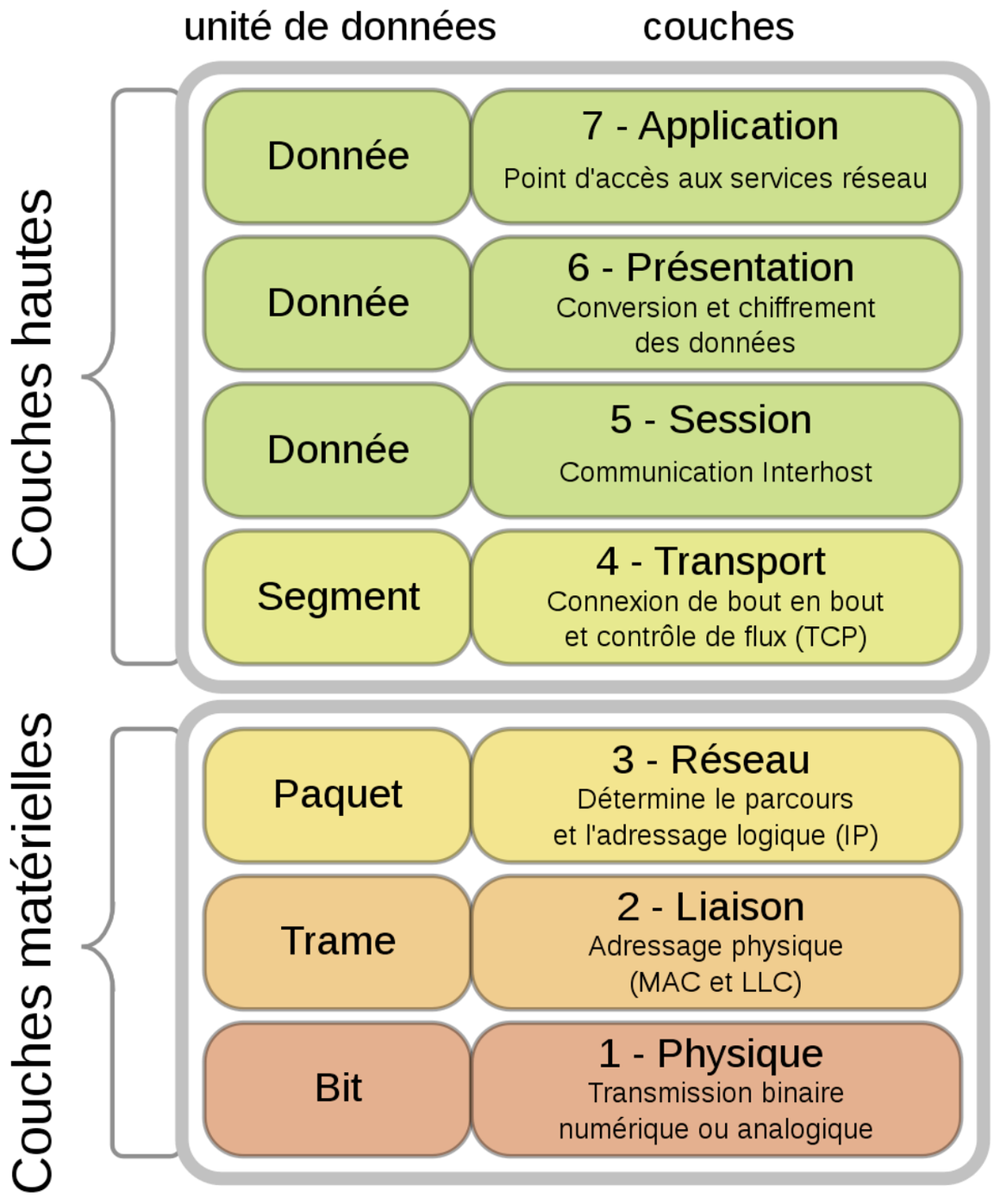
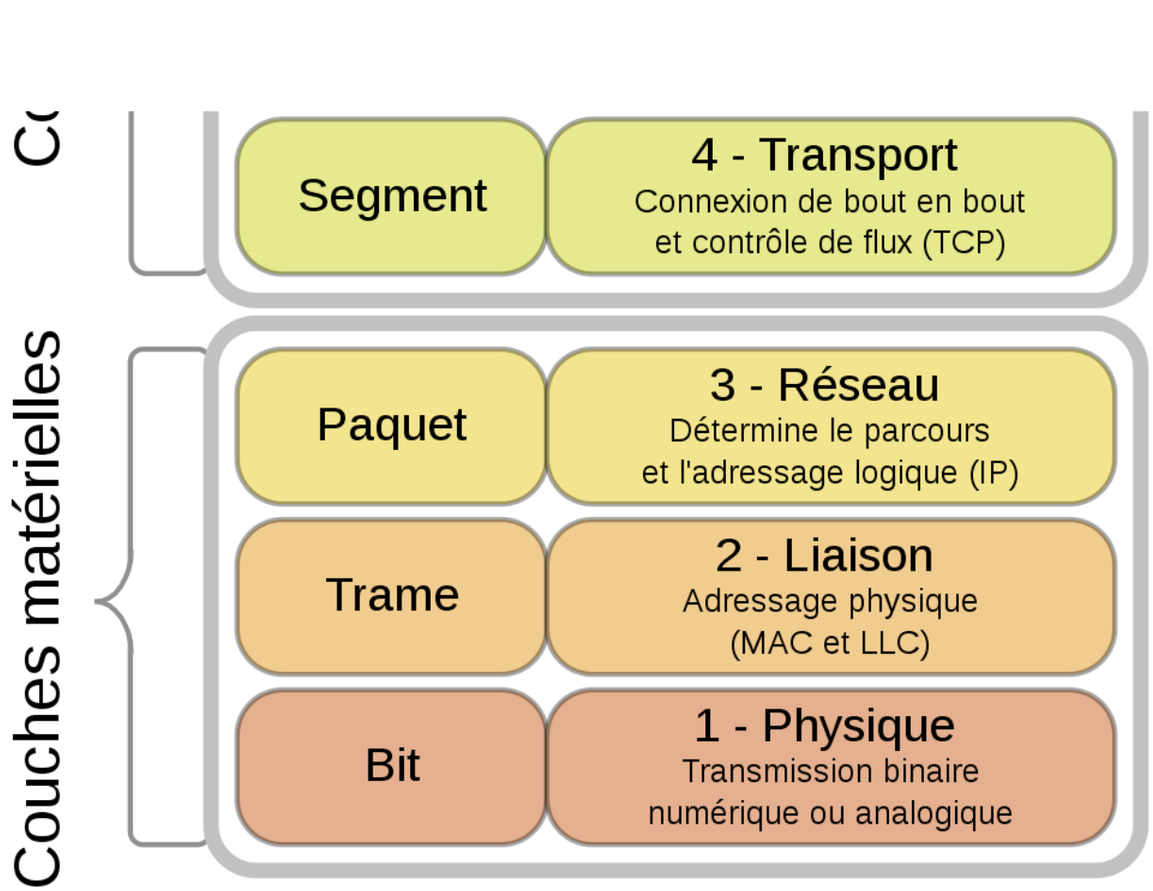
Algorithmes Distribués

Application réseau
L3 - AMU
2021-2022
corentin travers

Modèle en couche



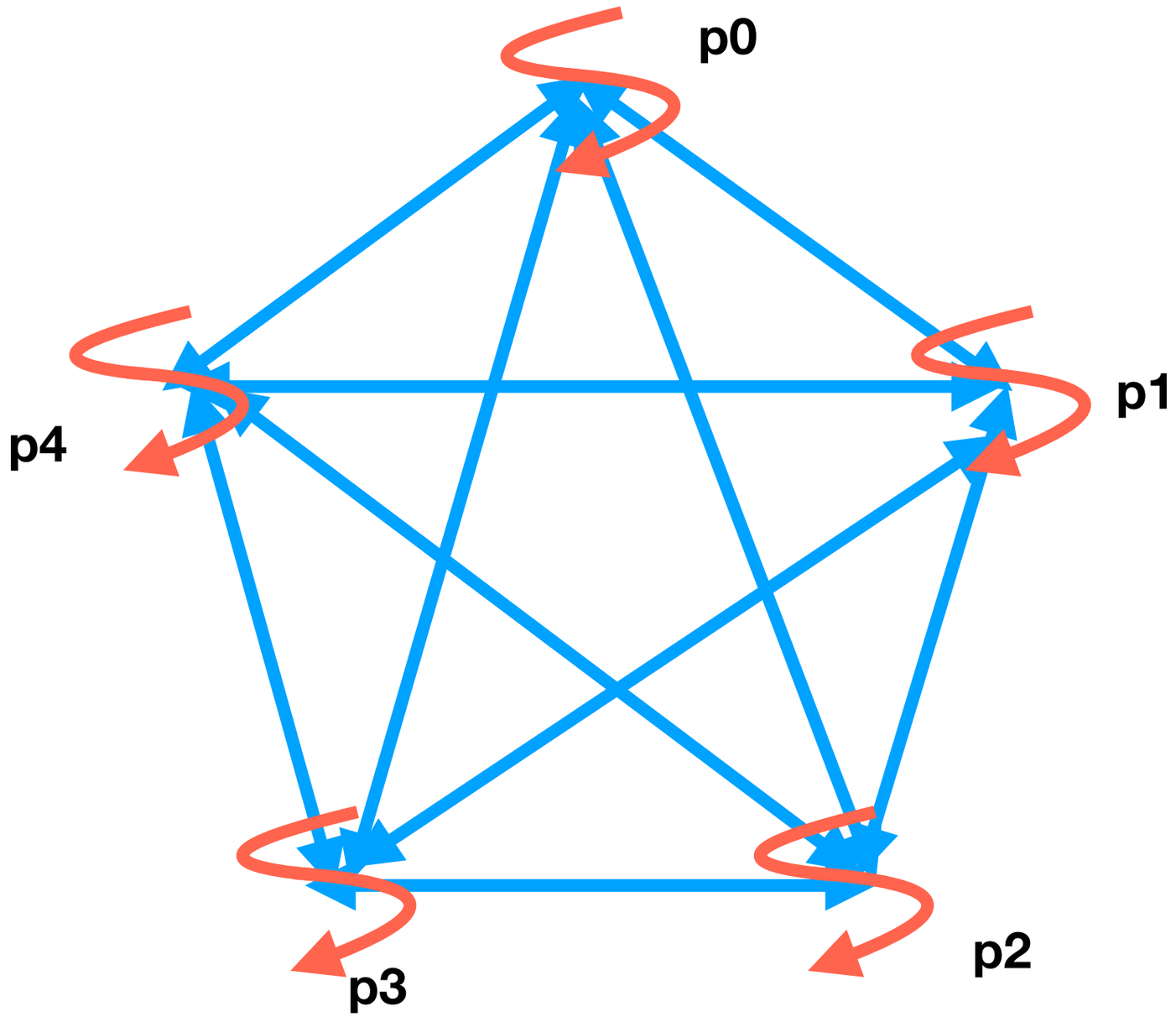
Abstraction



Couche transport

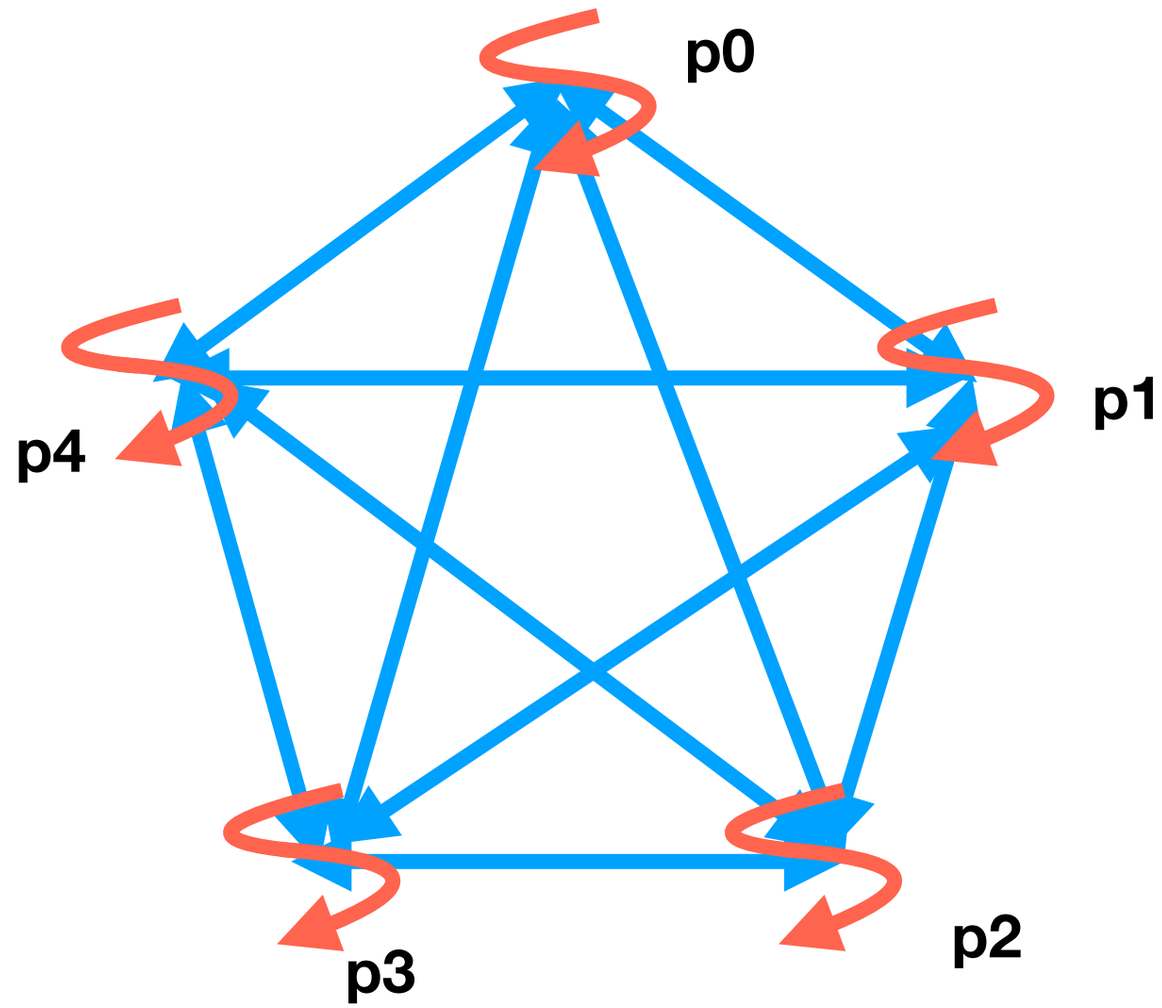
- **Canal** de communication
- Point-à-point : Entre **2 processus**
- **Fiable** (TCP) ou non (UDP)
- **Asynchrone** temps de transfert des messages non borné

