# Cours ASPD
# Algorithmes répartis: auto-stabilisation et nommage
## Telecom Nancy 2A Apprentissage

Dominique Méry
Telecom Nancy
Université de Lorraine

**Année universitaire 2024-2025**
**18 mars 2025**

# Summary

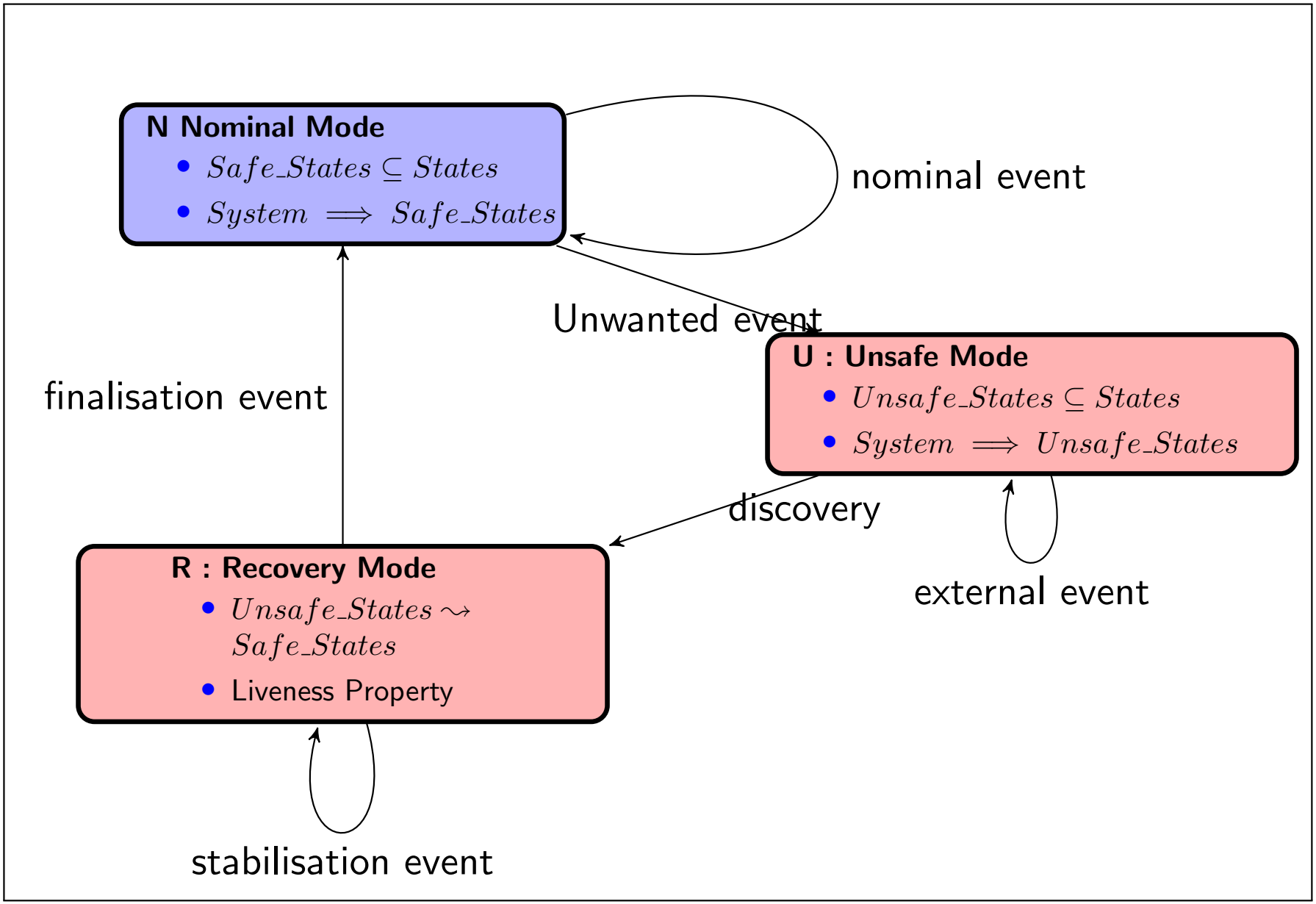
# Section Courante



]

# Etat courant

# Auto-stabilisation dans les systèmes répartis



# Stabilisation

- Les systèmes peuvent être victimes de fautes transitoires
- Un certain nombre d'algorithmes permettent d'anticiper de tels problèmes.
- Les algorithmes auto-stabilisants constituent une alternative à la perte de l'état consistant d'un système donné à la suite d'une défaillance.
- Différents phénomènes peuvent apparaître lors de l'exécution d'un algorithme : en particulier peuvent se produire des changements dans le réseau (ajout ou disparition d'un sommet ou d'un lien) ou bien encore des altérations de messages ou de mémoires.
- Un algorithme est dit auto-stabilisant s'il se termine correctement en dépit de l'apparition de ces phénomènes.
- Un algorithme auto-stabilisant ne nécessite pas d'initialisation particulière.

