sur un graphe de communication
    - ➡ arrêt des procs =?= fin de l'appli répartie ou état transitoire ?
      - ➡ est-ce qu'un msg en transit peut provoquer le réveil de l'appli ?
- Définitions & hypothèses
  - ✓ proc. *actif* si il exécute du code, *inactif* sinon
  - ✓ un proc. *inactif* n'envoie pas de msg
  - ✓ *inactif* → *actif* à la réception d'un msg, *actif* → *inactif* quand il veut
  - ✓ Initialement, ils sont tous actifs
  - ✓ Terminaison ≠ bonne exécution (satisfaction locale ou globale hors sujet)
  - ✓ réseau FIFO
  - ✓ un proc. qui ne reçoit plus de msg se termine en un tps fini

Anneau uni-directionnel

Principe

- ✓ Un jeton à 2 états (**`termine/transitoire`**) envoyé par le proc. testant la terminaison
  - ➡ s'il revient dans l'état initial, **`termine`**, alors l'appli est terminée
  - ➡ s'il revient dans l'état **`transitoire`**, alors il faut relancer le test de terminaison...
- ✓ <u>L'algorithme :</u>
  - ➡ chq proc P_i dispose de deux variables locales : $\texttt{couleur}_i$ (blanc/noir) et $\texttt{etat}_i$ (actif/inactif)
  - ➡ dès que P_i reçoit un msg alors $\texttt{etat}_i \leftarrow \texttt{actif}$
  - ➡ le jeton est transmis de P_i à P_{i+1} quand $\texttt{etat}_i\texttt{=inactif}$ à la fin d'un calcul (ou directement)
  - ➡ si pdt son calcul P_i envoie un msg à $P_j \mid j<i$, alors $\texttt{couleur}_i \leftarrow \texttt{noir}$
  - ➡ si $\texttt{couleur}_i\texttt{=noir}$, alors P_i transmet le jeton avec l'état **`transitoire`** au site suivant (et $\texttt{couleur}_i \leftarrow \texttt{blanc}$), sinon le jeton reste dans un état identique à celui de réception
  - ➡ quand le jeton revient en P_o, si son état=**`termine`**, alors terminaison validée !
    sinon on repart pour un tour !..

Anneau uni-directionnel

- Principe

  - ✓ Un jeton à 2 états (
    - Pourquoi le test si j < i ?
    - A quoi sert la variable couleur ?

    c. testant la terminaison
    - ➡ s'il revient dans ée
    - ➡ s'il revient dans l'état **`transitoire`**, alors il faut relancer le test de terminaison...
  - ✓ <u>L'algorithme :</u>
    - ➡ chq proc P_i dispose de deux variables locales : **`couleur`**$_i$ (blanc/noir) et **`etat`**$_i$ (actif/inactif)
    - ➡ dès que P_i reçoit un msg alors **`etat`**$_i$ ← **`actif`**
    - ➡ le jeton est transmis de P_i à P_{i+1} quand **`etat`**$_i$**`=inactif`** à la fin d'un calcul (ou directement)
    - ➡ si pdt son calcul P_i envoie un msg à P_j | j<i, alors **`couleur`**$_i$ ← **`noir`**
    - ➡ si **`couleur`**$_i$**`=noir`**, alors P_i transmet le jeton avec l'état **`transitoire`** au site suivant (et **`couleur`**$_i$ ← **`blanc`**), sinon le jeton reste dans un état identique à celui de réception
    - ➡ quand le jeton revient en P_o, si son état=**`termine`**, alors terminaison validée !
      sinon on repart pour un tour !..

Anneau uni-directionnel

Principe

- ✓ Un jeton à 2 états (**`termine/transitoire`**) envoyé par le proc. testant la terminaison
  - ➡ s'il revient dans l'état initial, **`termine`**, alors l'appli est terminée
  - ➡ s'il revient dans l'état **`transitoire`**, alors il faut relancer le test de terminaison...
- ✓ <u>L'algorithme :</u>
  - ➡ chq proc P_i dispose de deux variables locales : $\texttt{couleur}_i$ (blanc/noir) et $\texttt{etat}_i$ (actif/inactif)
  - ➡ dès que P_i reçoit un msg alors $\texttt{etat}_i \leftarrow \texttt{actif}$
  - ➡ le jeton est transmis de P_i à P_{i+1} quand $\texttt{etat}_i\texttt{=inactif}$ à la fin d'un calcul (ou directement)
  - ➡ si pdt son calcul P_i envoie un msg à $P_j \mid j<i$, alors $\texttt{couleur}_i \leftarrow \texttt{noir}$
  - ➡ si $\texttt{couleur}_i\texttt{=noir}$, alors P_i transmet le jeton avec l'état **`transitoire`** au site suivant (et $\texttt{couleur}_i \leftarrow \texttt{blanc}$), sinon le jeton reste dans un état identique à celui de réception
  - ➡ quand le jeton revient en P_o, si son état=**`termine`**, alors terminaison validée !
    sinon on repart pour un tour !..

Arbre couvrant

- Quelques notations
  - ✓ Soit P_i un sommet dans l'arborescence, posons $etat\_local(P_i) = \begin{cases} 1 \text{ si } P_i \text{ a fini son code} \\ 0 \text{ sinon} \end{cases}$
    - ➡ soit D_{Pi} l'ensemble des descendants de P_i à qui il a transmis une demande
    - ➡ soit $etat_i(P_j)$, la vision que P_i a de l'état de P_j
      - ➡ alors $etat_i(P_i) = \begin{cases} etat\_local(P_i) \text{ si } D_{Pi} = \varnothing \\ etat\_local(P_i) \times \prod etat_i(P_j), \forall P_j \in D_{Pi} \text{ sinon} \end{cases}$
- La procédure
  - ✓ En phase d'initialisation ($etat_0(P_0) = 0$) :
    - ➡ msg du père P_i vers un fils P_j :
      - ➡ sur P_i : $etat_i(P_j) \leftarrow 0$, $D_{Pi} \leftarrow D_{Pi} \cup P_j$, P_i attend l'état de P_j avant de recalculer le sien
      - ➡ sur P_j : $(etat\_local(P_j), etat_j(P_j)) \leftarrow (0,0)$, P_j devient actif et exécute son code
  - ✓ En mode détection :
    - ➡ dès que $etat_{j\neq0}(P_{j\neq0}) = 1$, P_j transmet son état à son père
    - ➡ si $etat_0(P_0) = 1$ => l'appli est finie !

Graphe générique

Principe et définition

- ✓ Soit un un graphe de communication connexe => il existe un circuit C sur ce graphe
  - ➡ on «lance» un jeton sur C, si le nb de proc visité et inactif = taille(C) => terminaison !
- ✓ Soit *succ(P_i)* le successeur de P_i dans le circuit C
- ✓ Chaque proc P_i dispose de quatre variables locales : **`couleur`$_i$`, etat`$_i$`, jeton_in`$_i$`, nb`$_i$**

Procédure

- ✓ Si P_i reçoit msg : **`etat`$_i$ ← `actif`** / Si P_i envoie msg : **`couleur`$_i$ ← `noir`** / A la fin d'un calcul : **`etat`$_i$ ← `inactif`**
- ✓ A la réception d'un jeton, **`jeton_in`$_i$ ← `vrai, nb`$_i$ `= nb`** (valeur contenu dans le jeton)
  - ✓ si **`etat`$_i$ `== actif`** : soit il détecte la TERMINAISON, soit il transmet le jeton au successeur
  - ✓ sinon (**`etat`$_i$ `!= actif`**) attendre fin du calcul local et transmission jeton
- ✓ Pdt une transmission de jeton avec **`nb = nb`$_i$** vers *succ(P_i)*:
  - ✓ si **`couleur`$_i$ `== noir`** : **`nb`$_i$ `= 1`** /* on recommence la détection */ et **`couleur`$_i$ ← `blanc`**
  - ✓ sinon : **`nb`$_i$ `= nb + 1`**
- ✓ Détection de la TERMINAISON : **`nb == taille(C)`** et **`couleur`$_i$ `== blanc`**