Système distribué

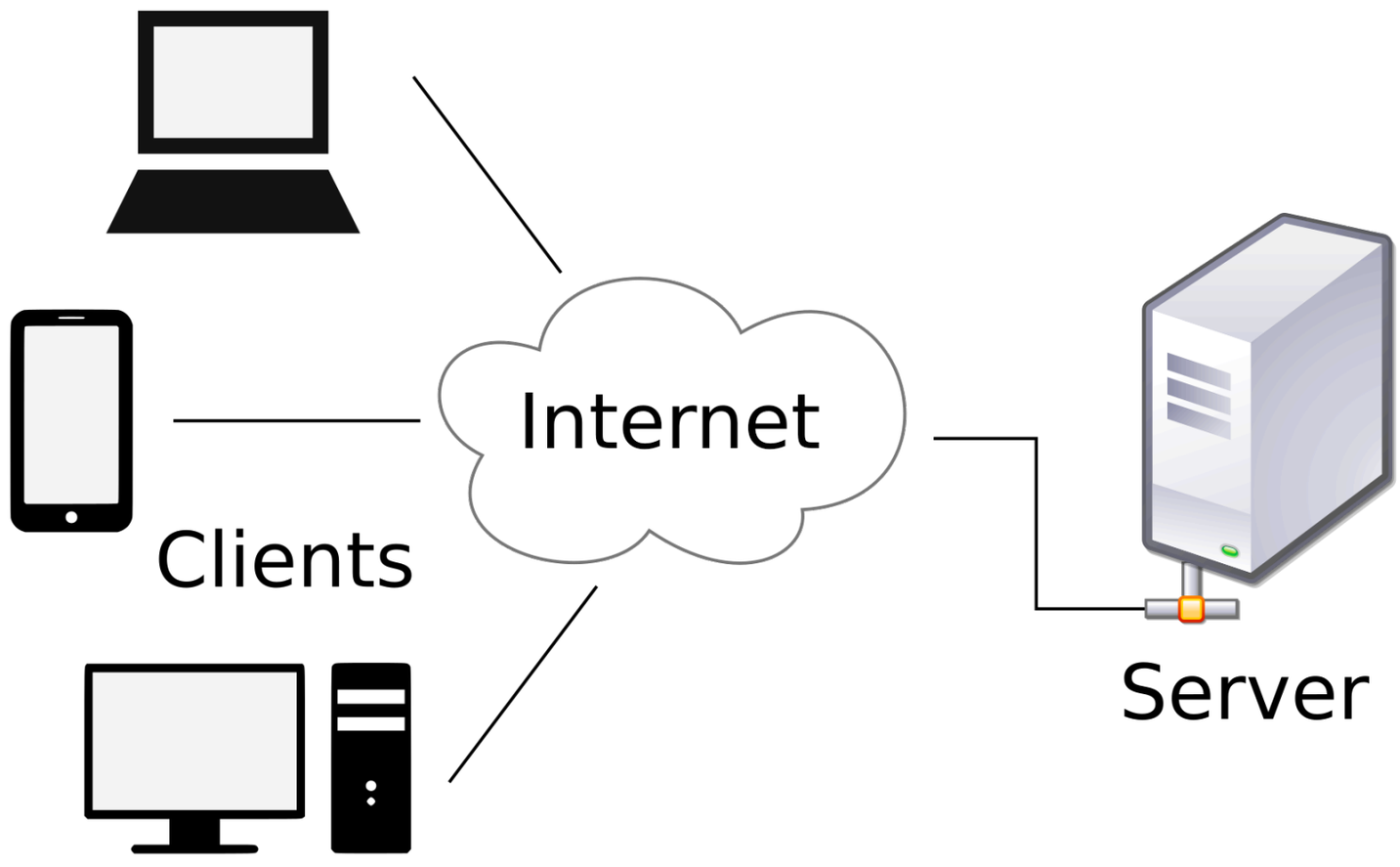


Système distribué

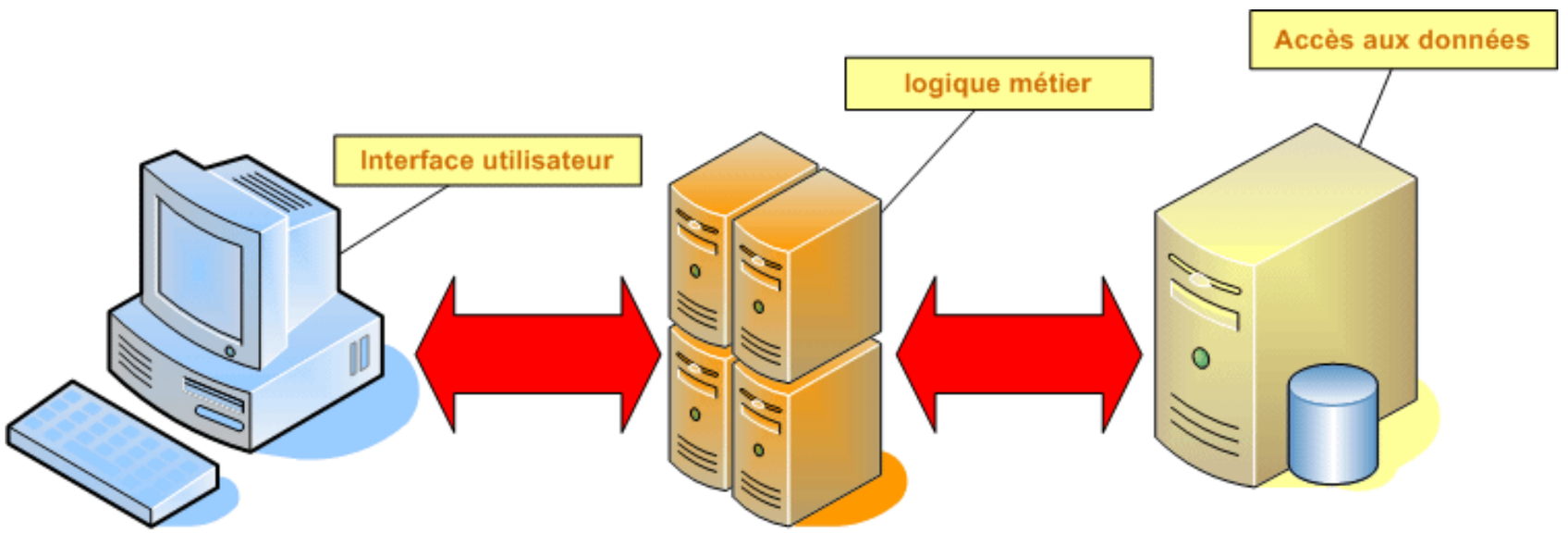
- Processus
- Canaux de communication
- Communication par **messages**



Client/Serveur

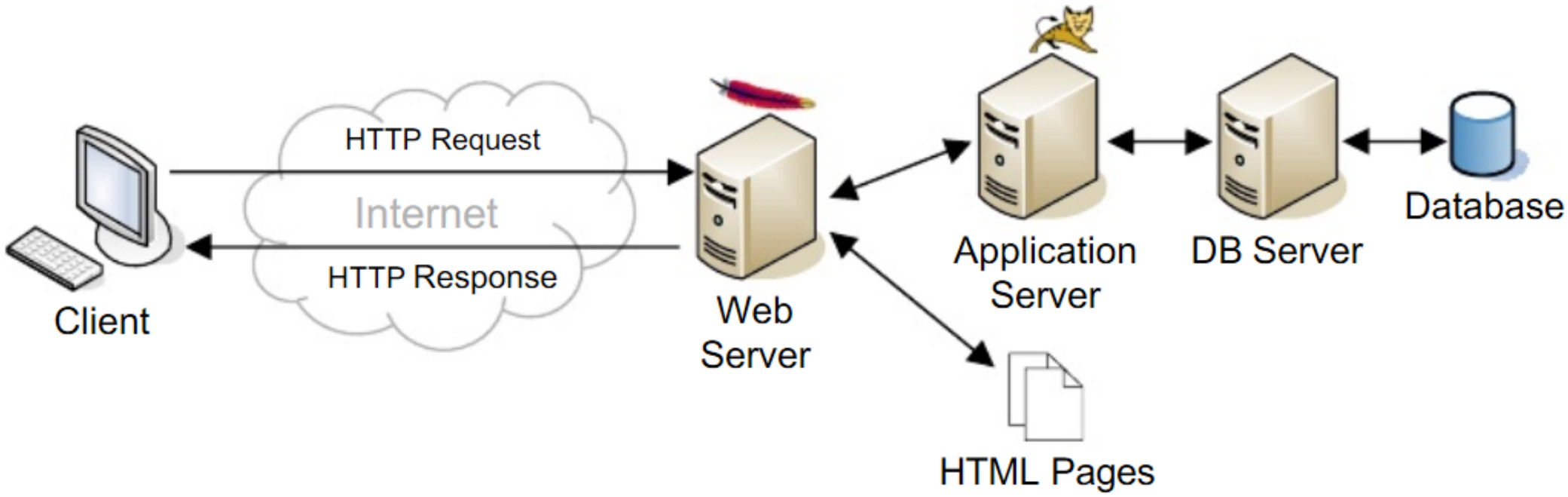


Architecture 3-tiers

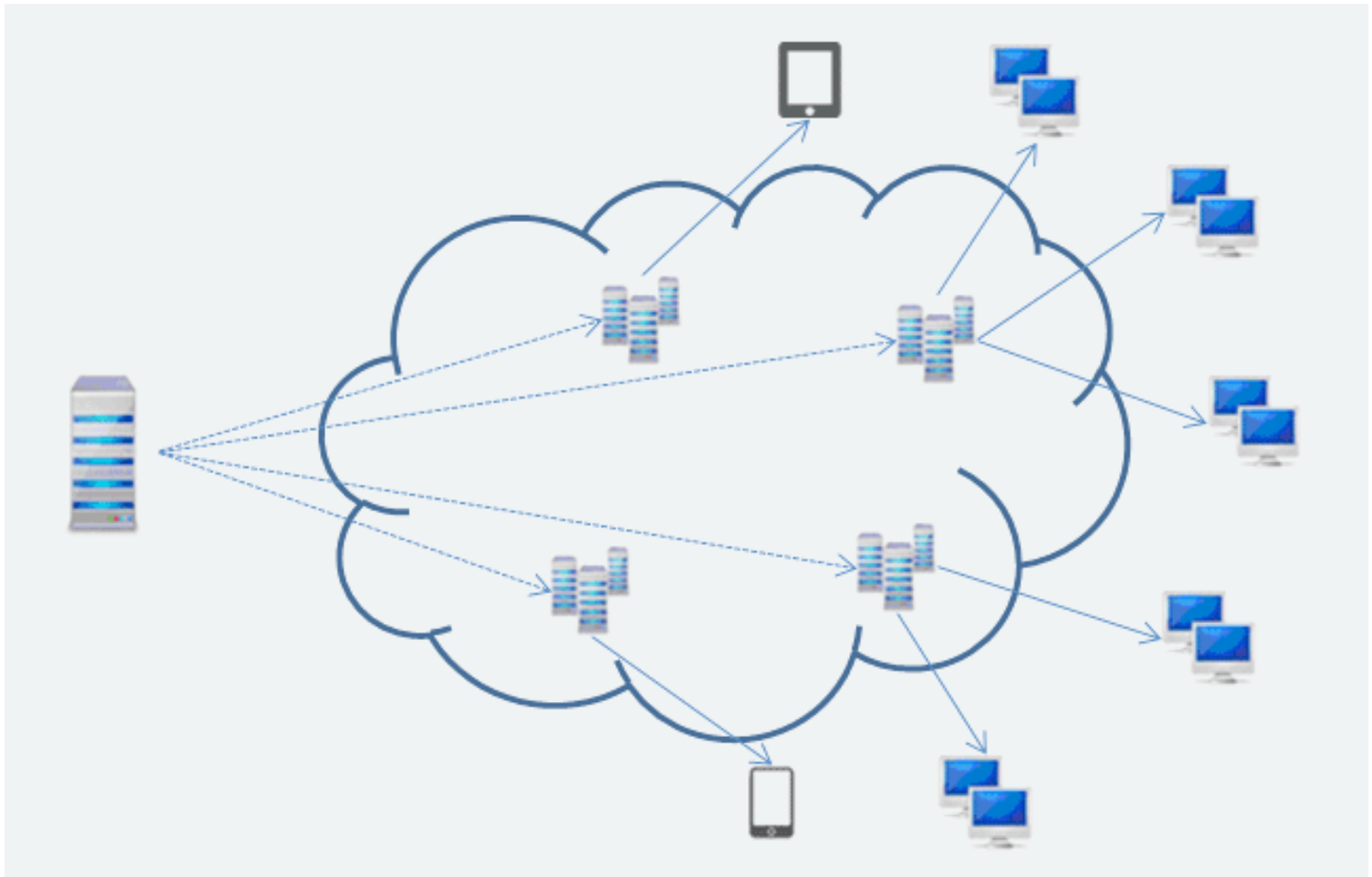


- **Présentation : le client**
- **Traitement : serveur(s)**
- **Accès/stockage des données:BDD**

Architecture N-tiers



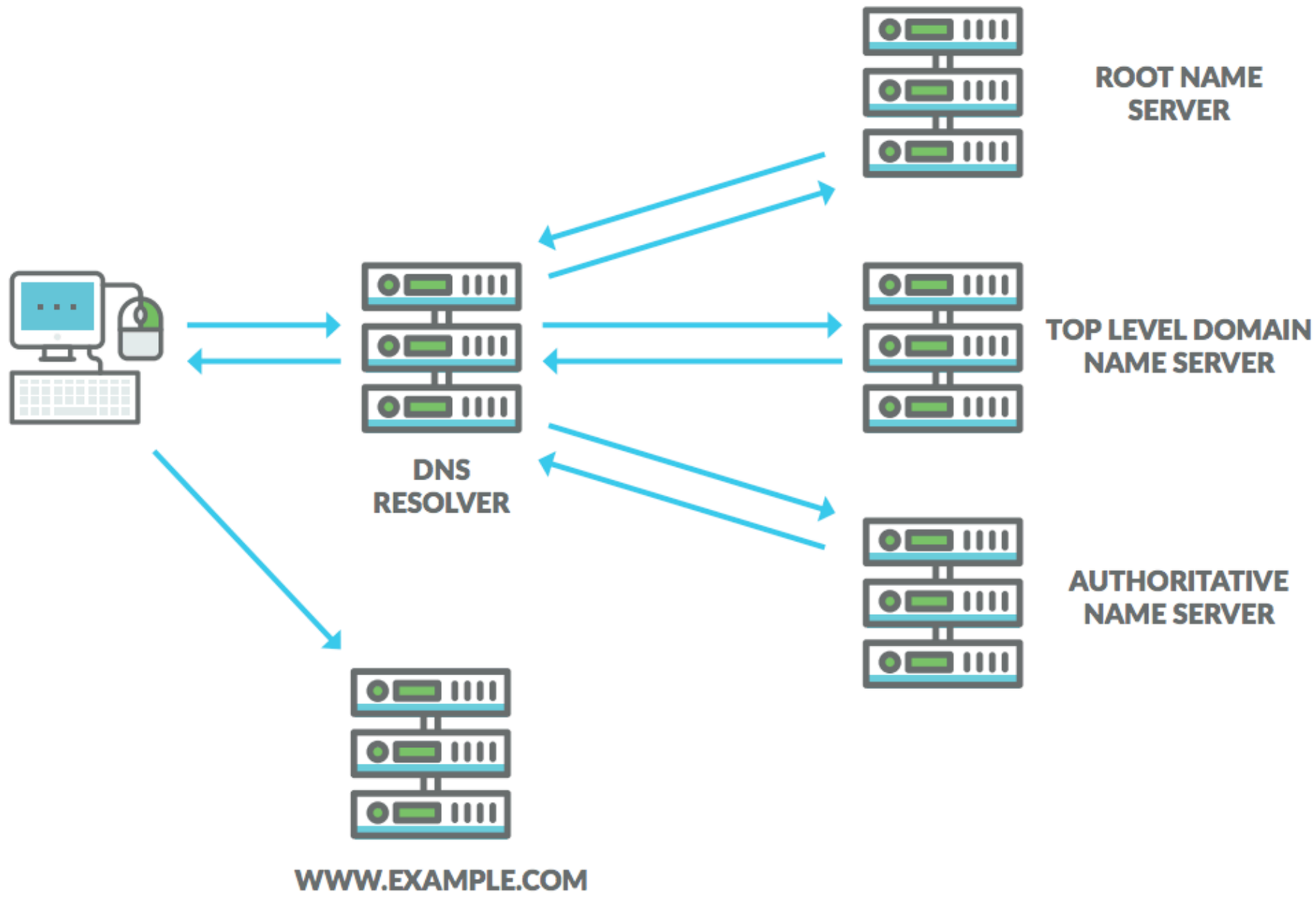
Content Delivery Network



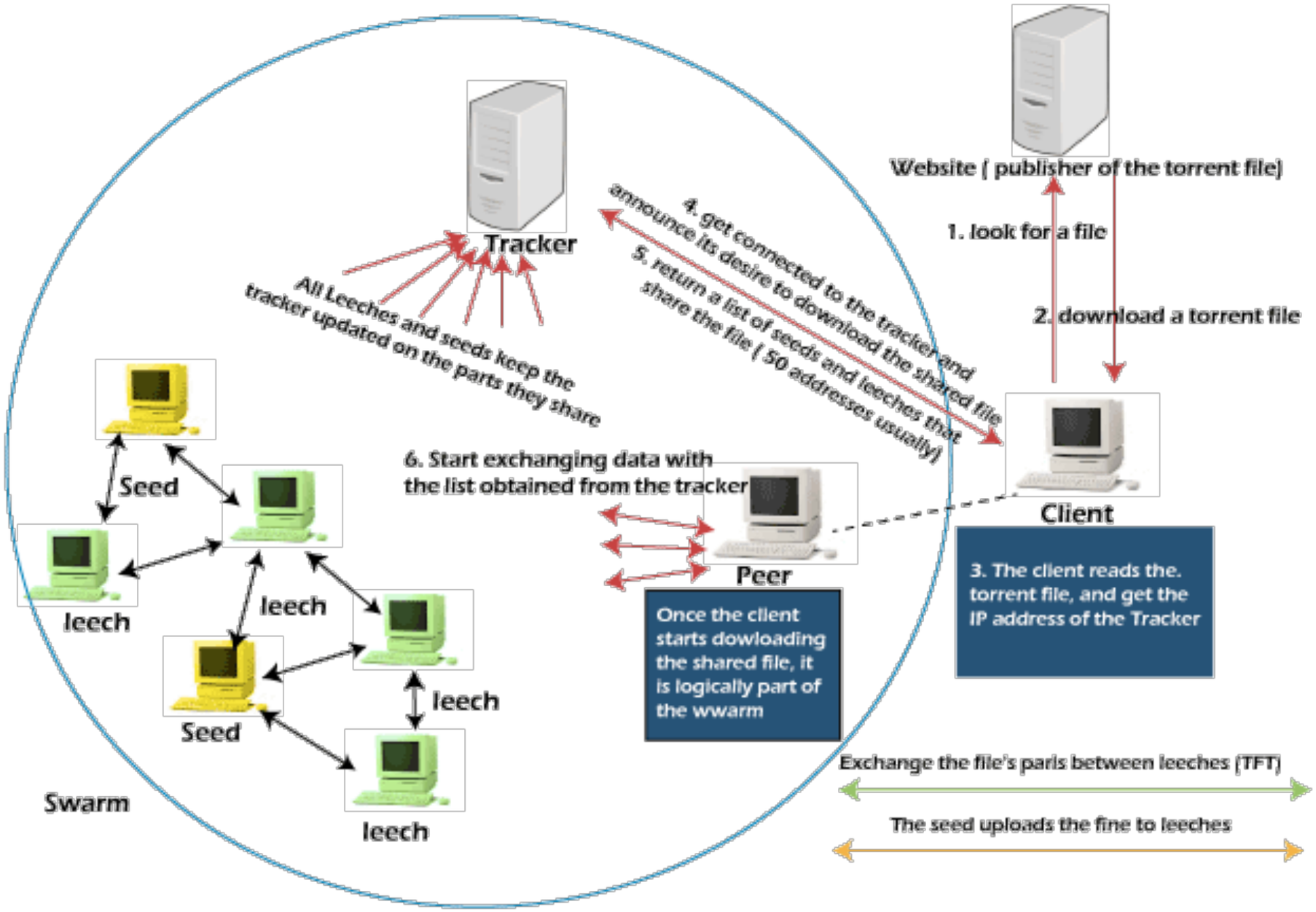
Diffusion de contenu

- **Répartition de la charge**
- **Caching**
- **Disponibilité**

DNS


ROOT NAME SERVER
TOP LEVEL DOMAIN NAME SERVER
DNS RESOLVER
AUTHORITATIVE NAME SERVER
WWW.EXAMPLE.COM

Partage de fichier P2P



Système distribué

- Ensemble de **processus** p0,p1,…pN
- Qui peuvent communiquer par transmission de **messages**
- Doivent se **coordonner** pour effectuer une tâche commune/rendre un service

Système distribué : problématiques

- Processus peuvent mal-fonctionnés : panne, crash, bug, piratage

- Messages peuvent être perdus (UDP), modifiés, espionnés, …

- Messages peuvent être retardés : délais de transfert **non bornés**

Exécution

p0:
```
m0 =  new Message()
send m0 to p1, p2
```

p1:
```
upon reception of m from p2
send m to p0, p2
```

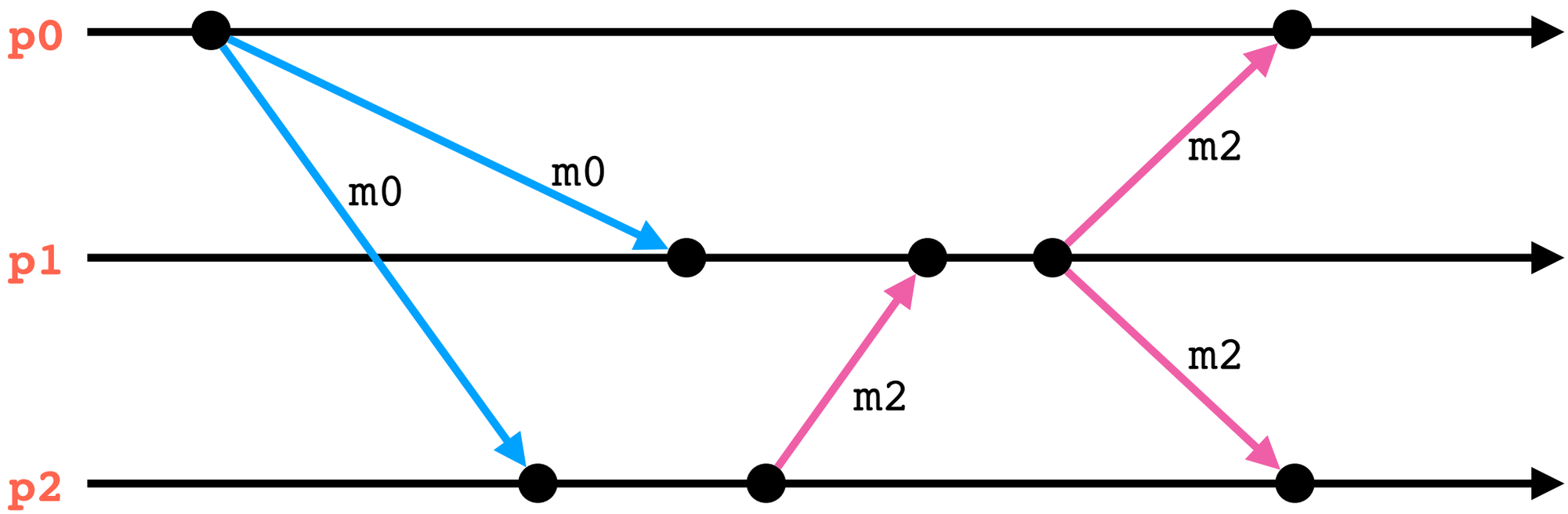
p2:
```
m2 =  new Message()
send m2 to p1
```

Exécutions

p0:
```
m0 =  new Message()
send m0 to p1, p2
```

p1:
```
upon reception of m from p2
send m to p0, p2
```

p2:
```
m2 =  new Message()
send m2 to p1
```

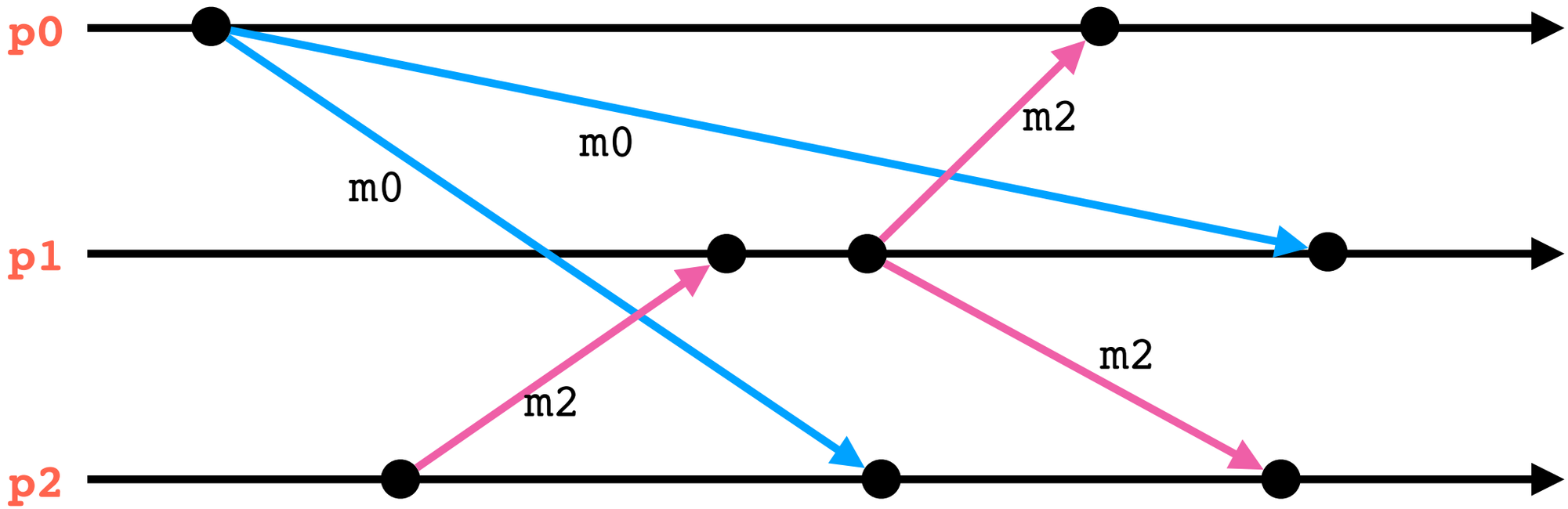


Exécutions

```
p0:
m0 =  new Message()
send m0 to p1, p2
```

```
p1:
upon reception of m from p2
send m to p0, p2
```

```
p2:
m2 =  new Message()
send m2 to p1
```

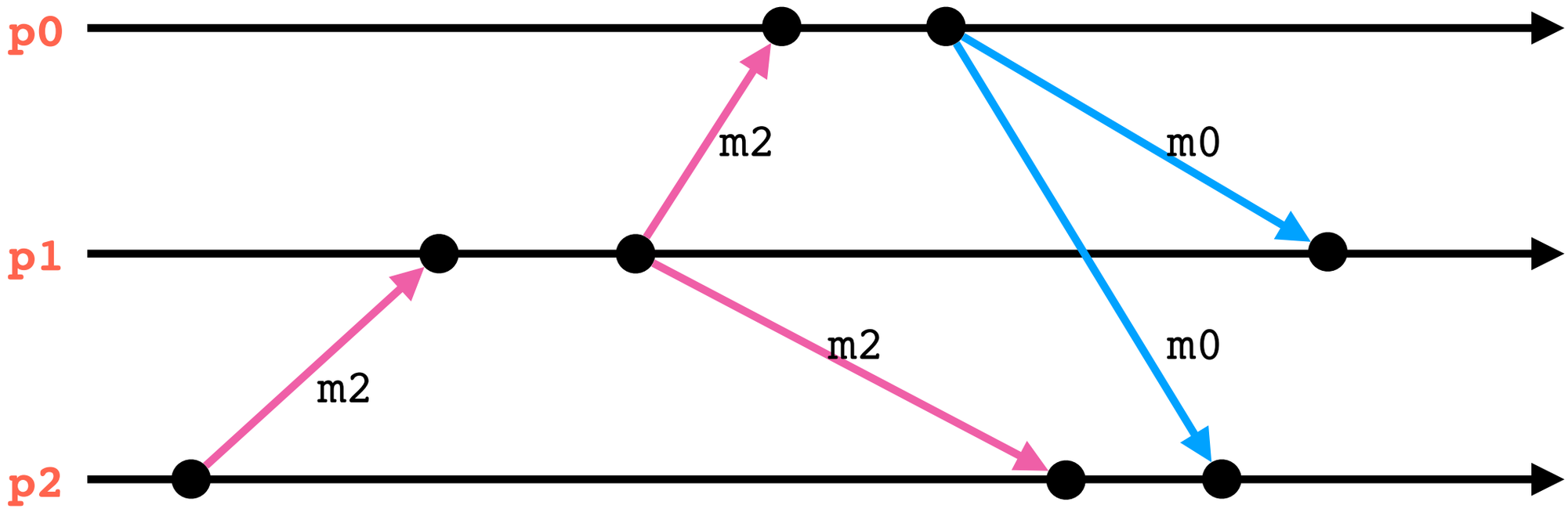


Exécutions

```
p0:
m0 =  new Message()
send m0 to p1, p2
```

```
p1:
upon reception of m from p2
send m to p0, p2
```

```
p2:
m2 =  new Message()
send m2 to p1
```



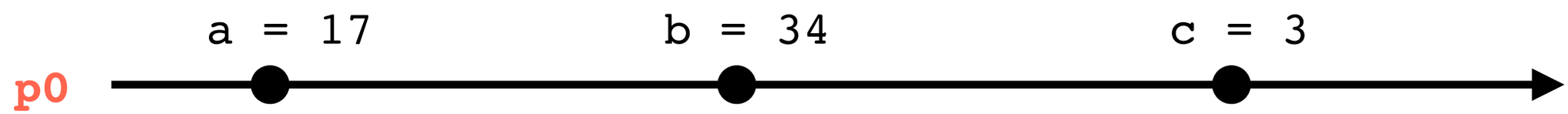
Exécutions distribuées

p0:	p1:	p2:
`m0 = new Message()` `send m0 to p1, p2`	`upon reception of m from p2` `send m to p0, p2`	`m2 = new Message()` `send m2 to p1`

- Même programme
- De nombreuses exécutions possibles ! Non déterminisme !
- Ordre de réception des messages imprévisibles !
- Vitesses relatives des processus imprévisibles !

Exécution séquentielle

```
p0:
a = 17;
b = 2*a;
if(b == 0) { …}
else {c= 3]
```



- Une seule exécution ! déterministe !