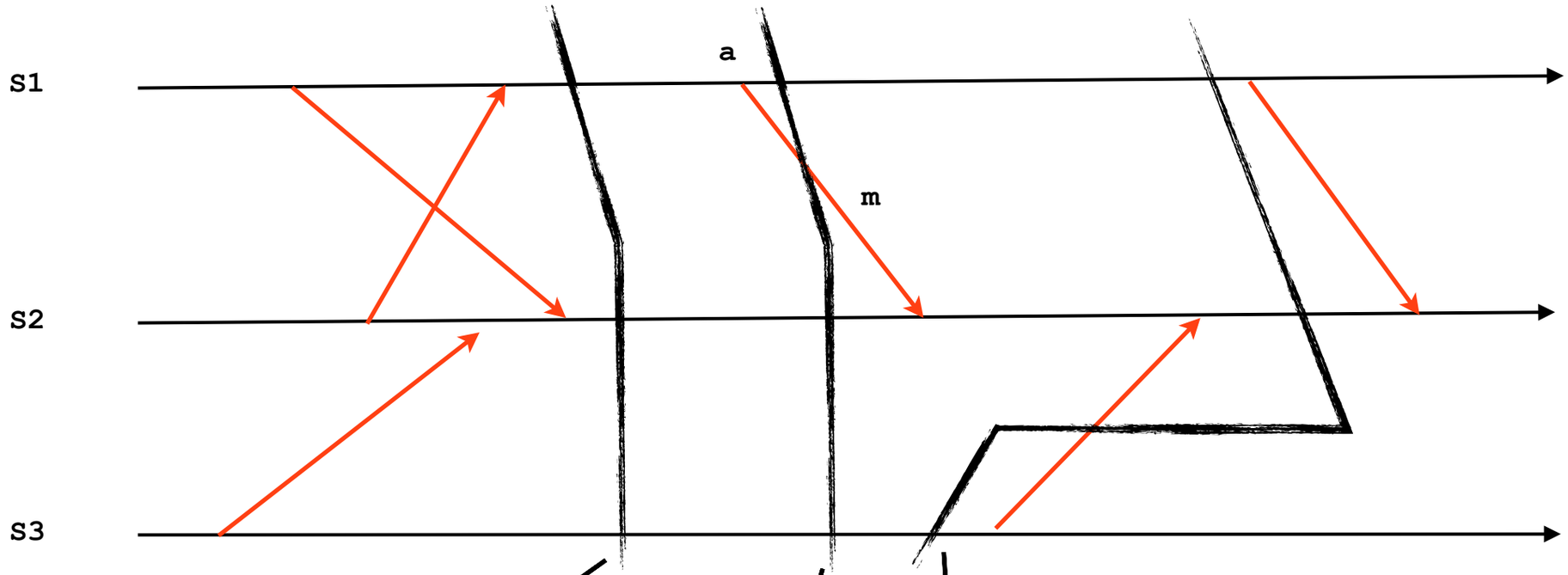
Etat global

- Pourquoi «observer» un système distribué ?
  - ✓ Capture de caractéristiques globales (terminaison, deadlocks, performances, ...)
  - ✓ Snapshot pour reprise sur panne
  - ➡ mais IMPOSSIBLE d'avoir une prise de vue simultanée des différents sites ...

Définition état global

- ✓ Chaque proc P_i et chaque canal $C_{i \to j}$ possède, à tout instant, un état dit local
- ✓ soit $el_i(k)$ *et* $ec_{i \to j}(k')$, respectivement l'état local d'un proc. et d'un canal
- ✓ Trois types d'evts :
  - ➡ evt interne à un site : transition $el_i(k)$ à $el_i(k+1)$
  - ➡ émission d'un msg m par P_i sur le canal $C_{i \to j}$ (noté $emission_i(m)$): provoque une transition interne et $ec_{i \to j}(k'+1) = ec_{i \to j}(k') \cup m$
  - ➡ réception d'un msg m sur P_i via canal $C_{j \to i}$ (noté $reception_i(m)$): provoque une transition interne et $ec_{j \to i}(k'+1) = ec_{j \to i}(k') \setminus m$
- ✓ L'état global d'un système se note $S = \{\cup_i el_i, \cup_{(i,j)} ec_{i \to j}\}$
- ✓ Un état global est **cohérent** si
  - ➡ **si l'evt $emission_i(m)$ est capté dans el_i alors l'evt $reception_i(m)$ est capté par el_j ou $m \in ec_{i \to j}$**
  - ➡ **si l'evt $emission_i(m)$ n'est pas capté dans el_i alors l'evt $reception_i(m)$ n'est pas capté par el_j**

Exemple de cohérence



Coupe cohérente

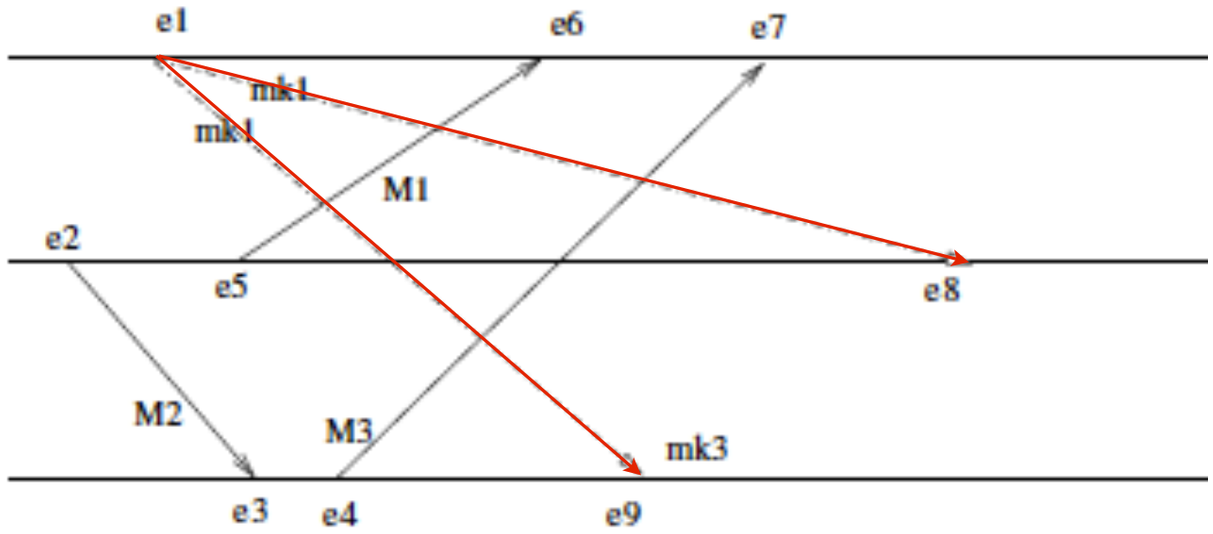
Coupe cohérente si on ajoute le msg m dans l'état : $S = \{\{a\}, \{ec_{1\rightarrow 2} = \{m\}\}\}$

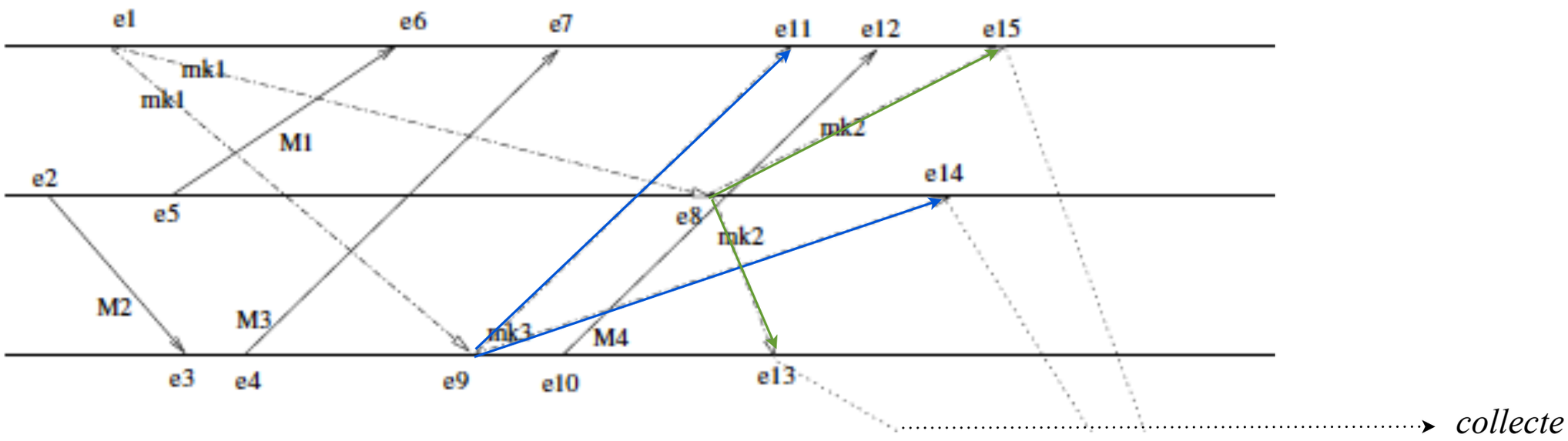
Coupe incohérente : la seconde condition est violée (la réception est prise compte alors que l'émission non!)

Solution avec canaux FIFO

- *Chandy & Lamport*

  - ✓ en FIFO, on a : $m_1 < m_2 \Leftrightarrow emission(m_1) < emission(m_2) \Leftrightarrow reception(m_1) < reception(m_2)$
  - ✓ msg de contrôle
    - ➡ assurance de cohérence



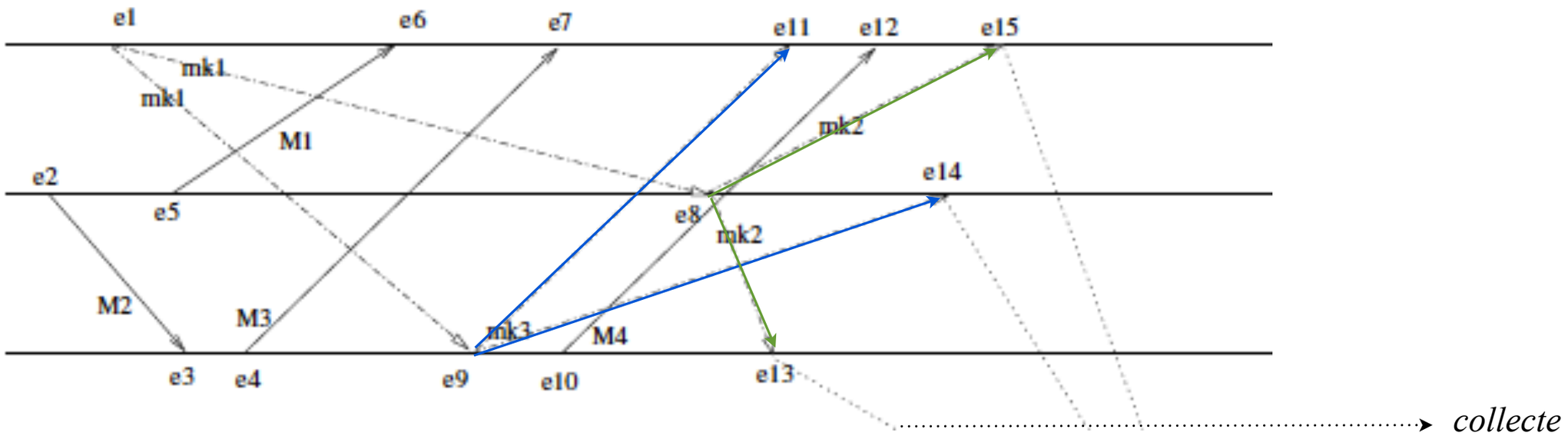


Algo *Chandy & Lamport*

- ✓ Soit P_i un proc, $\mathbf{EM_i}$ un booléen local (faux à l'init), P_c le site collecteur et N le nombre de procs.
- ✓ Chq proc dispose de deux tableaux :
  - ✓ $\mathbf{R_i[N]}$: l'ensemble de msgs $C_{j \rightarrow i}$ reçus entre le début du contrôle et sa fin locale, initialement vide
  - ✓ $\mathbf{EL_i[N]}$: l'ensemble des booléens $\mathbf{EL_i[j]}$ signifiant P_j a répondu au contrôle, initialement faux partout

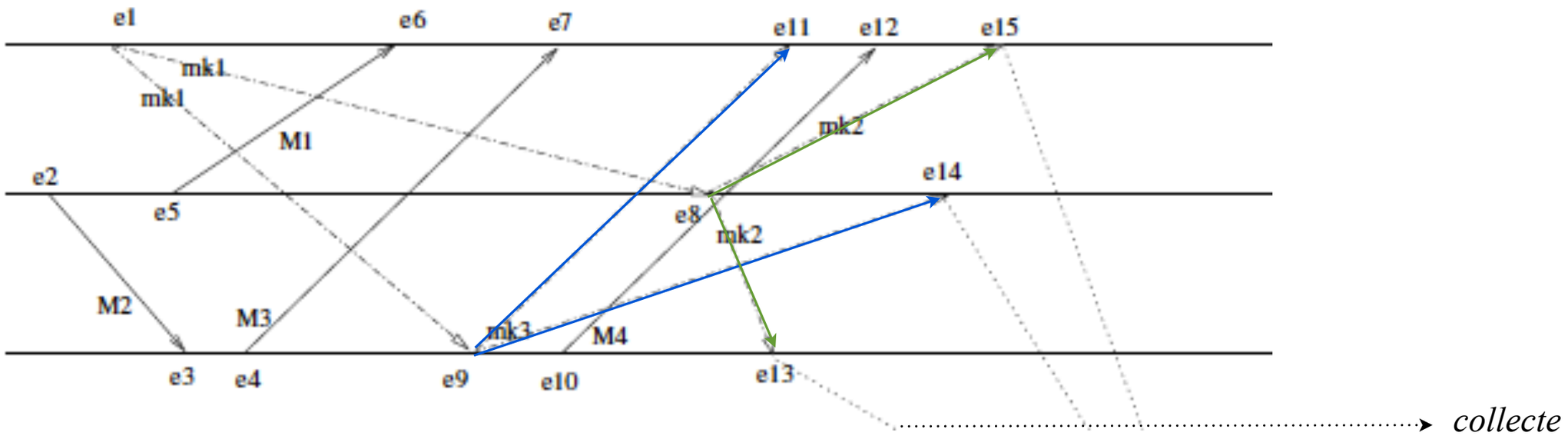
```
//Si Pi décide de prendre en compte son état local (par lui même ou via un msg de controle)
Launch_control(){
eli ← etat_local_courant;
EMi ← TRUE;
for all i≠j {
Ri[j] ← ∅;
envoyer msg de contrôle vers Pj;
}}

//A réception d'un msg m issue de Pj (pas de contrôle) sur Pi
if (EMi && !ELi[j]) Ri[j] ← m;

//A réception d'un msg de contrôle issue de Pj sur Pi
if (!EMi) Launch_control();
ELi[j] ← TRUE;
if (for all j≠i ELi[j] == TRUE)
envoyer l'état {eli, Ri} à Pc
```

Ex : *Chandy & Lamport*




```
//Si Pi décide de prendre en compte son état local (par lui même ou via un msg de controle)
Launch_control(){
eli ← etat_local_courant;
EMi ← TRUE;
for all i≠j {
Ri[j] ← ∅;
envoyer msg de contrôle vers Pj;
            }
                }

//A réception d'un msg m issue de Pj (pas de contrôle) sur Pi
if (EMi && !ELi[j]) Ri[j] ← m;

//A réception d'un msg de contrôle issue de Pj sur Pi
if (!EMi) Launch_control();
ELi[j] ← TRUE;
if (for all j≠i ELi[j] == TRUE)
envoyer l'état {eli, Ri} à Pc
```

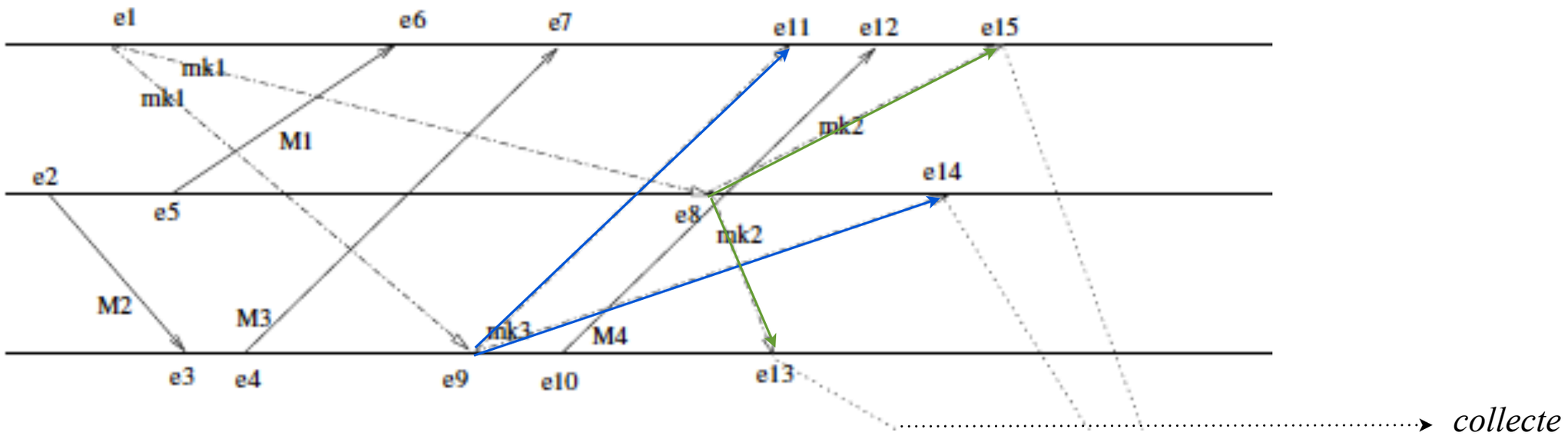
Ex : *Chandy & Lamport*




```
//Si Pi décide de prendre en compte son état local (par lui même ou via un msg de controle)
Launch_control(){
eli ← etat_local_courant;
EMi ← TRUE;
for all i≠j {
Ri[j] ← ∅;
envoyer msg de contrôle vers Pj;
            }
                }

//A réception d'un msg m issue de Pj (pas de contrôle) sur Pi
if (EMi && !ELi[j]) Ri[j] ← m;

//A réception d'un msg de contrôle issue de Pj sur Pi
if (!EMi) Launch_control();
ELi[j] ← TRUE;
if (for all j≠i ELi[j] == TRUE)
envoyer l'état {eli, Ri} à Pc
```

▶ A vous de jouer !