Evénements

- Un **événement** est l'exécution d'une instruction par un processus

- **événement** lié à un canal de communication
  - **envoi** d'un message sur ce canal
  - **réception** d'un message sur ce canal
- **événement interne :** tout autre événement
- On s'intéresse aux événements de communication et à certains événements internes (dépend de l'application)

Exécution séquentielle

- Une exécution séquentielle est une **séquence d'événements**

- Trace de l'exécution d'un programme (cf. déboggage avec `printf`, fichier de log)

- Ordre **total** sur les événements dans une exécution séquentielle

Exécution distribuée

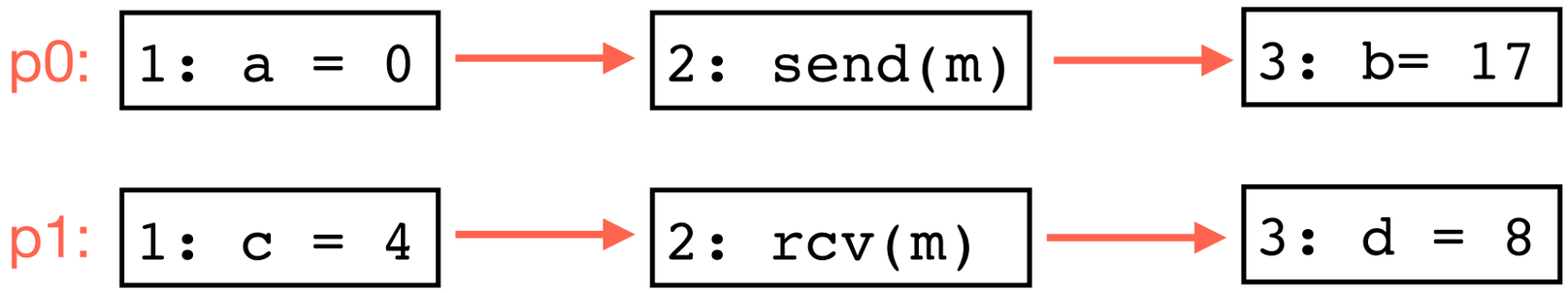
- (En général), pas d'**ordre total** sur les événements
- Certains événements sont toutefois ordonnés:
  - événements sur un même processus
  - envoi d'un message *précède* sa réception

Exécution distribuée

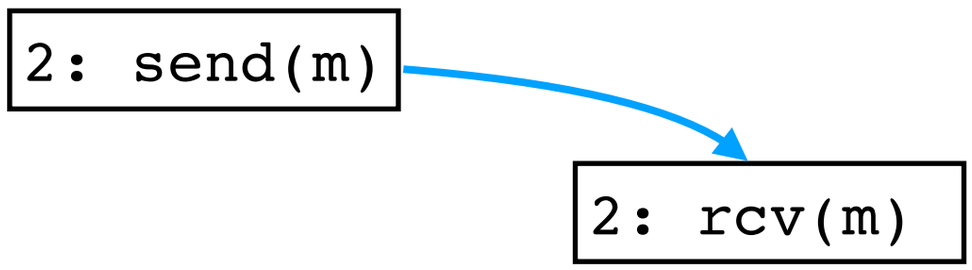
p0: trace
```
1: a = 0
2: send(m) to p1
3: b = 17
```

p1: trace
```
1: c = 4
2: rcv(m) from p1
3: d = 8
```

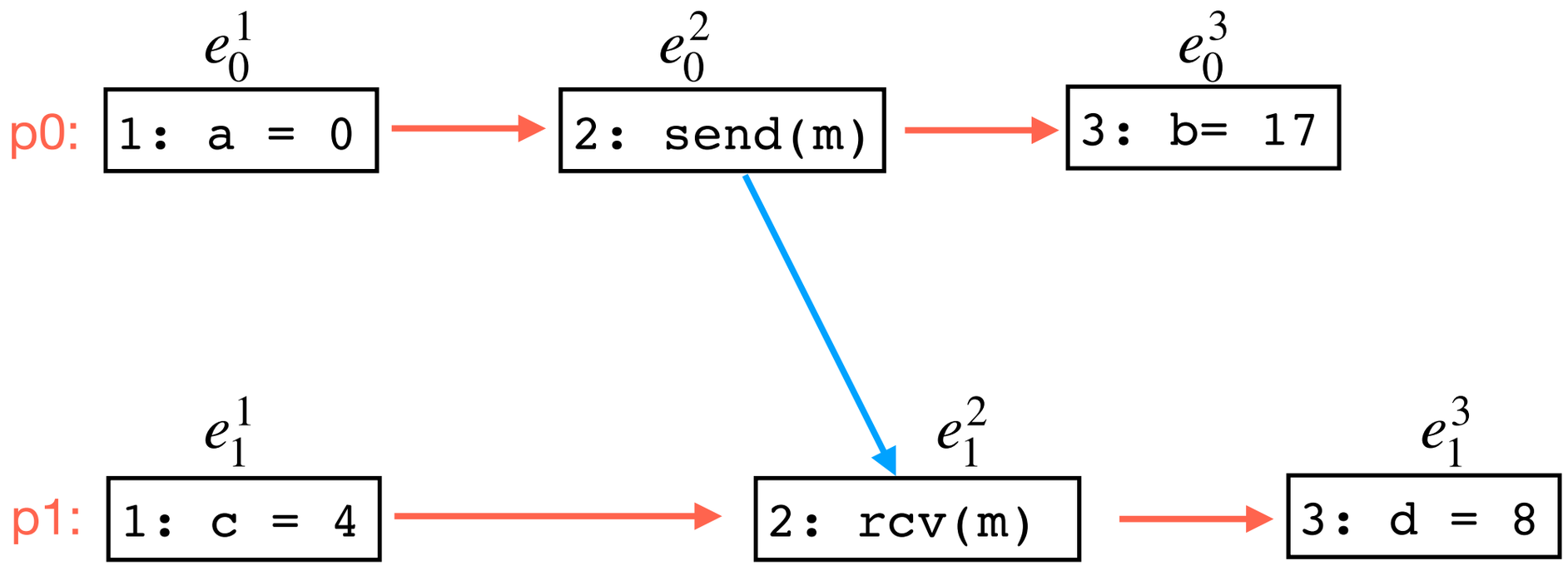
- Ordre induit par les processus :



- Ordre induit par les communication :



Ordre partiel

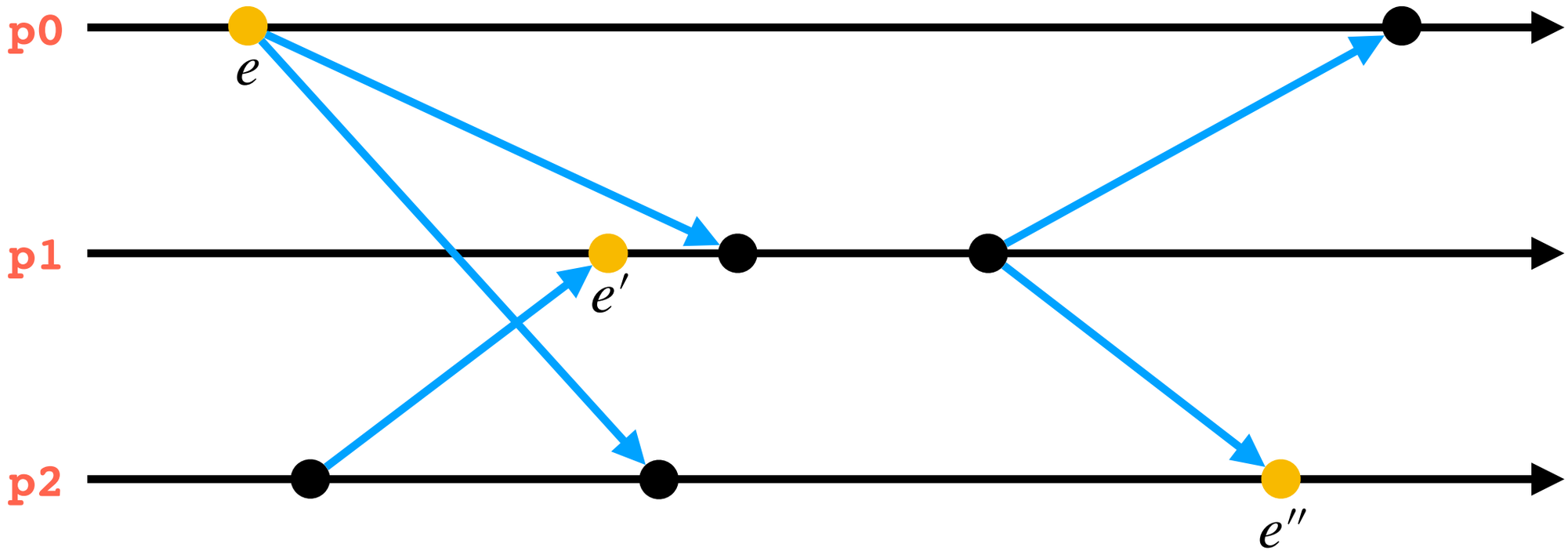


- e_0^1 précède e_0^2; e_0^2 précède e_1^2
- par transitivité, e_0^1 précède e_1^2
- e_0^1 et e_1^1 ne sont pas ordonnés

happens before
ordre causal

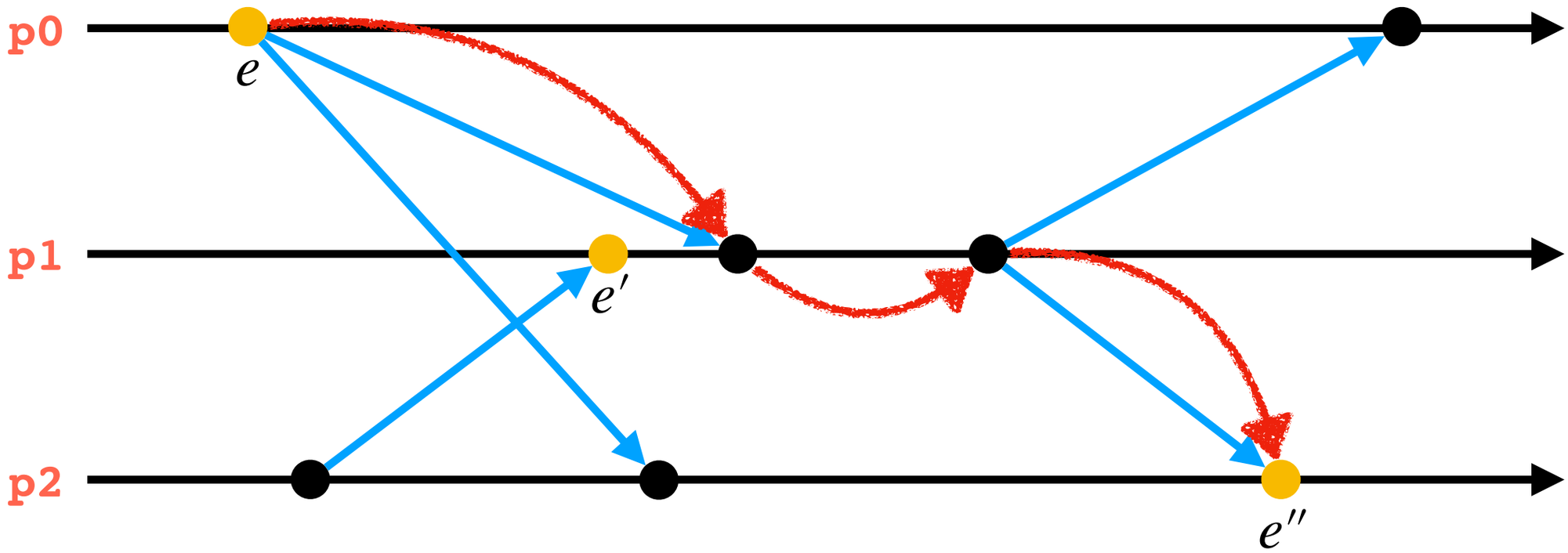
- $\mathbf{e \rightarrow e'}$: e **a lieu avant** e' ssi :
  - e,e' ont lieu sur le même processus et e précède e' ou
  - e = send(m) et e' = rcv(m) ou
  - il existe e" : $e \rightarrow e''$ et $e'' \rightarrow e'$

Ordre causal



- e et e’ ?
- e et e” ?

Ordre causal

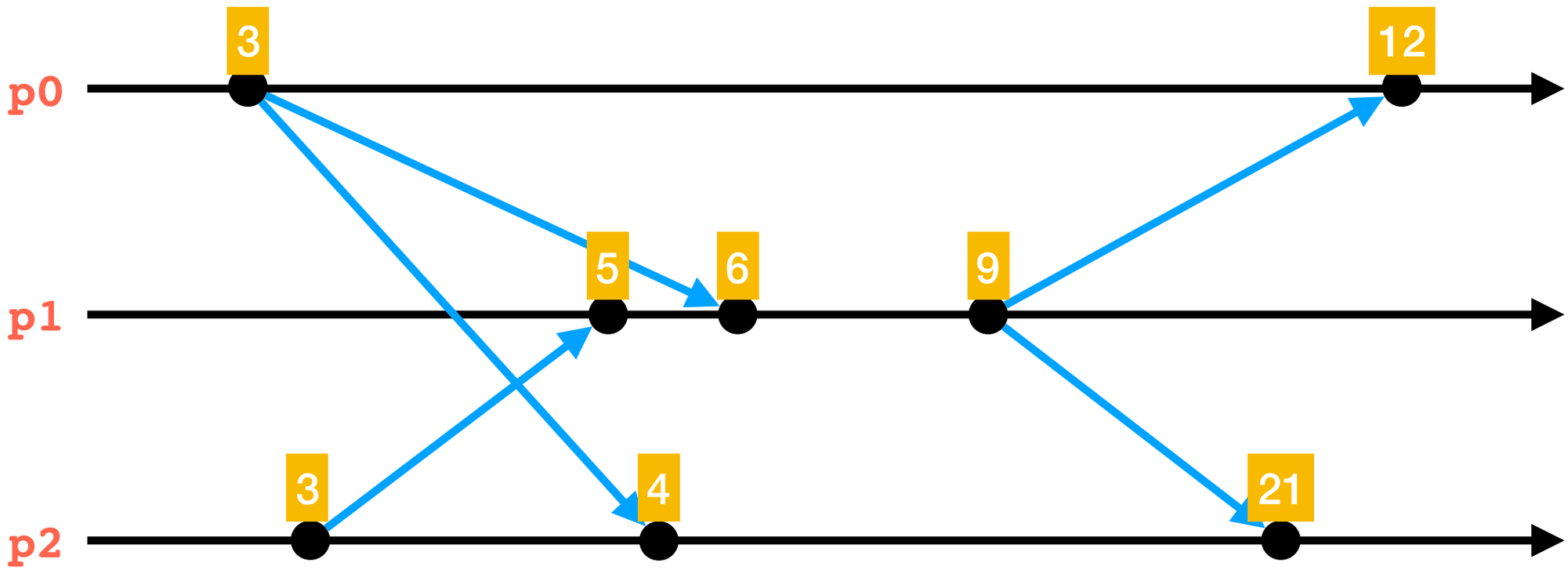


- e et e’ sont indépendants noté $\mathbf{e} \,||\, \mathbf{e}'$
- e précède causalement e” $\mathbf{e} \rightarrow \mathbf{e}''$

Horloge logique

- **Objectif :** attribuer à chaque évènement une *date* telle que
  - $\forall e, e' : e < e' \implies date(e) < date(e')$
- **Moyen :** algorithme distribué
- **Remarque:** on **ne** demande **pas** que
  $\forall e, e' : date(e) < date(e') \implies e < e'$

Exemple



- la date est un entier
- comment faire ?

Horloge de Lamport

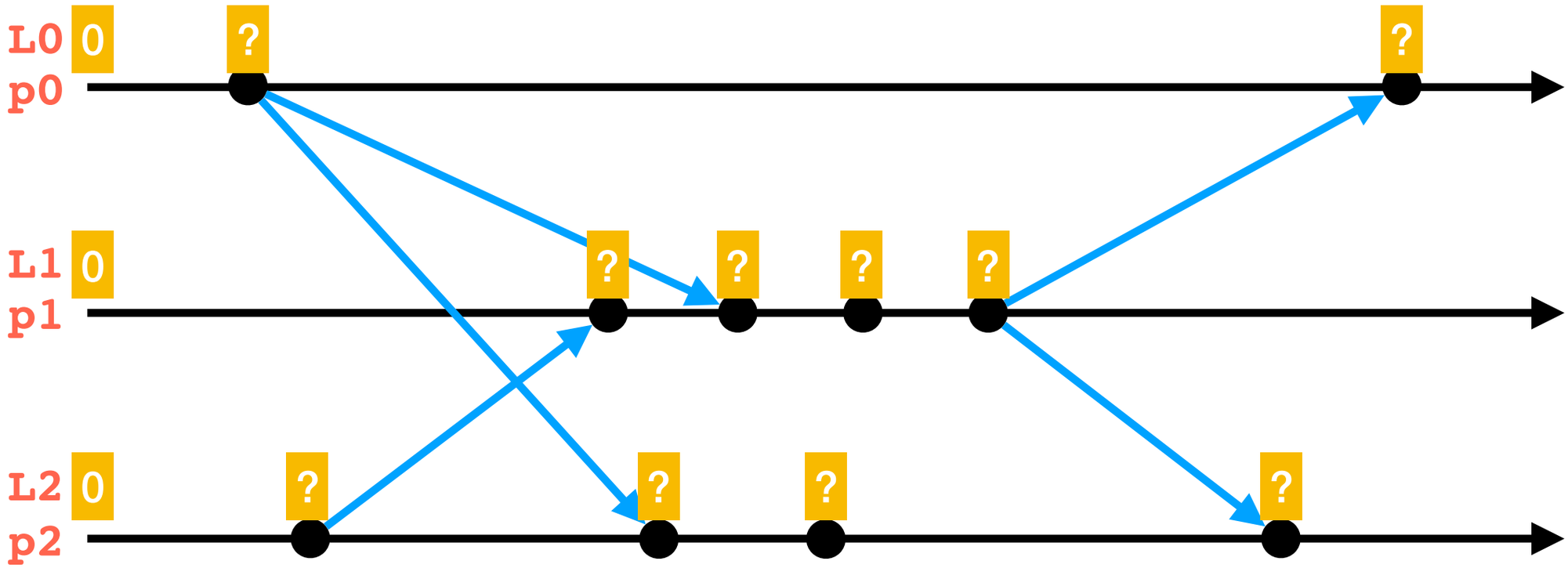
- Chaque processus p_i maintient une var. locale L_i
- 3 règles :

1.sur p_i incrémenter L_i avant chaque évènement interne:

2.lors de l'envoi d'un message m : incrémenter L_i puis envoyer le message avec L_i. I.e $L_i = L_i + 1$; `send(m,`L_i`)`

3.à la réception d'une paire (m,L_j) : m.a.j de L_i avec max(L_i,L_i) puis appliquer règle 1. `rcv(m,`L_j`);` $L_i = \max(L_i, L_j)$; $L_i = L_i + 1$

Horloge de Lamport



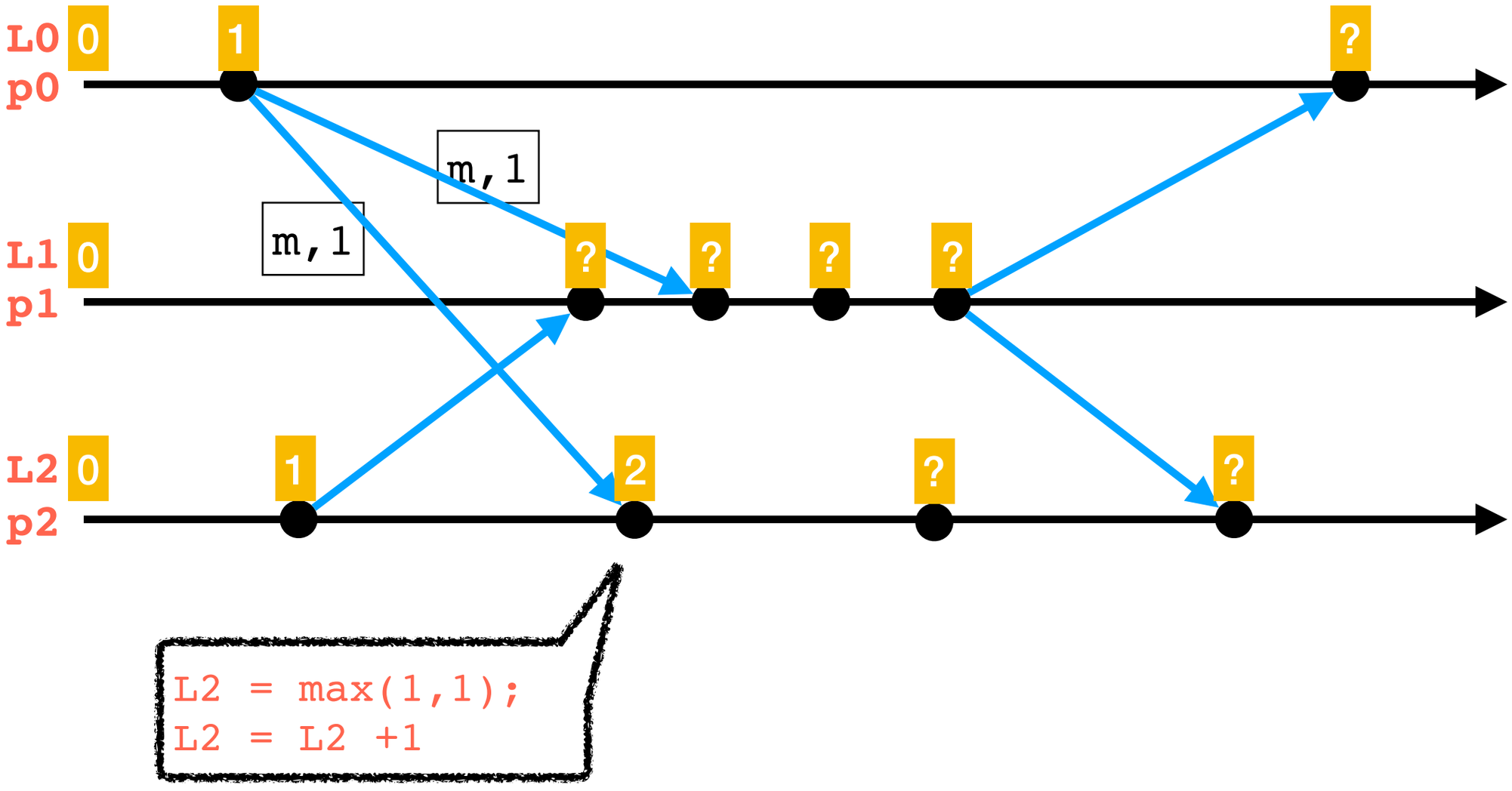
- 3 règles :

1.sur p_i incrémenter L_i avant chaque évènement interne: $L_i = L_i + 1$;

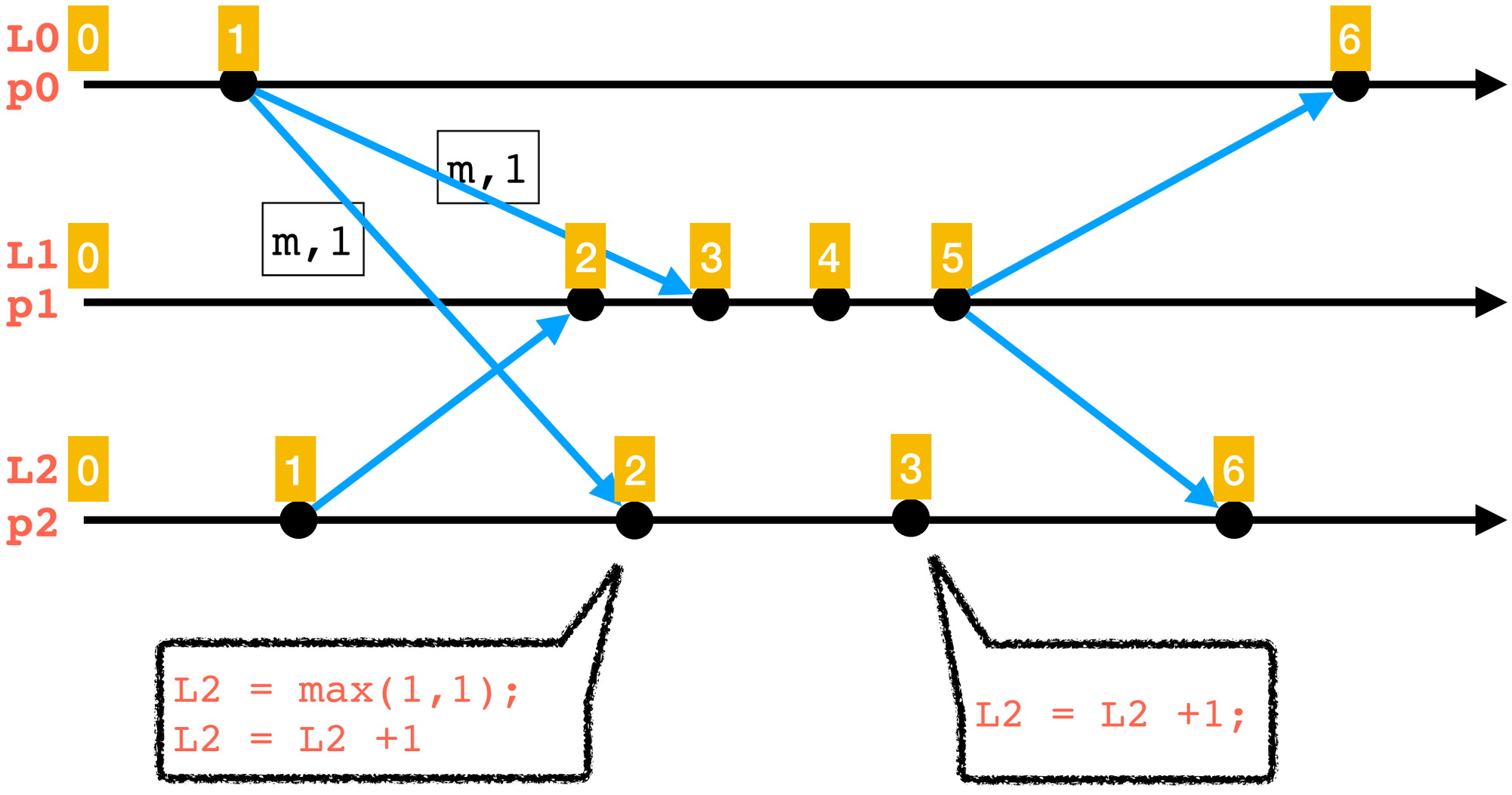
2.lors de l'envoi d'un message m : incrémenter L_i puis envoyer le message avec L_i.
$L_i = L_i + 1$; `send(m,`L_i`)`

3.à la réception d'une paire (m,L_j) : maj de L_i avec max(L_i,L_i) puis appliquer règle 1.
`rcv(m,`L_j`);` $L_i = \max(L_i, L_j)$; $L_i = L_i + 1$

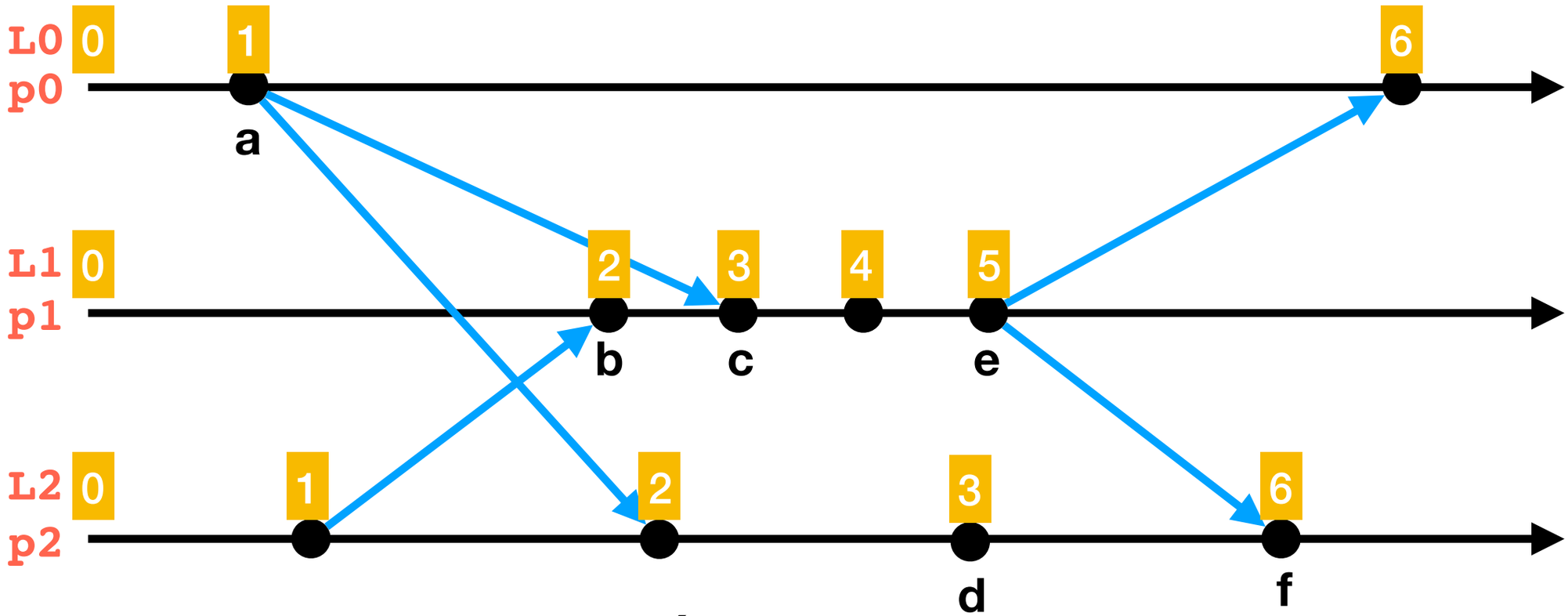
Horloge de Lamport



Horloge de Lamport



Horloge de Lamport

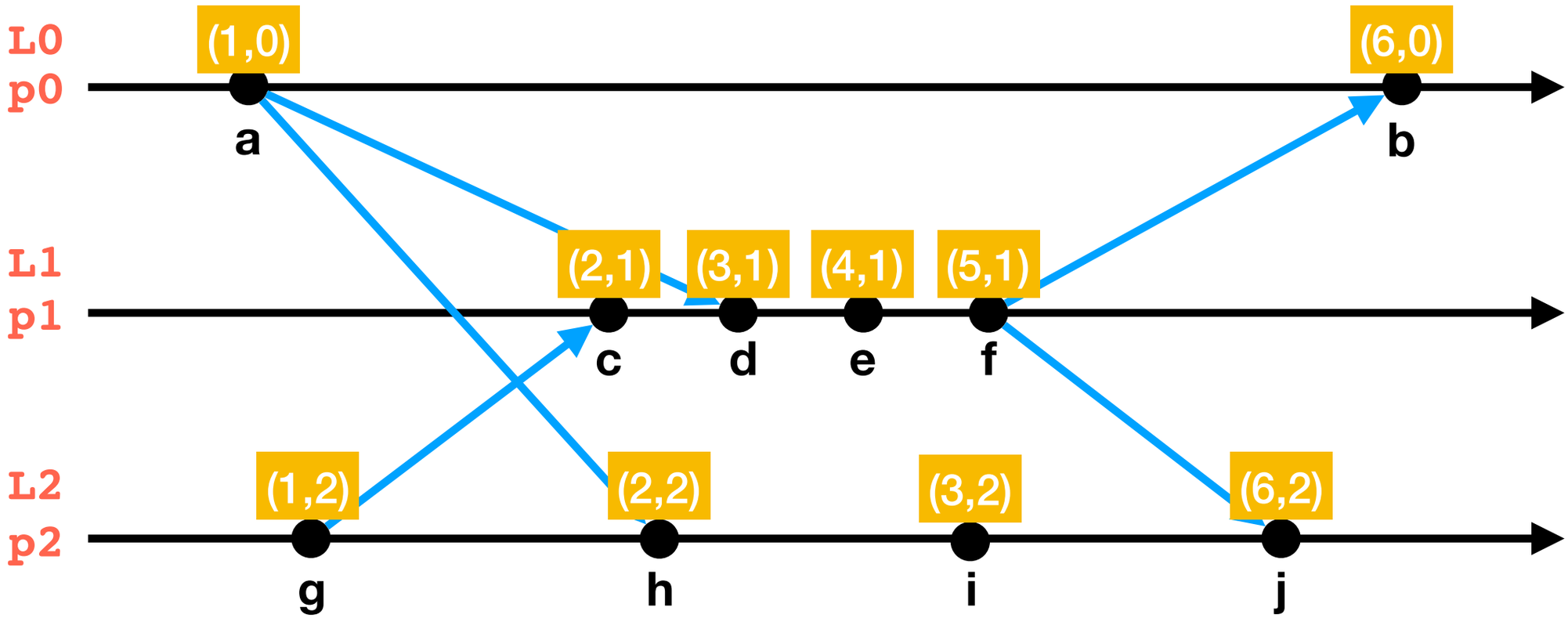


- **L est un ordre partiel cohérent avec l'ordre causal :**
- $\mathbf{a} \rightarrow \mathbf{f}$ on a bien L(a) < L(f)
- $\mathbf{a} \rightarrow \mathbf{c}$ on a bien L(a) < L(c)
- Par contre les évènements indépendants ont des dates arbitrairement ordonnées :
  - $\mathbf{b} \,||\, \mathbf{d}$ et L(b) < L(d) $\mathbf{c} \,||\, \mathbf{d}$ et L(c)= L(d) $\mathbf{e} \,||\, \mathbf{d}$ et L(e) < L(d)