Ex : *Chandy & Lamport*



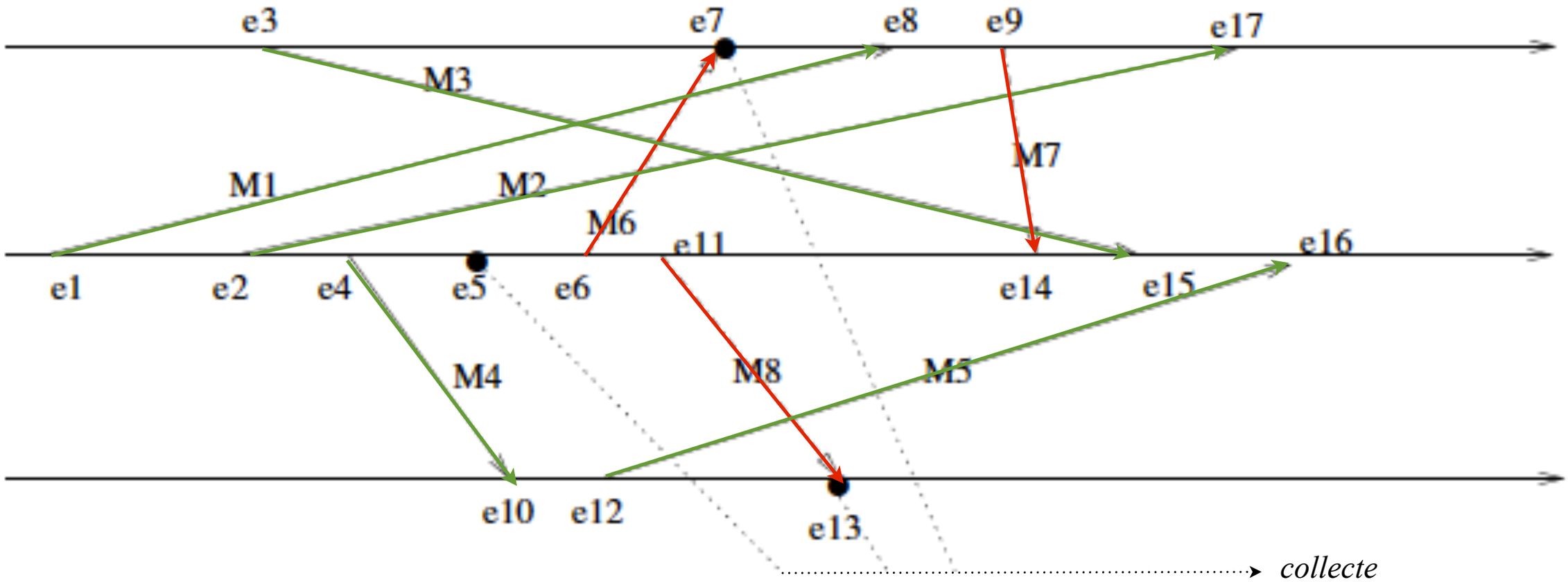

```
//Si Pi décide de prendre en compte son état local (par lui même ou via un msg de controle)
Launch_control(){
eli ← etat_local_courant;
EMi ← TRUE;
for all i≠j {
Ri[j] ← ∅;
envoyer msg de contrôle vers Pj;
            }
                }

//A réception d'un msg m issue de Pj (pas de contrôle) sur Pi
if (EMi && !ELi[j]) Ri[j] ← m;

//A réception d'un msg de contrôle issue de Pj sur Pi
if (!EMi) Launch_control();
ELi[j] ← TRUE;
if (for all j≠i ELi[j] == TRUE)
envoyer l'état {eli, Ri} à Pc
```

Canaux non FIFO

- Plusieurs types de solutions
  - ✓ Sans msg de contrôle
    - ➡ cumulatif et rapide ... *Lai & Yang*
    - ➡ non cumulatif et lent ... *Mattern*
  - ✓ Avec msg de contrôle
    - ➡ ... *Ahuja*
- Solutions basées sur l'ordre causal
  - ✓ centralisée
    - ➡ ... *Acharya & Badrika*
  - ✓ répartie
    - ➡ ... *Alagar & Venkatesan*

Solution cumulative *Lai & Yang*

- Dans le doute, on stocke tout...
  - ✓ ***vert*** avant enregistrement, ***rouge*** après



➡L'état $ec_{i \to j}$ d'un canal est la différence entre l'ensemble des messages verts émis par P_i vers P_j (noté **`msg_emis`**$_{\texttt{i}}$**`[j]`**) et l'ensemble des messages verts reçus par P_j depuis P_i (noté **`msg_recu`**$_{\texttt{j}}$ **`[i]`**)

Solution cumulative *Lai & Yang*

- 5 règles à vérifier

1. Un processus qui n'a pas encore enregistré son état est en mode vert : tous les messages qu'il émet sont verts.
2. Lorsqu'un processus enregistre son état local, il passe en mode rouge : tous les messages qu'il émet alors sont rouges.
3. Un processus doit enregistrer son état local (si ce n'est pas déjà fait) lorsqu'il reçoit un message rouge, et ce sans le prendre en compte.
4. Les messages verts émis et reçus sont stockés par le processus (aspect cumulatif).
5. Un processus rouge transmet au collecteur son état local et ses messages verts stockés

- Les deux conditions de cohérence sont satisfaites

✓ si l'émission de *m* est captée dans el_i , alors *m* est vert (R1) et m ∈ **`msg_emis`**$_i$**`[j]`** (R4)

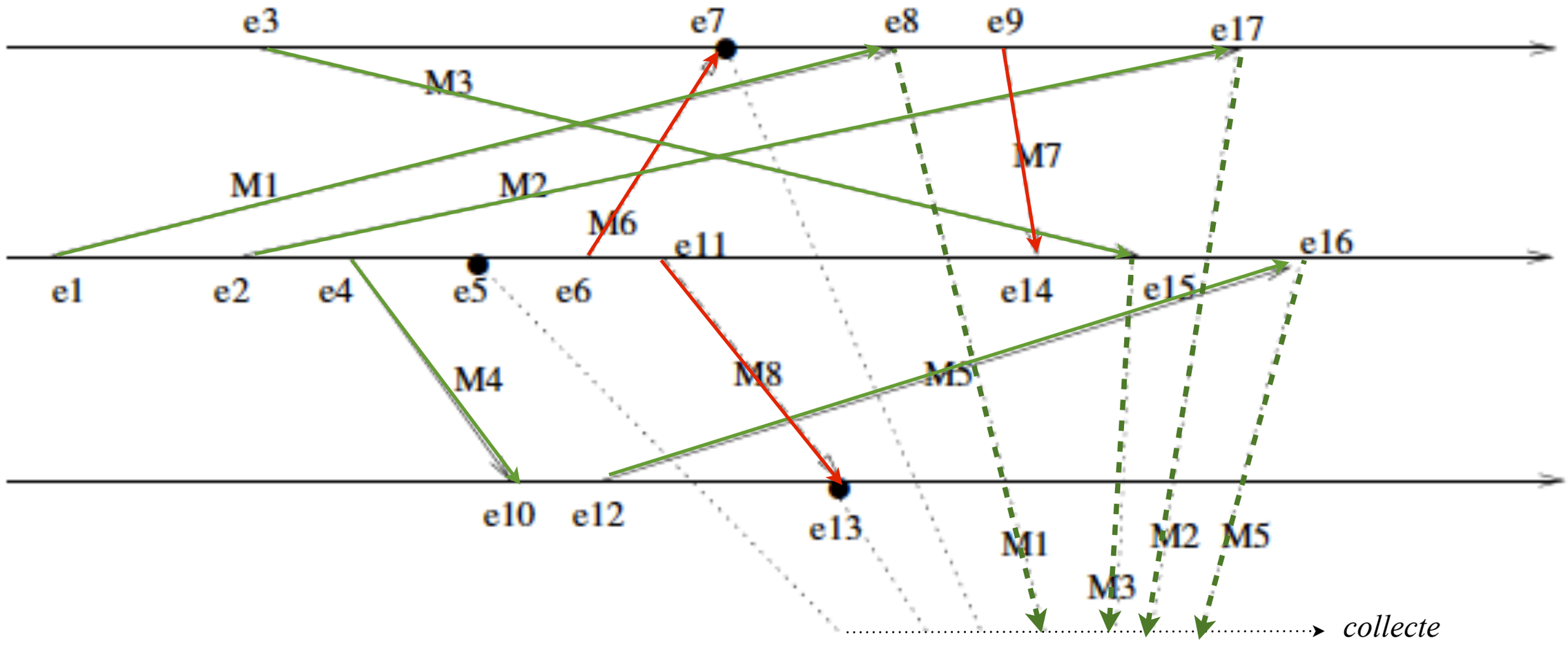
- ➡ si sa réception est captée dans el_j alors m ∈ **`msg_recu`**$_j$**`[i]`** (R4) => $m \notin ec_{i \to j}$
- ➡ si sa réception n'est pas captée dans el_j alors m ∉ **`msg_recu`**$_j$**`[i]`** (R4) => $m \in ec_{i \to j}$

✓ si l'émission de *m* n'est pas captée dans el_i : *m* est rouge (R2).