D’un ordre partiel à un ordre total

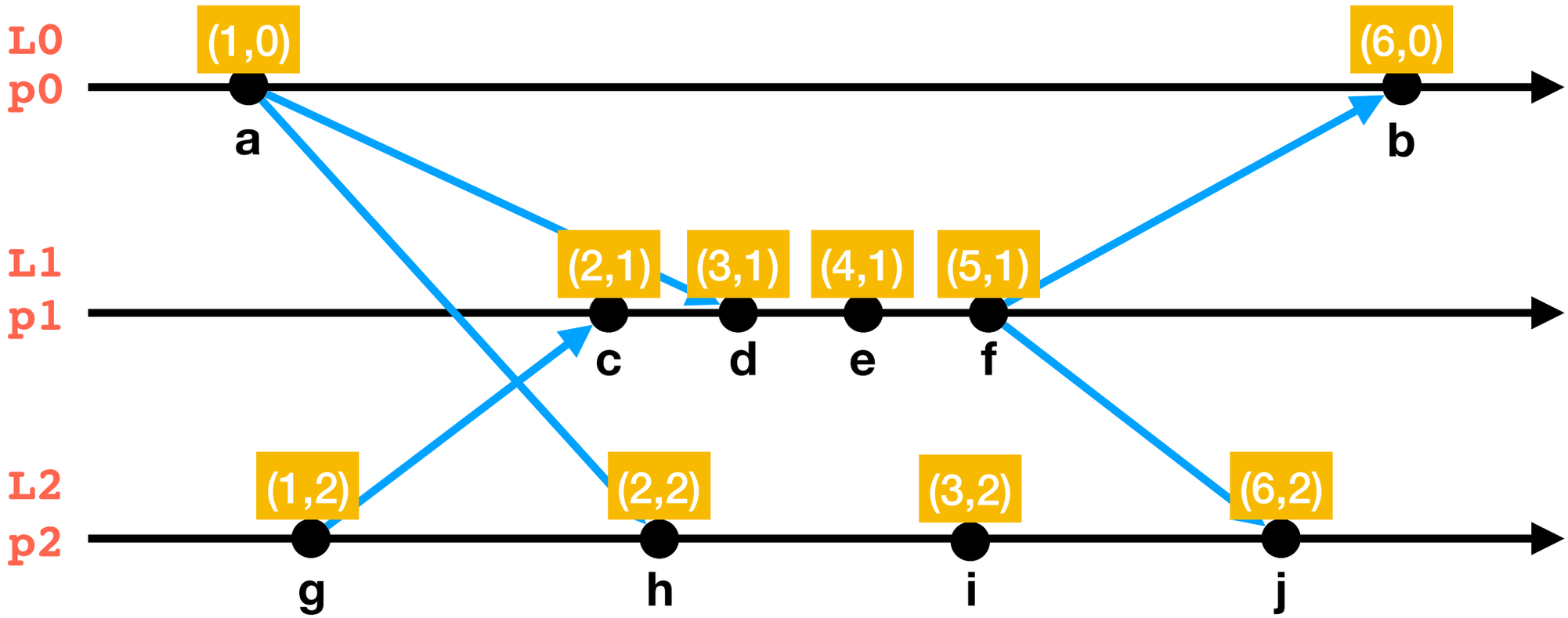
- Les horloges de Lamport ordonnent **partiellement** les évènements
  - pour e || e’, il est possible que L(e) == L(e’)
- Il est souvent souhaitable d’avoir un **ordre total**
- **Solution** : si **e** est un évènement sur p_i, lui associer le **couple** (L(e),i)
- **Ordre total cohérent avec l’ordre causal**
  $(L(e), i) < (L(e'), j) \iff L(e) < L(e')$ **ou** $(L(e) = L(e) \wedge i < j)$

Ordre total



- **Ordre total ?**

Ordre total (L,ID)



- **Ordre total (a,g,c,h,d,i,e,f,b,j)**
- **Cohérent avec l’ordre causal**

Horloge vectorielle

- Horloge de Lamport : on ne peut pas détecter les évènements indépendants. Pour e || e', on peut avoir L(e) = L(e'), L(e) < L(e') ou L(e) > L(e')

- **Objectif** attribuer à chaque événement une date qui capture l'ordre causal :

$$\forall e, e', e \rightarrow e' \iff date(e) < date(e')$$

Horloge vectorielle

- Chaque processus p_i maintient un **vecteur** local $V_i = [c_0, \ldots, c_{n-1}]$
  - (n est le nombre de processus)
- Chaque évènement e est associé à un vecteur $V(e) = [c_0, \ldots, c_{n-1}]$
  - c_j est le nombre d'évènements qui précèdent causalement e sur le processus p_j

Mise à jour des vecteurs

- Initialement, pour chaque processus p_i, $V_i = [0,\ldots,0]$

1. à chaque évènement par p_i, incrémenter $V_i[i]$
   $V_i[i] = V_i[i] + 1$

2. envoi d'un message m incrémenter $V_i[i]$, puis envoi de m et V_i $V_i[i] = V_i[i] + 1$; `send(m,`V_i`)`

3. réception de (m,V) : m.a.j V_i avec max. par composante de V_i et V, incrémenter V_i[i]
   `rcv(m,`V_i`)` $V_i = \max(V_i, V); V_i[i] = V_i[i] + 1$

Opérations sur les vecteurs

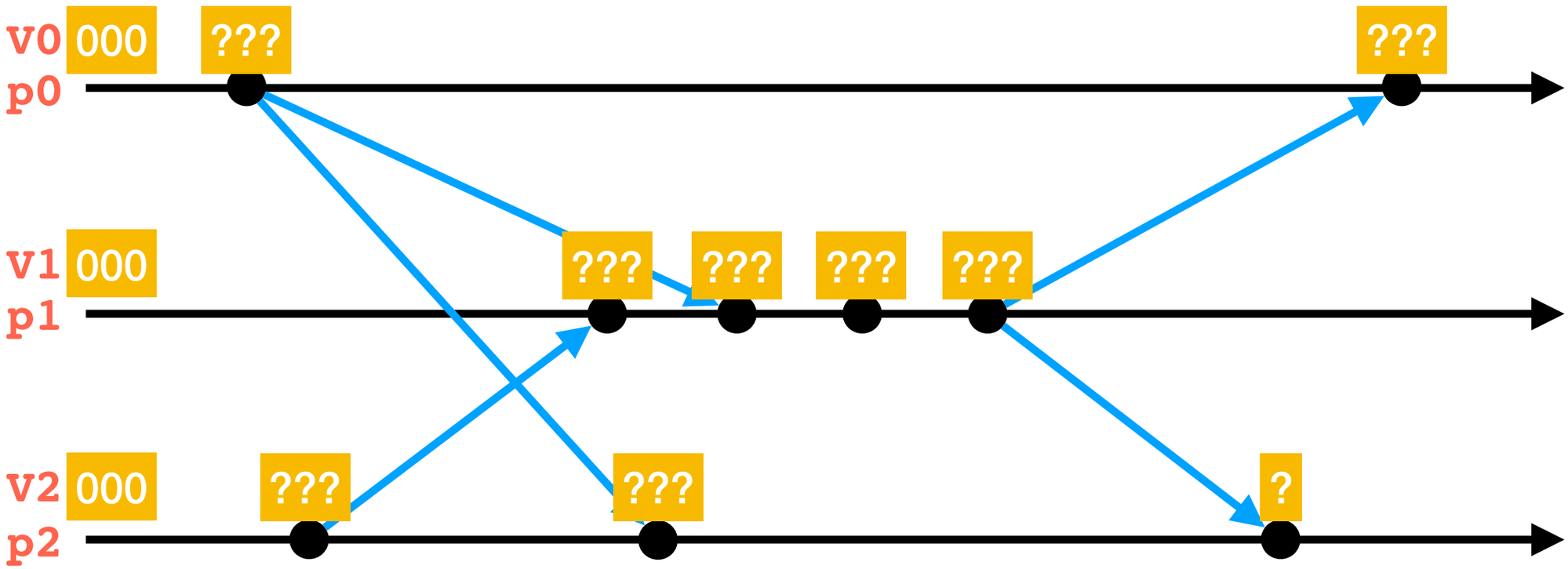
- $U = \max(V, W)$: pour chaque $i, 0 \leq i \leq n-1$:
  $U[i] = \max(V[i], W[i])$

- $U \leq V$ si et seulement si pour chaque $i, 0 \leq i \leq n-1$:
  $U[i] \leq V[i]$

  - $[2,2,3] = \max([1,2,3], [2,1,3])$

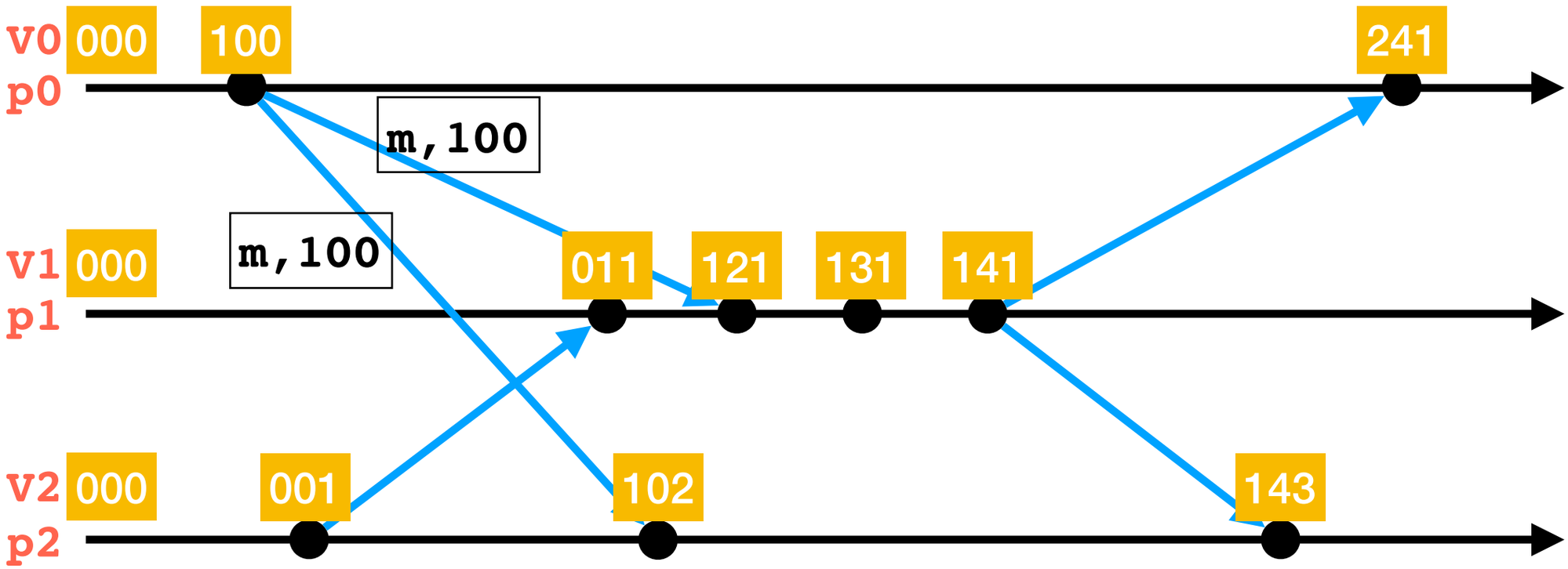
  - $[1,2,3] < [2,3,4]$, $[1,2,3]$ et $[2,1,3]$ sont incomparables

Horloge vectorielle



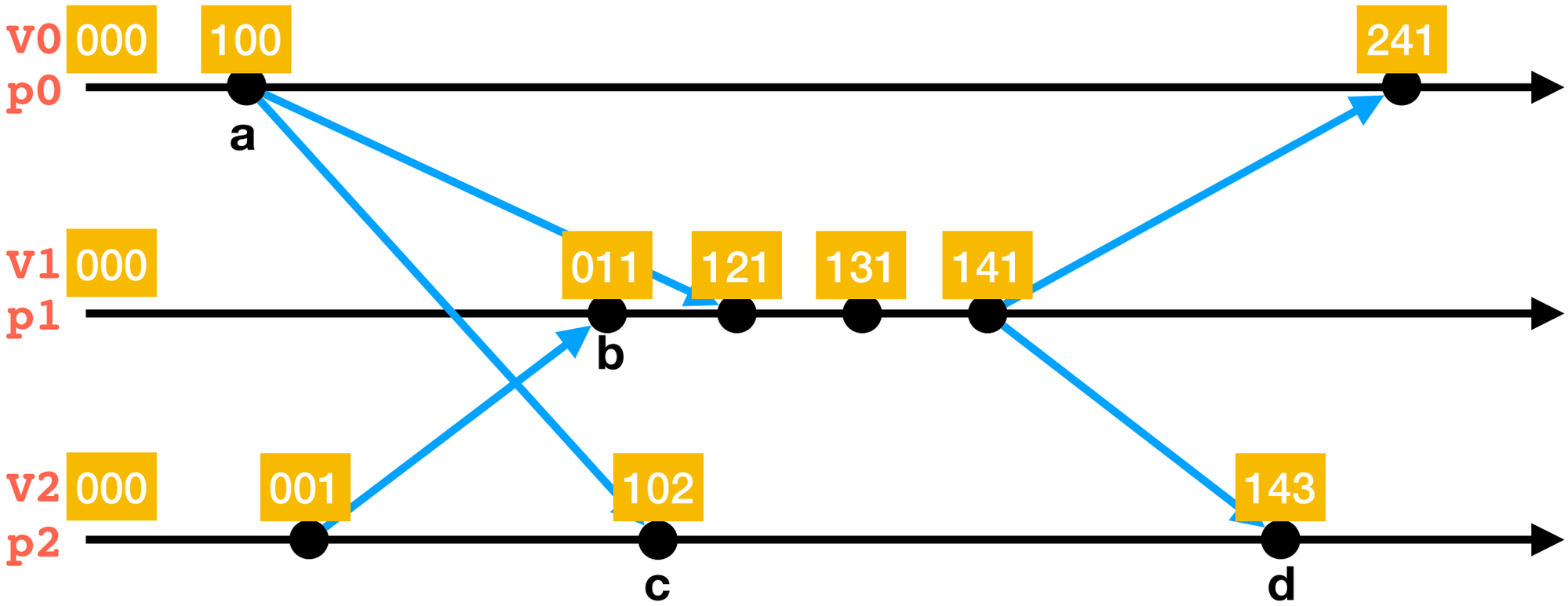
1. à chaque évènement interne par p_i : $V_i[i] = V_i[i] + 1$
2. envoi d'un message m : $V_i[i] = V_i[i] + 1$; `send(m,`V_i`)`
3. réception de (m,V) : `rcv(m,`V_i`)` $V_i = \max(V_i, V); V_i[i] = V_i[i] + 1$

Horloge vectorielle



1. à chaque évènement interne par p_i : $V_i[i] = V_i[i] + 1$
2. envoi d'un message m : $V_i[i] = V_i[i] + 1$; `send(m,`V_i`)`
3. réception de (m,V) : `rcv(m,`V_i`)` $V_i = \max(V_i, V); V_i[i] = V_i[i] + 1$