- ➡ lorsque *m* est reçu par P_j => la réception de *m* n'est pas captée dans el_j (R3)

Non cumulative *Mattern*

- On «récupère les messages verts qui trainent»...
  - ✓ utilisation d'un compteur de messages sur le site collecteur



➡ Chaque proc. P_i envoie une différence d_i de msg verts émis et reçus (en + de son état local)
➡ *mt :* nb de messages en transit (émis mais non reçus) = $\sum d_i$

Non cumulative *Mattern*

- A nouveau, 5 règles à vérifier

  ✓ Les trois premières sont identiques à la méthode cumulative...

  4. Chaque processus devenant rouge transmet au collecteur son état local et son différentiel ***d_i = nb. msg verts émis - nb. msg verts reçus*** (jusqu'ici !)

  5. Un processus rouge transmet au collecteur tous les messages verts qu'il reçoit

- Procédures algorithmiques sur le collecteur

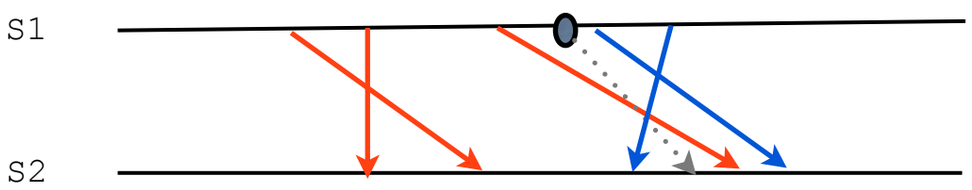
  ✓ A réception d'un msg de type
  - ➡ enregistrement *(el_i , d_i)* : ***N = N - 1*** **(où N = $|P_i|$);** ***mt += d_i;***
  - ➡ message vert (en retard) en provenance de P_j (M, site emetteur= P_i, site recepteur=P_j) :

    $C_{i\to j} = C_{i\to j} \cup M$; mt--;

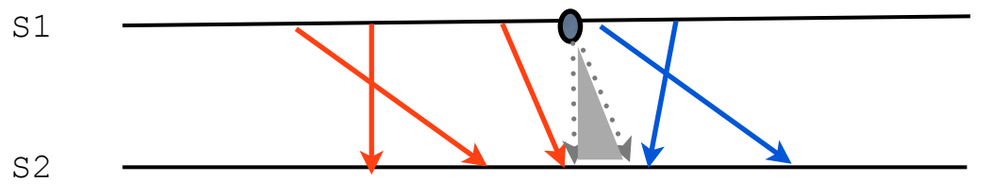
  ✓ Dès lors que *N == 0 && mt == 0* , le collecteur a reçu tous les états locaux et tous les messages en transit => il dispose de l'état global !

Avec msg de contrôle *Ahuja*

- Adaptation de l'algo de Chandy & Lamport
  - ✓ ajout de contraintes pour différencier les msgs émis AVANT / APRES le msg contrôle
  - ✓ sans modifier l'ordre de réception des msgs (sans perturber le phénomène mesuré)

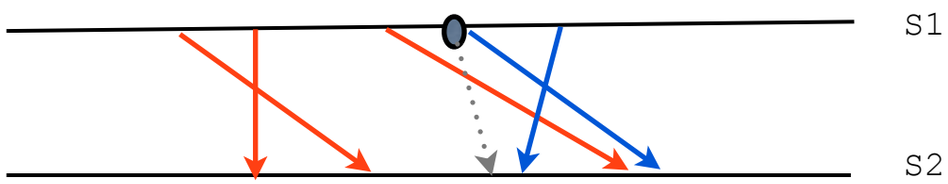


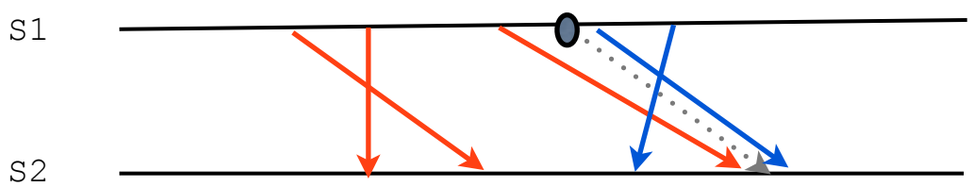
➡ canaux non FIFO



➡ **`two_ways_flush`** : perturbation des msgs :(

➡ **`backward_flush`** : tout message émis APRES arrivera APRES (je ne dépasse personne!)



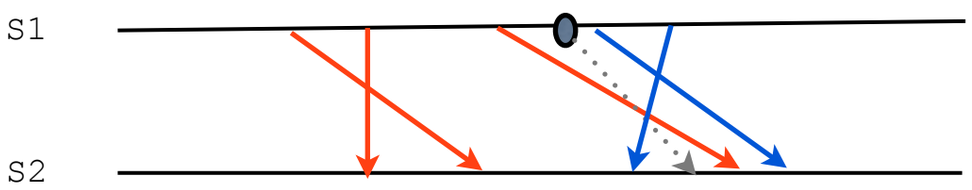


➡ **`forward_flush`** : tout message emis AVANT arrivera AVANT (personne ne me dépasse !)

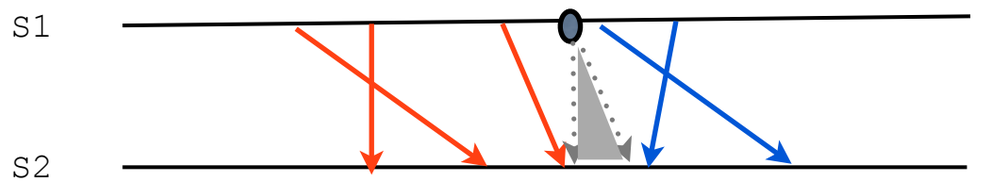
Avec msg de contrôle *Ahuja*

- Adaptation de l'algo de Chandy & Lamport
  - ✓ ajout
    - Comment mettre en pratique de tels mécanismes ?

    msg contrôle
  - ✓ sans modifier l'ordre de réception des msgs (sans perturber le phénomène mesuré)

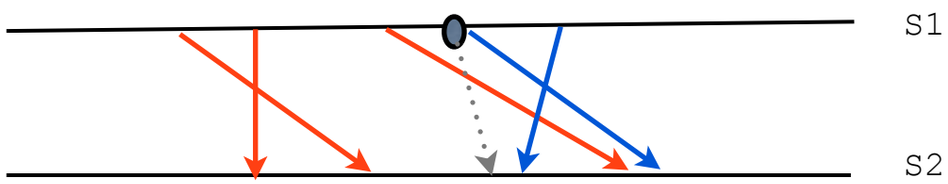


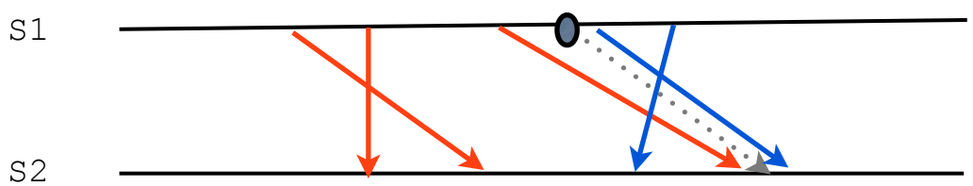
➡ canaux non FIFO



➡ **two_ways_flush** : perturbation des msgs :(

➡ **backward_flush** : tout message émis APRES arrivera APRES (je ne dépasse personne!)

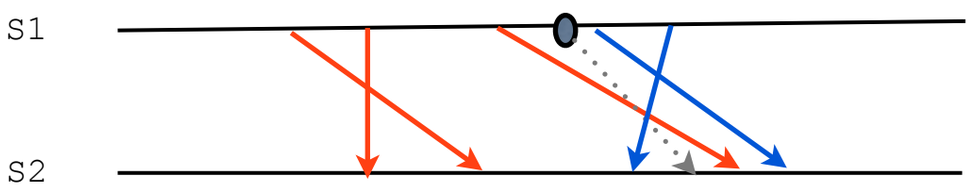




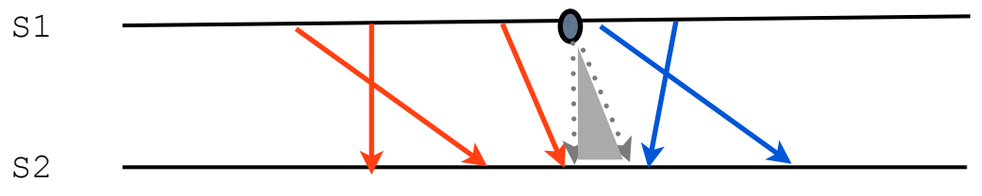
➡ **forward_flush** : tout message emis AVANT arrivera AVANT (personne ne me dépasse !)