Horloge vectorielle



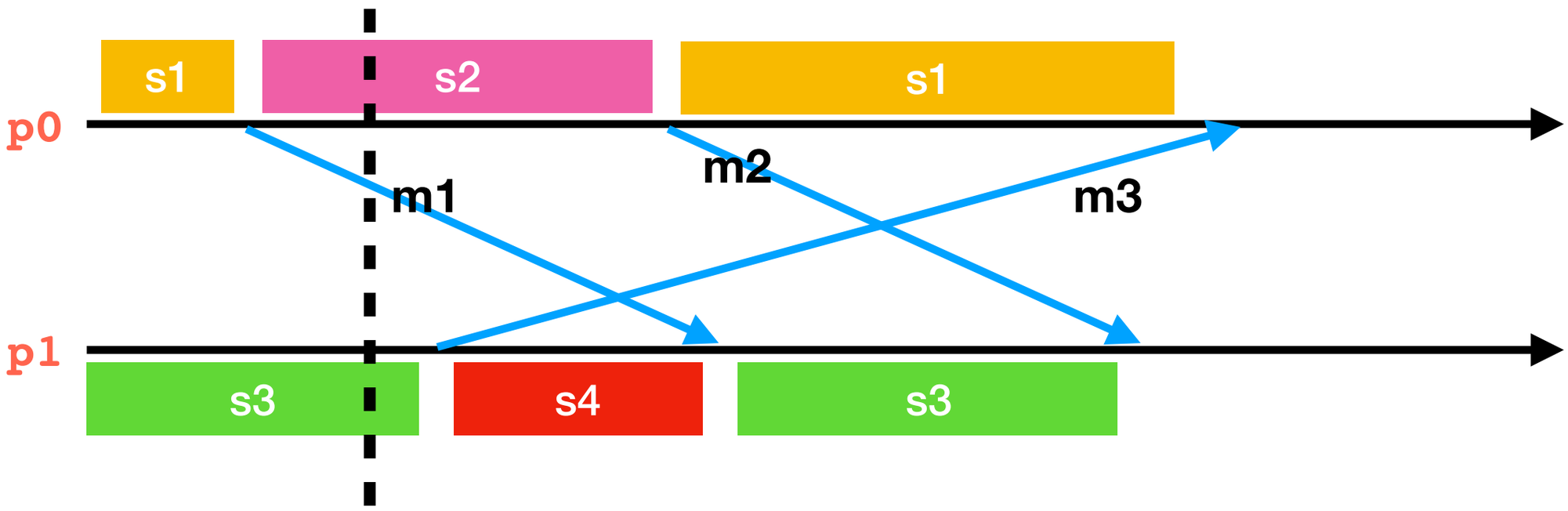
- événements indépendants ont des vecteurs incomparables
  - **b** $\parallel$ **c** : $V(b) \not< V(c)$ et $V(c) \not< V(b)$
- événements causalement ordonnés ont des vecteurs ordonnés
  - $a \rightarrow d : V(a) < V(d)$

Etat global

- **Etat global :** états locaux des processus + états des canaux (messages en transit)

- **Cohérent :** ont pu se produire ensemble à un moment de l’exécution

état global



- **état global**
  - état p0 : s2
  - état p1 : s3
  - canal c01 : m1
  - canal c10 : vide

Application

- Point de reprise après panne (checkpointing)
- Garbage collection
- Débogage
- Détection de terminaison
- Vérification de propriétés globales (absence de deadlock)

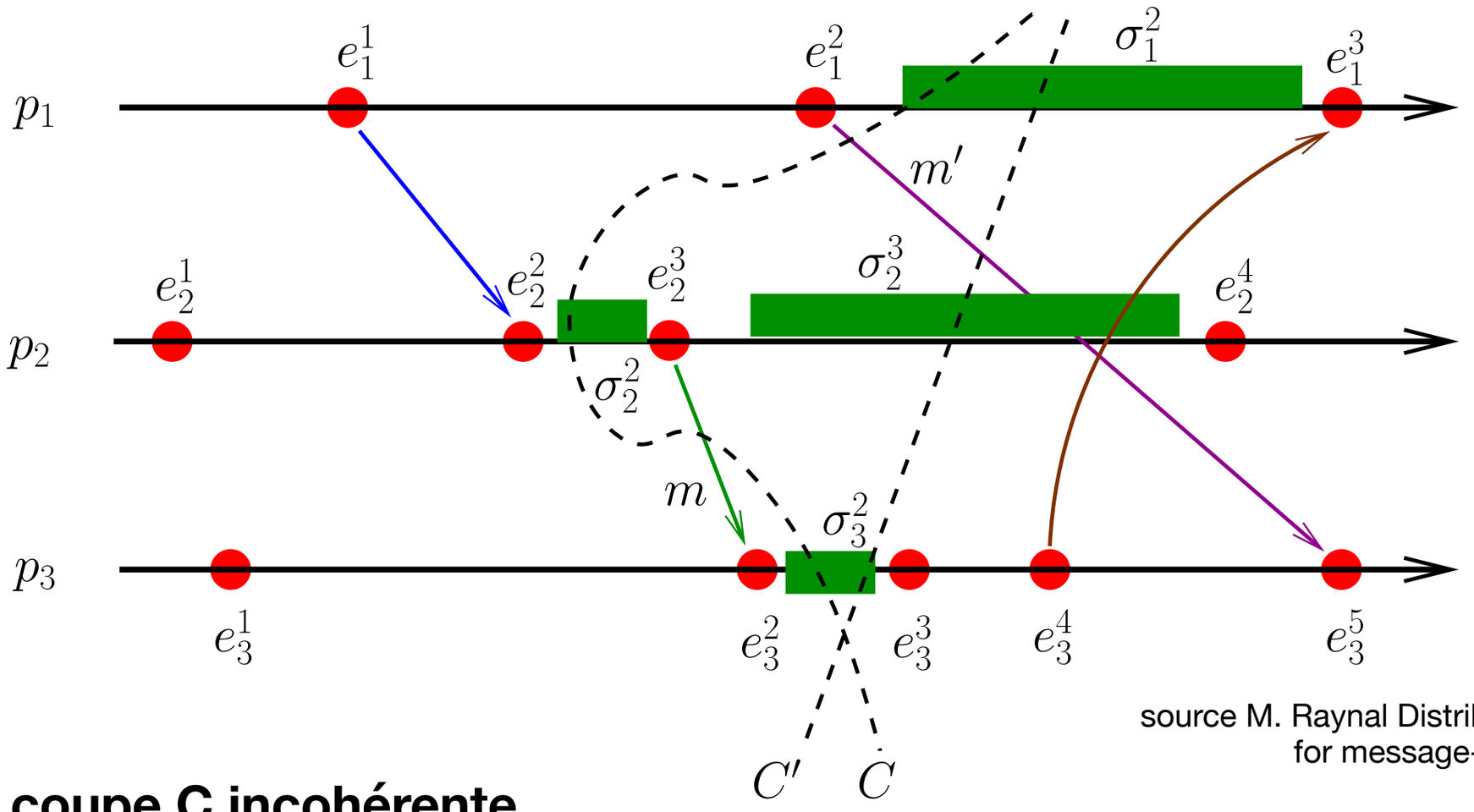
Coupe cohérente

- Coupe : préfixes des événements de processus
- Coupe C cohérente si et seulement si :

$$\forall e, e' : e' \in C \wedge e \rightarrow e' \implies e' \in C$$

- I.e., pour chaque évènement e dans C, le passé causal de e est aussi dans C

Coupes (in)cohérentes



source M. Raynal Distributed Algorithms for message-passing systems

- **coupe C incohérente**
- **coupe C’ cohérente**

Modèle

- **n** processus
- **canaux FIFO** first-in first-out : messages de p à q sont reçus dans l’ordre de leur envoi
- **asynchrone** ni perte, ni altération des messages mais délai non bornés

Algorithme de Chandy-Lamport

La machine initiatrice :

- sauvegarde son état
- envoie un jeton à tous les autres
- sauvegarde les messages reçus après l'envoi du jeton

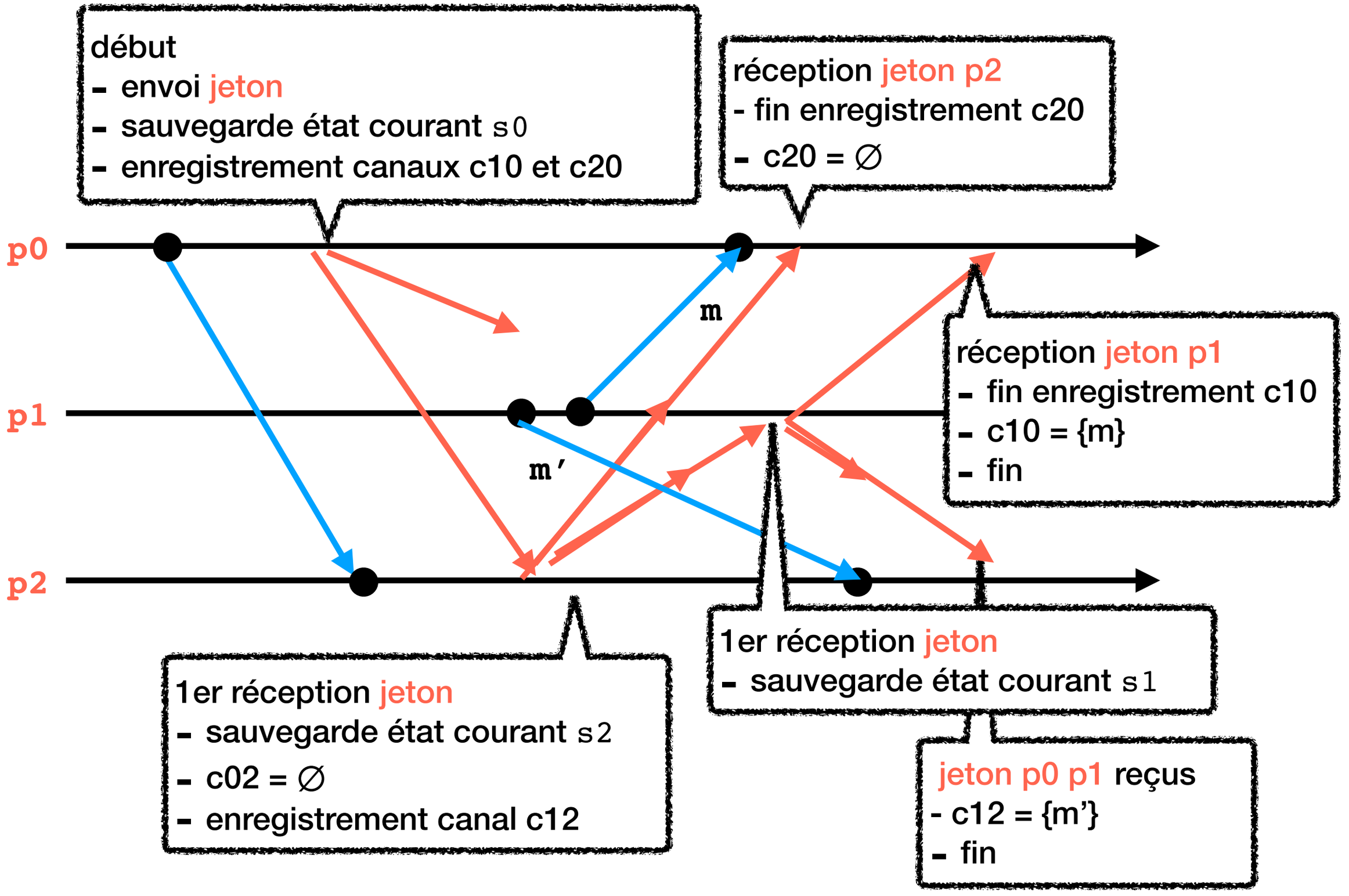
Quand une machine reçoit un jeton pour la première fois :

- sauvegarde son état
- envoie un jeton à tous les autres
- sauvegarde les messages reçus après l'envoi du jeton sauf ceux venant de machines dont elle a déjà reçu un jeton

Quand une machine a reçu un jeton de tous les membres :

- Elle envoie son état et ses messages sauvegardés à l'initiateur

$\Rightarrow$ État global cohérent !
(en incluant les messages sauvegardés)


début
- envoi jeton
- sauvegarde état courant s0
- enregistrement canaux c10 et c20
réception jeton p2
- fin enregistrement c20
- c20 = ∅
p0
p1
p2
m
m'
réception jeton p1
- fin enregistrement c10
- c10 = {m}
- fin
1er réception jeton
- sauvegarde état courant s2
- c02 = ∅
- enregistrement canal c12
1er réception jeton
- sauvegarde état courant s1
jeton p0 p1 reçus
- c12 = {m'}
- fin

début
- envoi jeton
- sauvegarde état courant s0
- enregistrement canaux c10 et c20

réception jeton p2
- fin enregistrement c20
- c20 = ∅

p0

s0

m

réception jeton p1
- fin enregistrement c10
- c10 = {m}
- fin

p1

s1

m'

p2

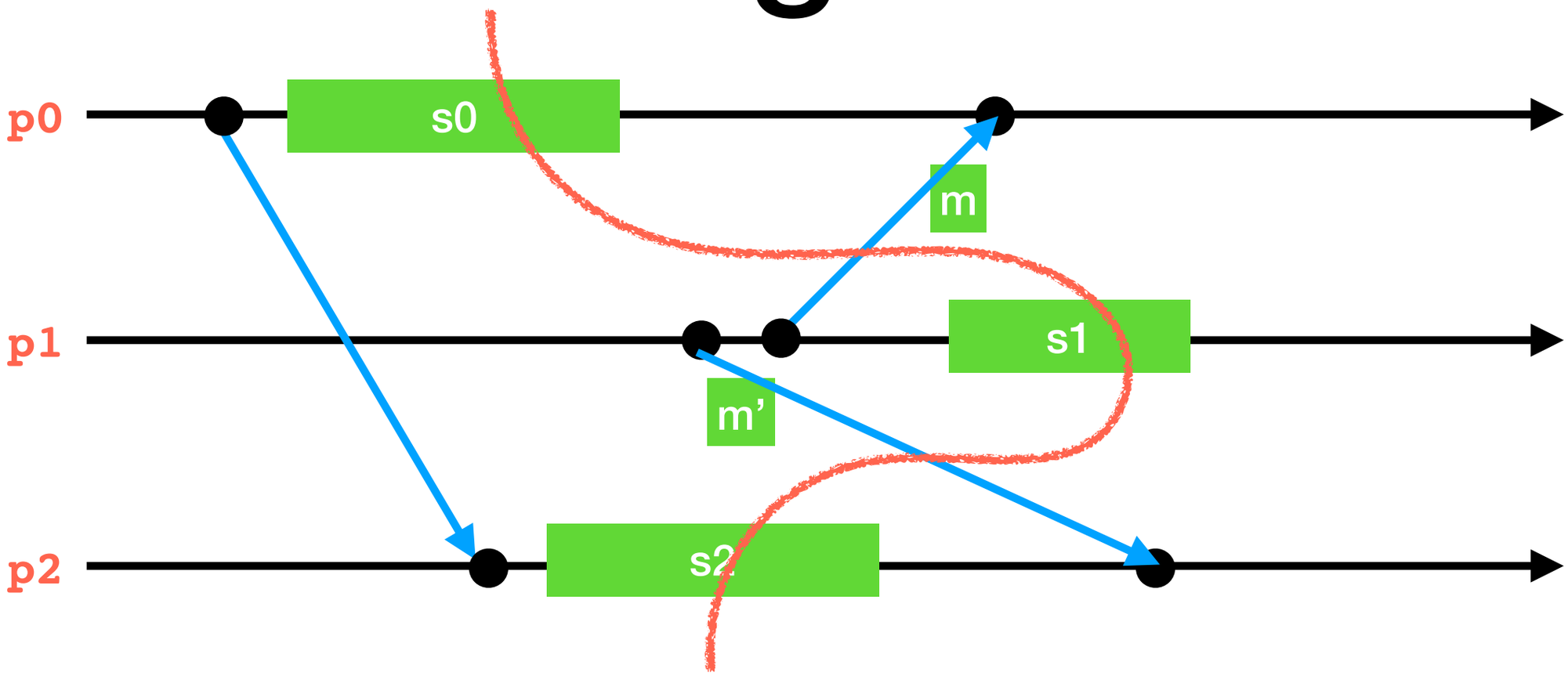
s2

1er réception jeton
- sauvegarde état courant s2
- c02 = ∅
- enregistrement canal c12