Avec msg de contrôle *Ahuja*

- Adaptation de l'algo de Chandy & Lamport
  - ✓ ajout de contraintes pour différencier les msgs émis AVANT / APRES le msg contrôle
  - ✓ sans modifier l'ordre de réception des msgs (sans perturber le phénomène mesuré)

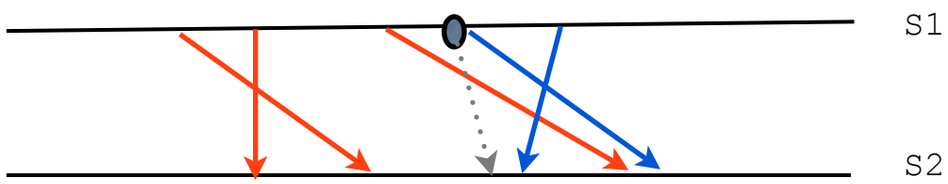


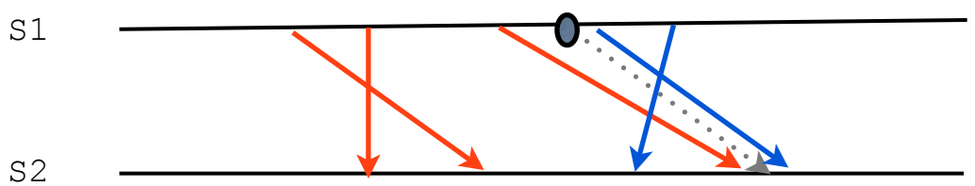
➡ canaux non FIFO



➡ **`two_ways_flush`** : perturbation des msgs :(

➡ **`backward_flush`** : tout message émis APRES arrivera APRES (je ne dépasse personne!)



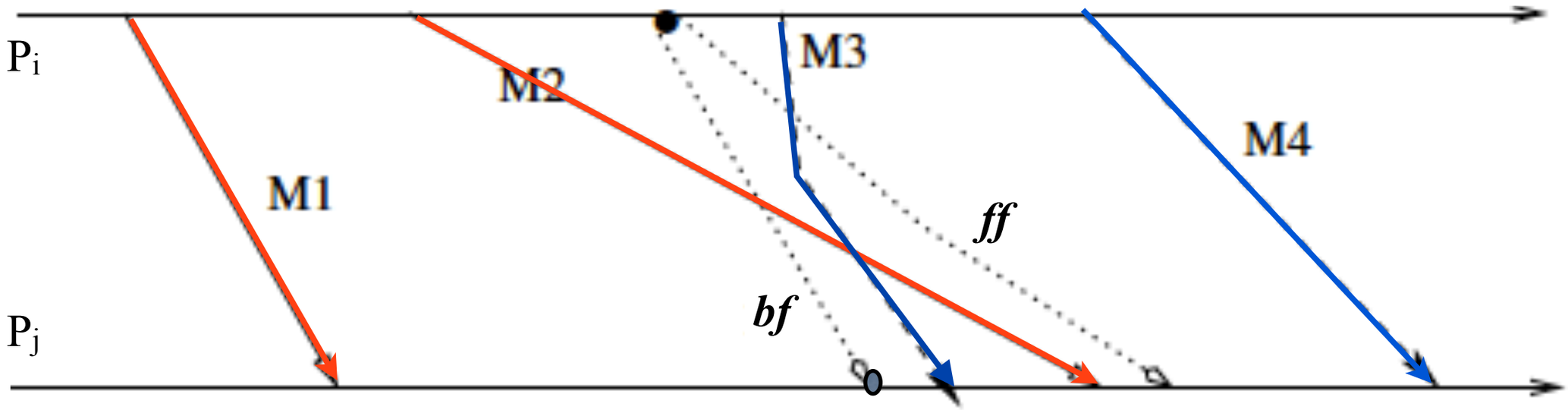


➡ **`forward_flush`** : tout message emis AVANT arrivera AVANT (personne ne me dépasse !)

Avec msg de contrôle *Ahuja*

- Comment ? Chaque P_i enregistre son état local el_i et l'état de ses canaux $C_{i\rightarrow j}$ en respectant les règles suivantes

1. P_i lorsqu'il enregistre son état local, envoie sur TOUS ses canaux de sortie, un message ***bf*** du type **`backward_flush`** avant d'envoyer tout autre message
2. à la réception d'un message ***bf***, le processus P_j enregistre son état selon la règle R1 (s'il ne l'a pas déjà fait)
3. P_i lorsqu'il enregistre son état local envoie sur CHACUN de ses canaux sortants, après le message ***bf*** (et ce avant d'envoyer tout autre message), un message ***ff*** du type **`forward_flush`** avec le même numéro que le dernier message émis sur ce canal (les messages ordinaires sont numérotés par canal - le message estampillé t est le $t^{ième}$ message envoyé sur ce canal)
4. à réception d'un message ***ff*** numéroté t, sur un canal $C_{i\rightarrow j}$, P_j enregistre $ec_{i\rightarrow j}$ comme étant l'ensemble des messages de numéro inférieur ou égal à t reçus après l'enregistrement de el_j



Avec msg de contrôle *Ahuja*

Information Processing Letters 34 (1990) 5–12
North-Holland

22 February 1990

***Flush* PRIMITIVES FOR ASYNCHRONOUS DISTRIBUTED SYSTEMS**

Mohan AHUJA

Department of Computer and Information Science, The Ohio State University, 2036 Neil Avenue, Columbus, OH 43210, USA

- Preuve

✓ R1 et R2 assure que si une réception est captée alors l'émission associée aussi
si un message *m* traversant $C_{i\to j}$ est tel que $emission_i(m)$ n'est pas captée dans el_i, alors *m* est émis APRÈS le message ***bf*** sur $C_{i\to j}$ => *m* est reçu APRÈS ce message
=> P_j aura déjà capté son état (dès la réception de ***bf***) et donc $reception_j(m)$ ne sera pas captée dans el_j ni dans $ec_{i\to j}$ (son estampille est > à celle de ***ff***)

✓ R3 et R4 assure que si une émission est captée alors sa réception ou le msg en transit est captée
si un message *m* traversant $C_{i\to j}$ est émis AVANT l'enregistrement de el_i ($emission_i(m)$ est donc captée dans celui-ci) : son estampille est nécessairement ≤ à celle de ***ff***
➡ soit *m* arrive avant ***bf*** : pas de problème, $reception_j(m)$ sera captée directement dans el_j
➡ soit *m* arrive après ***bf*** : alors il arrive nécessairement AVANT ***ff***
=> *m* est stocké dans l'état du canal !