1er réception jeton
- sauvegarde état courant s1

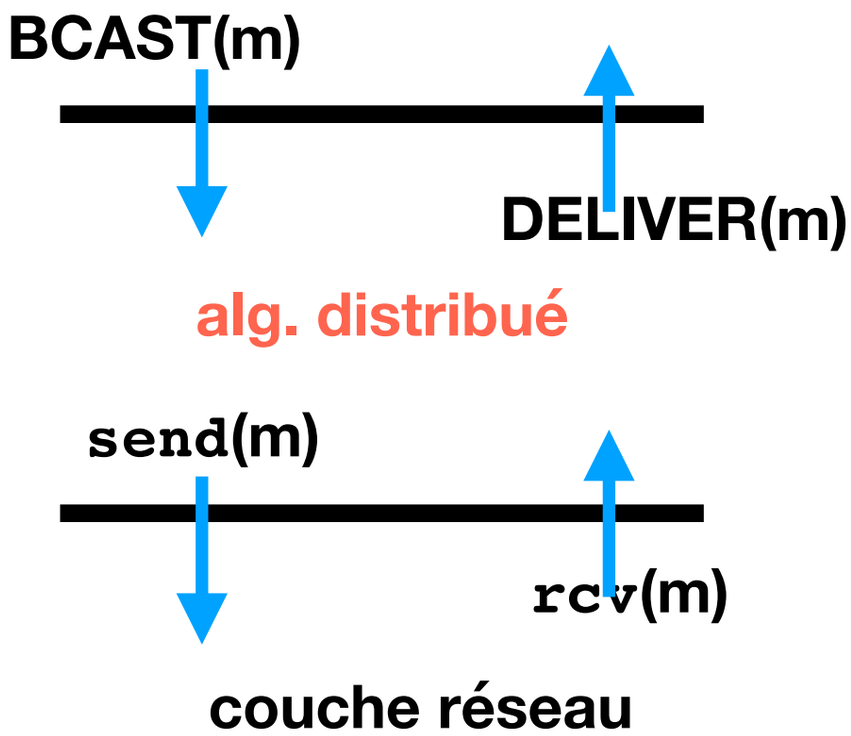
jeton p0 p1 reçus
- c12 = {m'}
- fin

état global



- états processus **p0** : **s0, p1 : s1, p2 : s2**
- états canaux : **c01:vide, c02: vide, c10: {m} , c12: {m'}, c20: vide, c21:vide**
- **-> coupe cohérente**

Diffusion (broadcast)



Total Order Broadcast

- Tout message diffusé (**BCAST(m)**) doit être livré (**DELIVER(m)**) par tous les processus

- Pour un processus et un message **m**, **m** est livré au plus une fois

- Les messages sont livrés dans **le même ordre** sur chaque processus

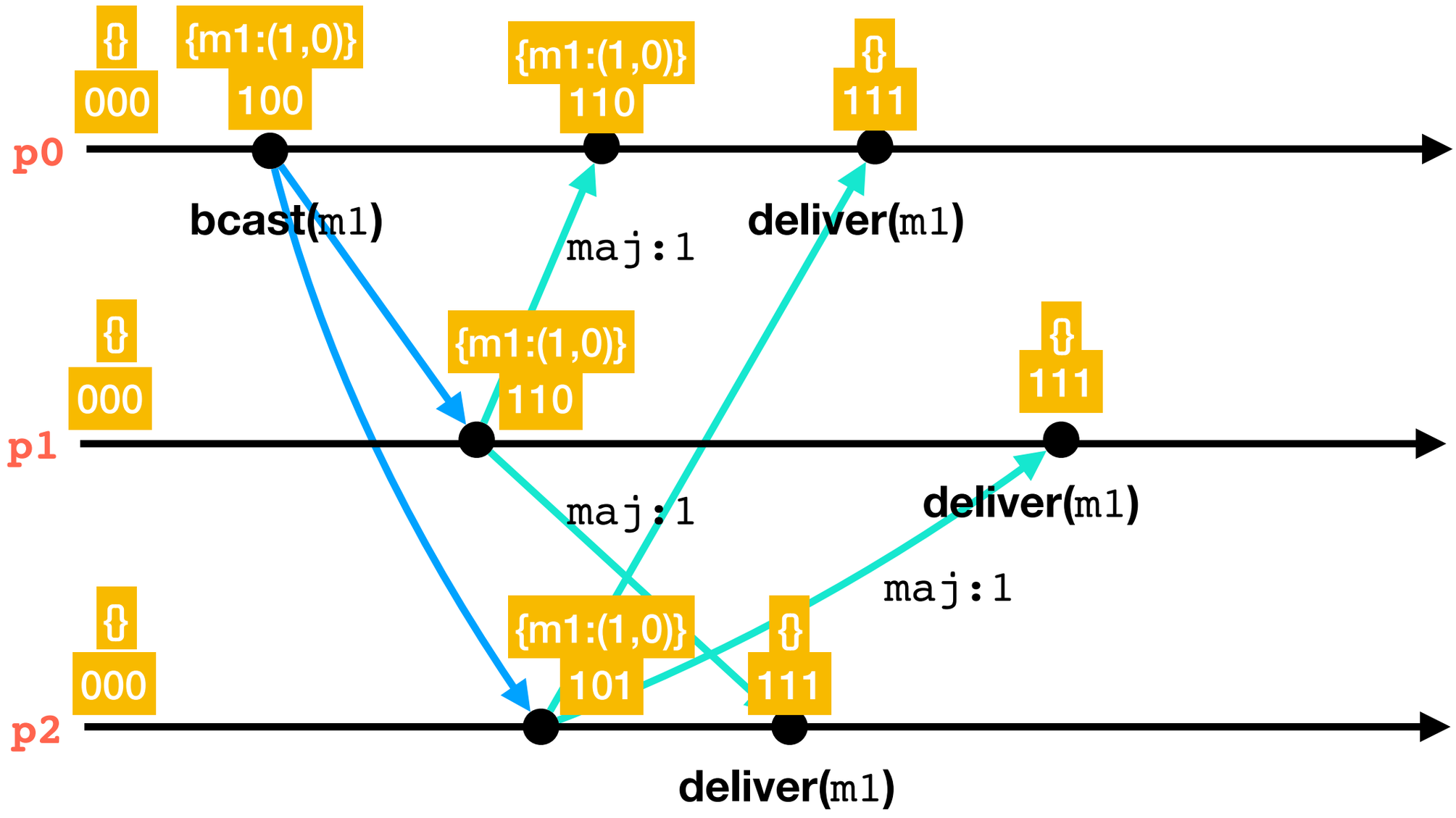
Modèle

- canaux asynchrones mais fiables
- chaque canal est FIFO
- n processus

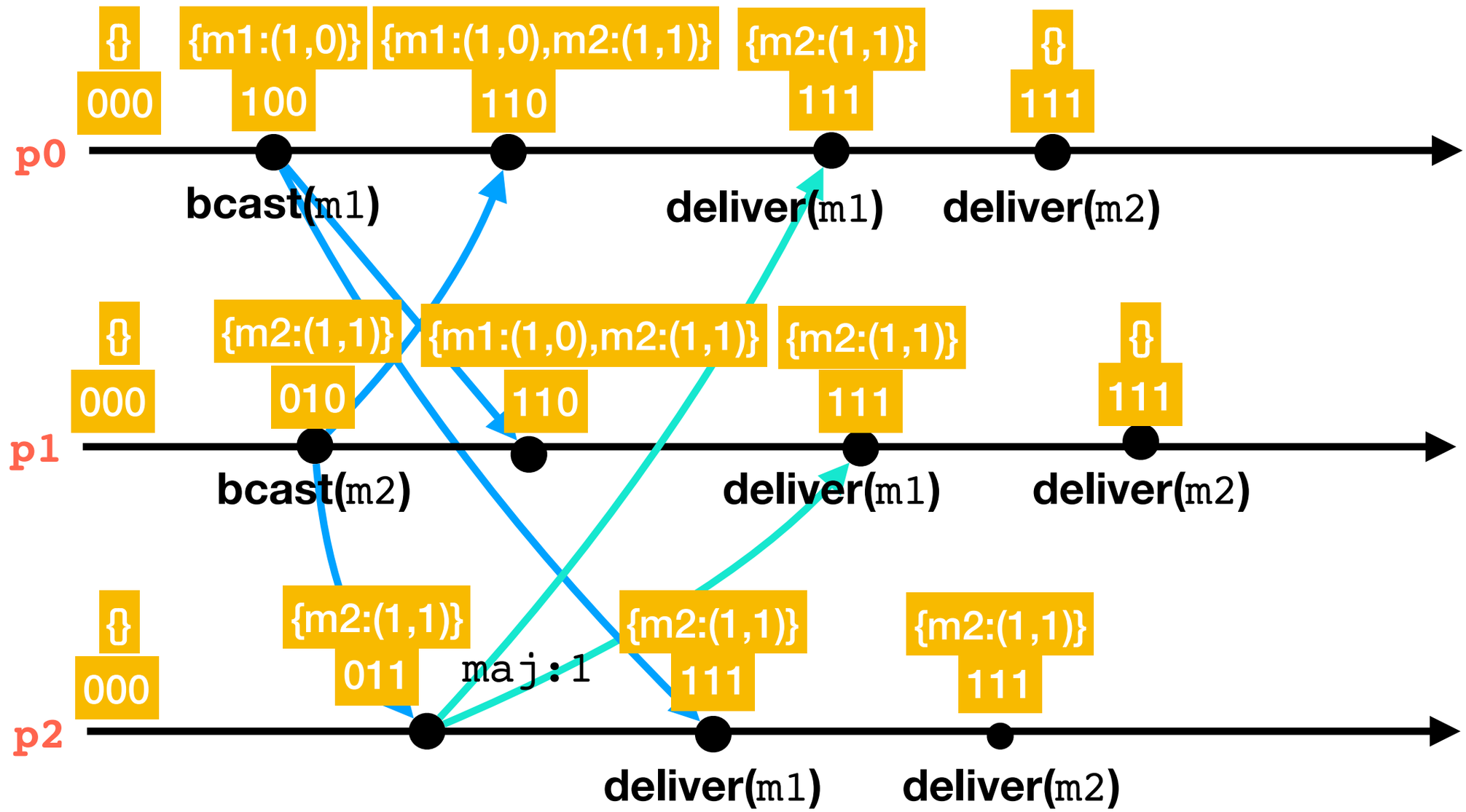
Algorithme

- **Variables locales** chaque processus p_i : V[0..n-1] et attente
  - initialement V = [0,…,0] et attente = { }
- **BCAST(m) :**
  - V[i] ++; ajouter (m,V[i],i) à attente ; `send`(m,V[i],i) à chaque processus $p_j, j \neq i$
- **réception de** (m,v,j) :
  - V[j] = v; ajouter (m,v,j) à attente.
  - si v>V[i] alors V[i] = v; **`send`**`(MAJ,`V[i],i) à chaque processus $p_j, j \neq i$
- **réception de** (`MAJ`,v,j) :V[j] = v
- **DELIVER(m,j)** quand :
  - (m,v,j) a le plus petit couple (v,j) dans attente ET $v \leq V[k]$ pour tout k
  - enlever (m,v,j) de attente

Exemple


{}
000
{m1:(1,0)}
100
{m1:(1,0)}
110
{}
111
p0
bcast(m1)
maj:1
deliver(m1)
{}
000
{m1:(1,0)}
110
{}
111
p1
maj:1
deliver(m1)
maj:1
{}
000
{m1:(1,0)}
101
{}
111
p2
deliver(m1)

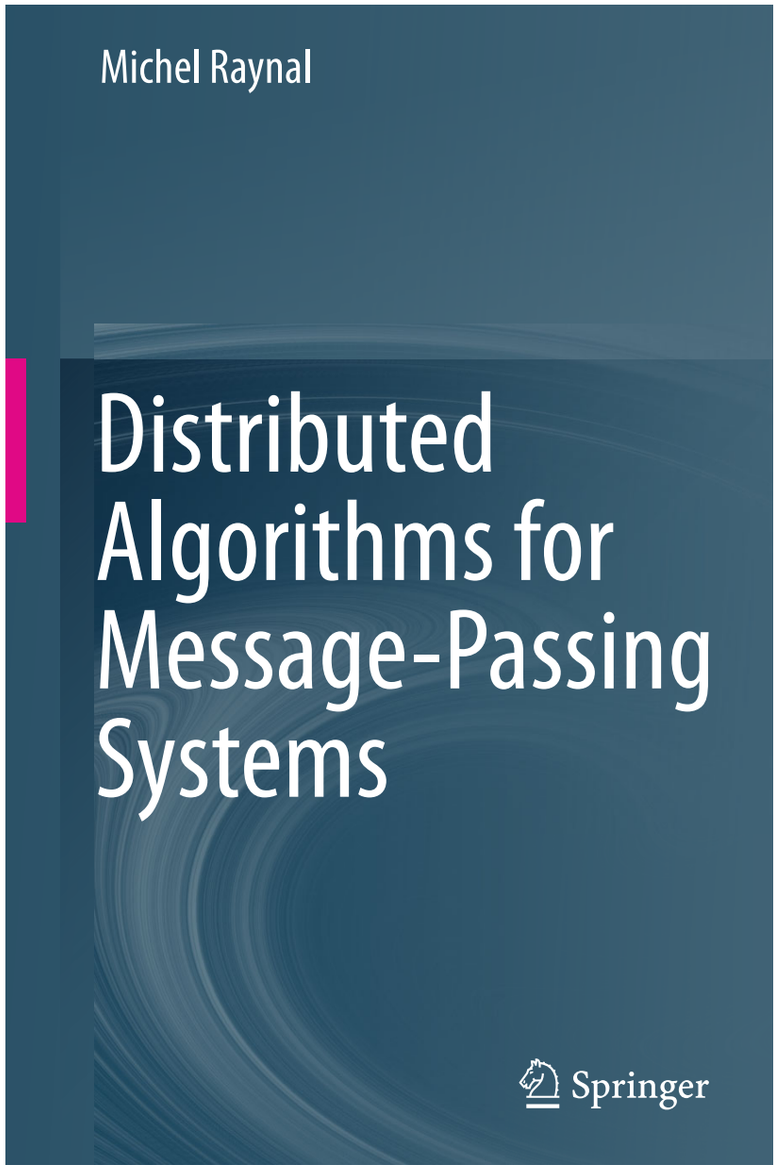
ordre total


{}
000
{m1:(1,0)}
100
{m1:(1,0),m2:(1,1)}
110
{m2:(1,1)}
111
{}
111
p0
bcast(m1)
deliver(m1)
deliver(m2)
{}
000
{m2:(1,1)}
010
{m1:(1,0),m2:(1,1)}
110
{m2:(1,1)}
111
{}
111
p1
bcast(m2)
deliver(m1)
deliver(m2)
{}
000
{m2:(1,1)}
011
maj:1
{m2:(1,1)}
111
{m2:(1,1)}
111
p2
deliver(m1)
deliver(m2)

Total order broadcast

- Ordre causal + ordre total
  - ordre causal : si **deliver(m)** -> **bcast(m')** alors deliver(m) -> deliver(m') pour tous les processus
- Canaux asynchrone, mais fiable et FIFO
- Non-tolérant aux pannes

Référence


Michel Raynal
Distributed Algorithms for Message-Passing Systems
Springer