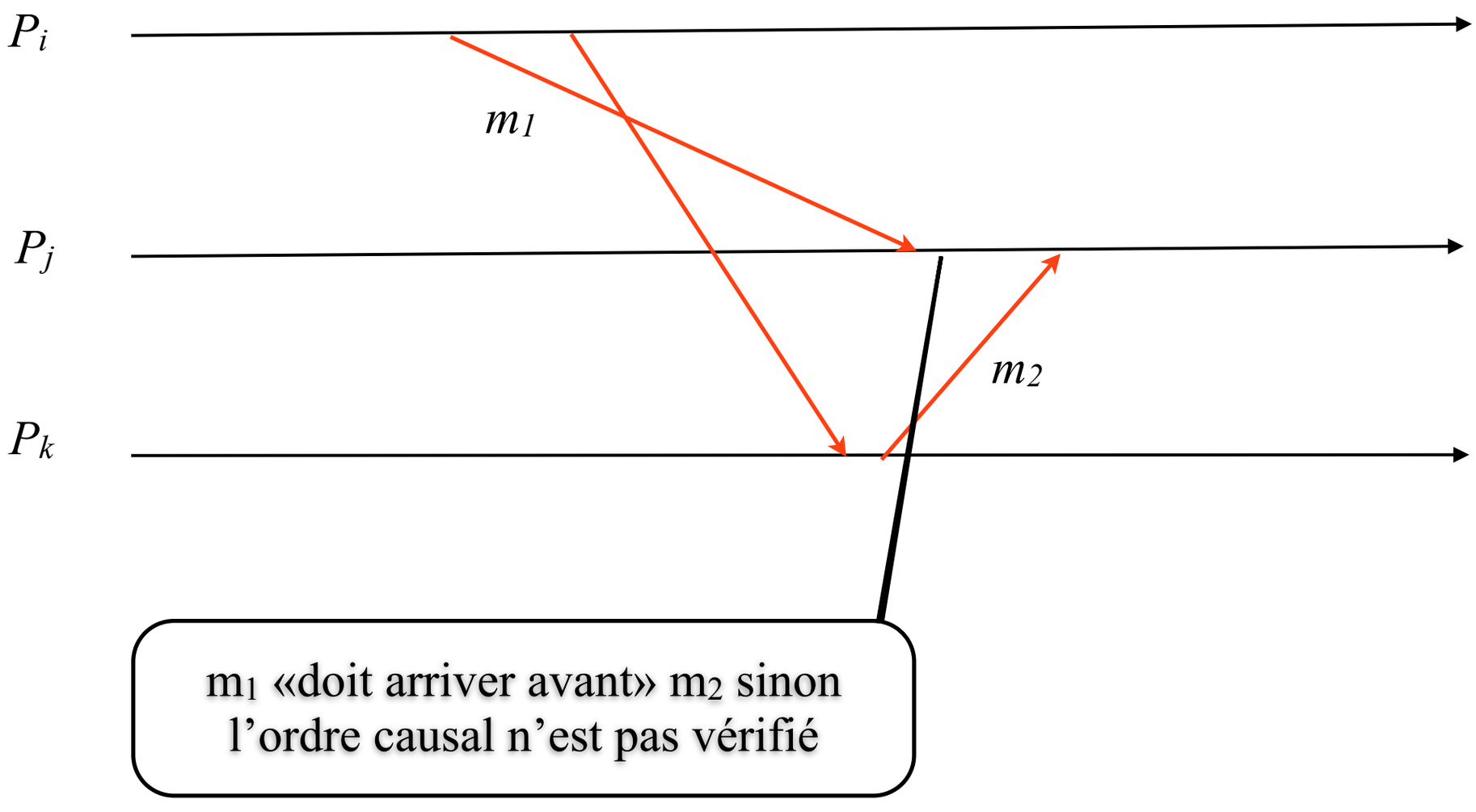
Rappel : ordre causal

- Ordre causal ≠ précédence causal (horloge vectorielle)

✓ $\forall P_i, P_j, P_k \; \forall m_1$ *émis sur* $C_{i \to j}$, $\forall m_2$ *émis sur* $C_{k \to j}$:

$emission_i(m_1) \rightarrow emission_k(m_2) \Rightarrow reception_j(m_1) \rightarrow reception_j(m_2)$

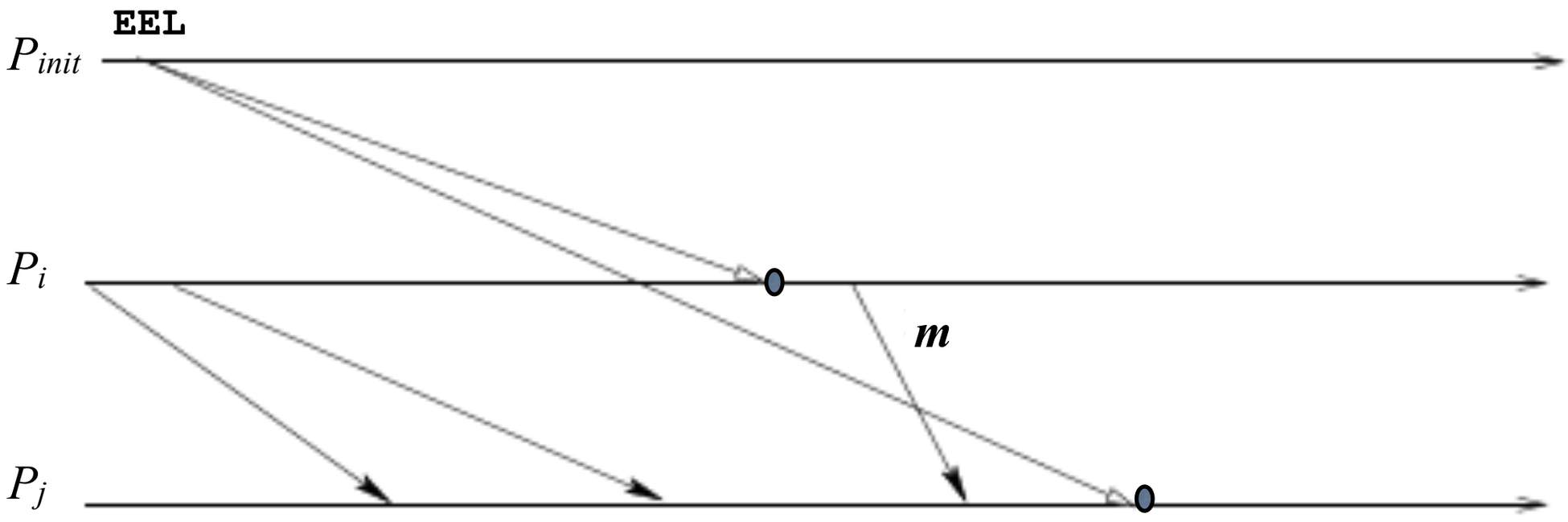
✓ Ordre causal => FIFO mais la réciproque n'est pas vrai...



Ordre causal, centralisée *Acharya & Badrika*

- Calcul des états locaux

1. L'initiateur P_{init} diffuse (y compris à lui même) un message de contrôle **`EEL`**
2. Lorsqu'un processus P_i reçoit ce message de contrôle, il enregistre son état local el_i



Considérons un message *m* envoyé sur le canal $C_{i \to j}$ tel que $emission_i(m)$ n'est pas capté dans el_i mais $reception_j(m)$ captée dans el_j

➡ Le message *m* a donc été émis par P_i après que celui-ci ait reçu le dernier message de contrôle **`EEL`** : $reception_i(\mathbf{EEL}) \rightarrow emission_i(m) \Rightarrow emission_{init}(\mathbf{EEL}) \rightarrow reception_i(\mathbf{EEL}) \rightarrow emission_i(m) \Rightarrow emission_{init}(\mathbf{EEL}) \rightarrow emission_i(m)$.

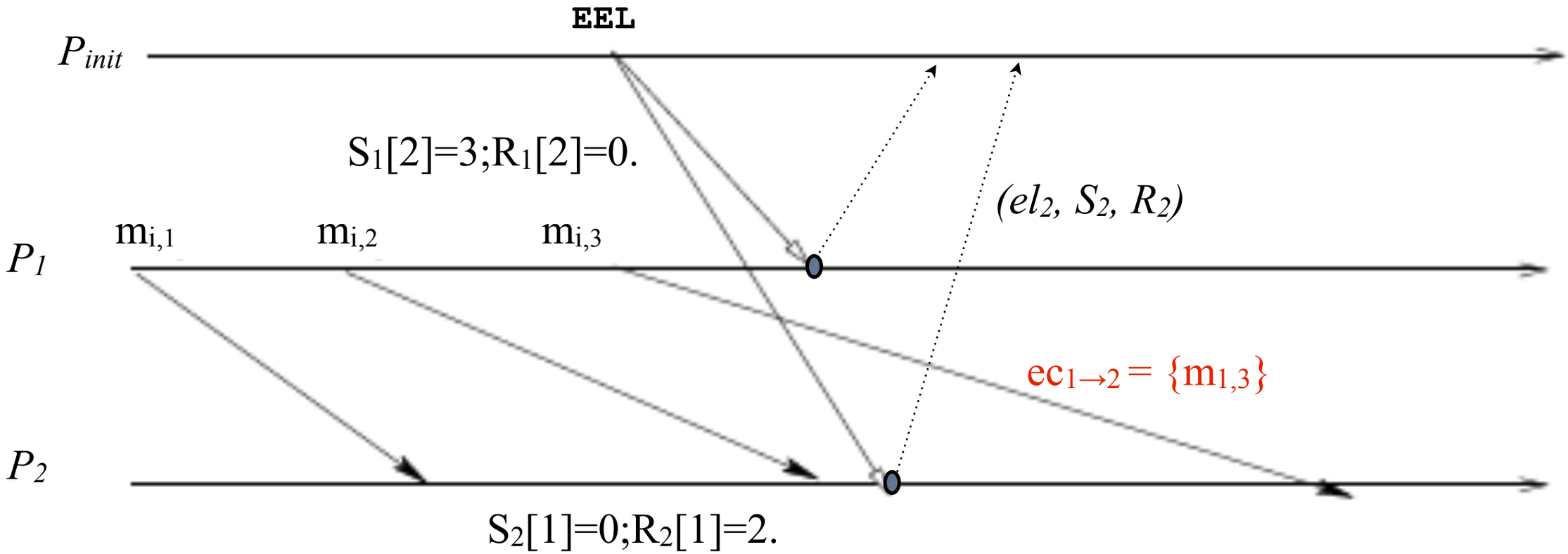
➡ Si sa réception était captée dans el_j on aurait $reception_j(m) \rightarrow reception_j(\mathbf{EEL})$ et l'hypothèse d'ordre causal serait violée.

Ordre causal, centralisée *Acharya & Badrika*

Calcul de l'état des canaux

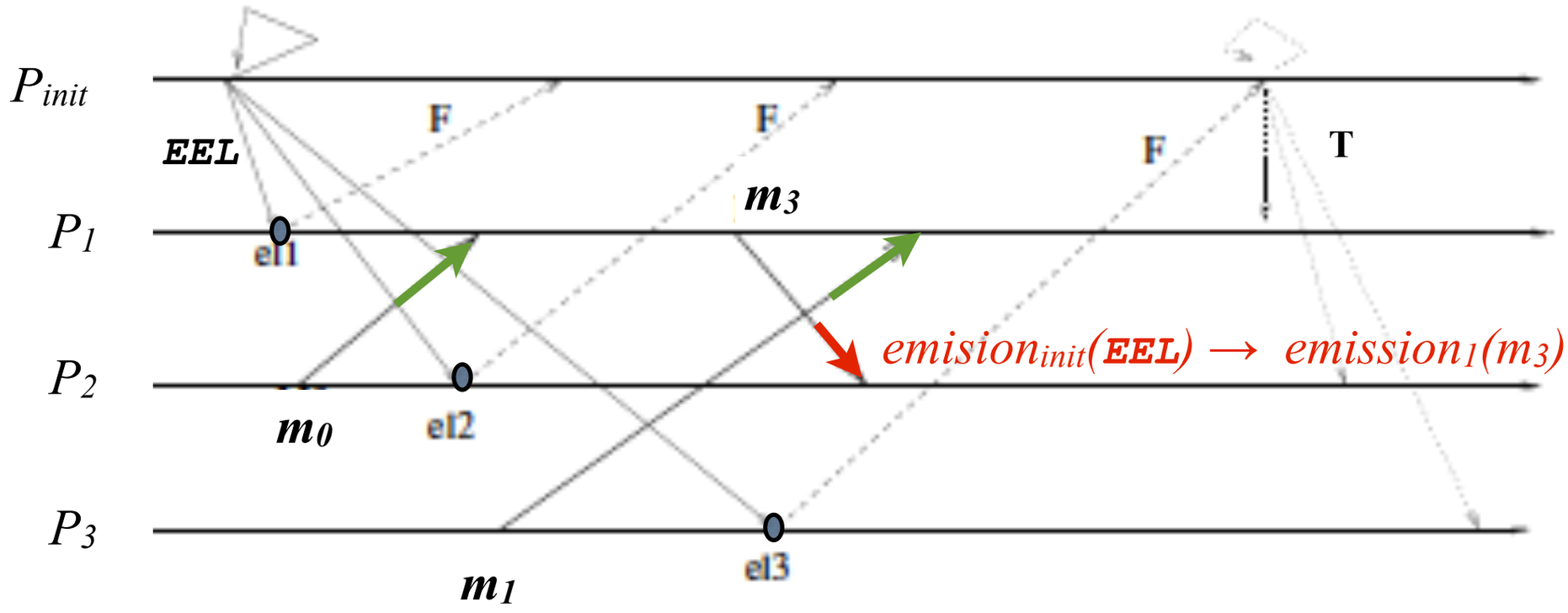
✓ Pour assurer la bonne capture de l'état des canaux, chaque P_i gère deux tableaux :

- ➡ **$S_i[N]$** tel que **$S_i[j] = n_{ij}$** : P_i a envoyé n_{ij} messages à P_j
- ➡ **$R_i[N]$** tel que **$R_i[j] = n_{ji}$** : P_i a reçu n_{ji} messages de P_j
  - ‣ Règle 2++ : Lorsqu'un P_i reçoit le message **EEL**, il enregistre son état local el_i et ses deux tableaux, puis il envoie à P_{init} le message (el_i, S_i, R_i)
  - ‣ Règle 3 : lorsque l'initiateur P_{init} a reçu tous les messages (el_i, S_i, R_i), il possède tous les états locaux et calcule l'état des canaux par : $\forall i,j,\ ec_{i\rightarrow j} = \{m_{i,\,Rj[i]+1} \ldots m_{i,\,Si[j]}\}$ *si* $S_i[j] \geq R_j[i]+1$



Ordre causal, répartie *Alagar & Venkatesan*

- Coloration à la réception...
  - ✓ un msg *m* reçu est coloré en rouge ssi $emision_{init}(\mathbf{EEL}) \rightarrow emission_i(m)$ (relation causal)



R2 : Lorsque P_i reçoit **EEL**, il enregistre son état local el_i, initialise l'état de ses canaux entrants à ∅ et répond à P_{init} un message **F**(AIT)
R3 : Après avoir enregistré son état local, si P_j reçoit un message sur son canal entrant $C_{i \rightarrow j}$ il teste si ce message est à colorer en rouge. S'il ne l'est pas, il le met dans l'ensemble $ec_{i \rightarrow j}$
R4 : Lorsque P_{init} a reçu un **F** de tous les processus, il diffuse **T**(ERMINE)
R5 : Lorsque P_i reçoit **T** il enregistre l'état de ses canaux

Démonstration *Alagar & Venkatesan*

‣ Lorsque que P_j reçoit **T**, il a nécessairement reçu de chaque $P_{i\neq j}$ tous les messages m enregistrés dans les états locaux des $P_{i\neq j}$

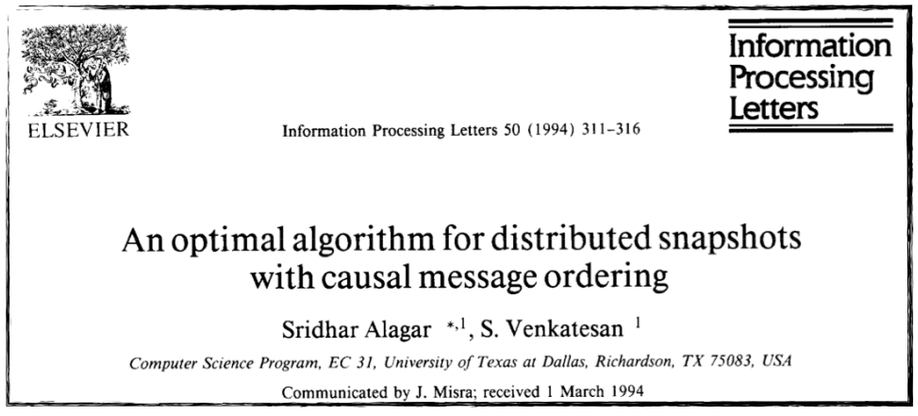
➡ Soit m un tel message sur $C_{i\to j}$:

- $emission_i(m) \rightarrow reception_i(\boldsymbol{EEL})$: $m \in el_i$
- $reception_i(\boldsymbol{EEL}) \rightarrow emission_i(\boldsymbol{F})$: règle R2
- $emission_i(\boldsymbol{F}) \rightarrow reception_{init}(\boldsymbol{F})$: par définition
- $reception_{init}(\boldsymbol{F}) \rightarrow emission_{init}(\boldsymbol{T})$: règle R4

=> $emission_i(m) \rightarrow emission_{init}(\boldsymbol{T})$: transitivité de la relation de causalité

=> $reception_j(m) \rightarrow reception_j(\boldsymbol{T})$: ordre causal !

=> P_j a reçu tous les messages nécessaires lorsqu'il reçoit **T**


ELSEVIER

Information Processing Letters 50 (1994) 311–316

Information Processing Letters

An optimal algorithm for distributed snapshots with causal message ordering

Sridhar Alagar *,[1], S. Venkatesan [1]

Computer Science Program, EC 31, University of Texas at Dallas, Richardson, TX 75083, USA

Communicated by J. Misra; received 1 March 1994

‣ Les «bons» messages reçus après $reception_j(\boldsymbol{EEL})$ ($reception_j(m)$ non captée directement dans el_j) seront bien mis dans l'état du canal $C_{i\to j}$

➡ Soit m un tel message => $reception_j(EEL) \rightarrow reception_j(m)$, or m n'est pas à colorer en rouge ($emission_i(m)$ est captée dans el_i) car $emission_{init}(EEL) \not\rightarrow emission_i(m)$

=> P_j enregistre le message dans $ec_{i\to j}$ (R3 & R5)

‣ Réciproquement, il est évident que si m est enregistré dans $ec_{i\to j}$

=> $reception_j(EEL) \rightarrow reception_j(m) \rightarrow reception_j(T)$ => m n'est pas coloré en rouge

=> $reception_j(m)$ n'est pas captée dans el_j et $emission_i(m)$ est capturée dans el_i