# Contexte

- Les systèmes répartis sont exposés à des risques de défaillances transitoires qui peuvent placer un système donné dans un état ou une configuration arbitraire.
- Cette configuration arbitraire peut être une configuration ne satisfaisant plus l'invariant du système.
- Dans ce cas, la question est de concevoir un algorithme réparti permettant de faire converger le système global d'une configuration arbitraire vers une configuration dite stable et satisfaisant l'invariant.

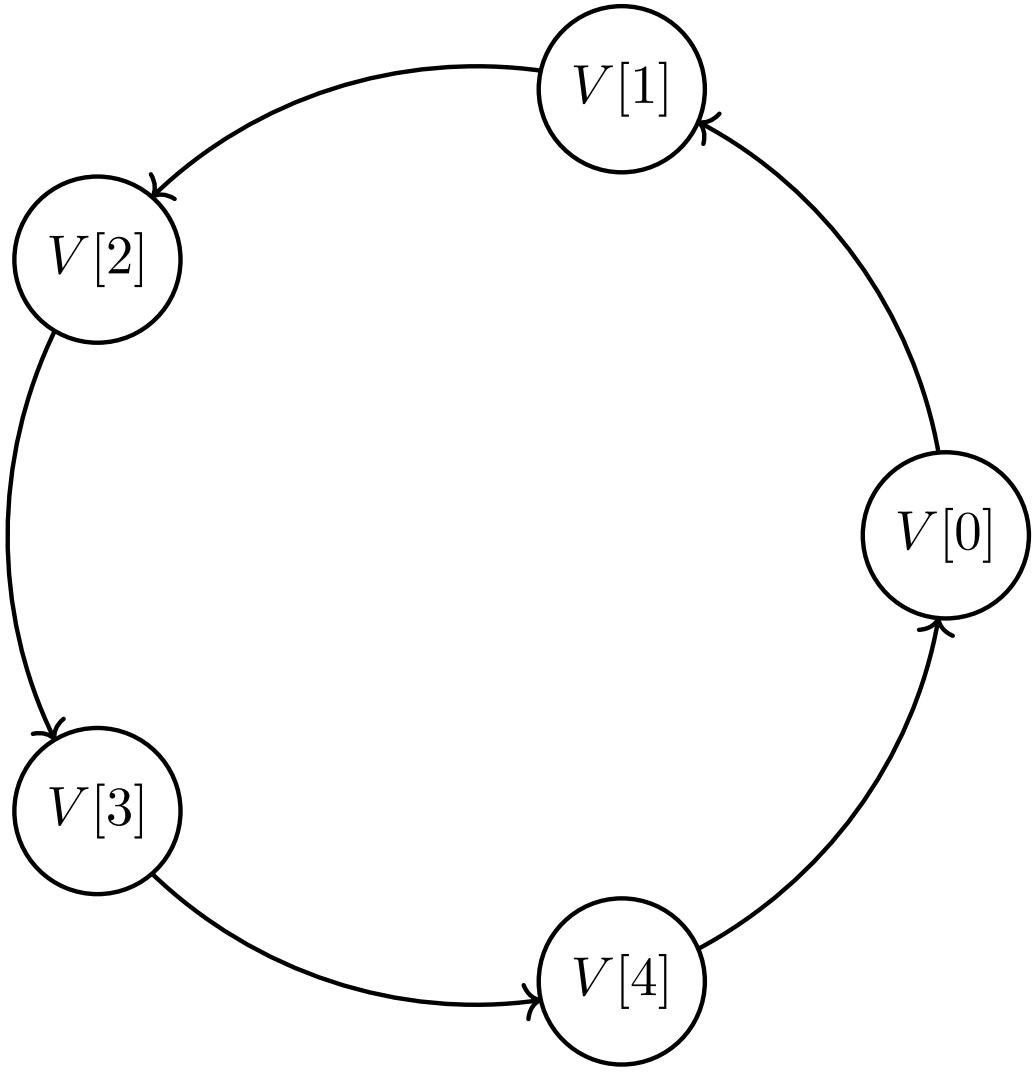
# Etat courant

# Modélisation d'un système auto-stabilisant

- Soit un système réparti $\mathcal{S}$ modélisé par un système de transition $(States, \longrightarrow)$ où $States$ est un ensemble d'états ou de configurations et où $\longrightarrow$ modélise la relation de transition simulant l'activité du système.
- Une exécution ou un calcul de $\mathcal{S}$ est une suite maximale $(\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_i, \ldots)$ engendrée par $(States, \longrightarrow)$.
- Tout préfixe d'une telle suite est un calcul puisqu'il n'y a pas d'états initiaux déterminés.
- Tout préfixe d'exécution est donc un calcul ou une exécution de $\mathcal{S}$ (pas de dépendance d'un état initial)

# Etat courant

# Réseau en anneau



# Etat courant

# Problème de l'exclusion mutuelle

## Description

- Dans toute configuration, au plus un processus a le privilège.
- Chaque processus a le privilège infiniment souvent.

- Dans un réseau avec une topologie en anneau, des processus ont accès à une ressource.
- Le privilège est accordé à un processus à la fois.
- Le problème est que le privilège est géré par un jeton qui peut être perdu et donc être dans une configuration ne satisfaisant plus la propriété.

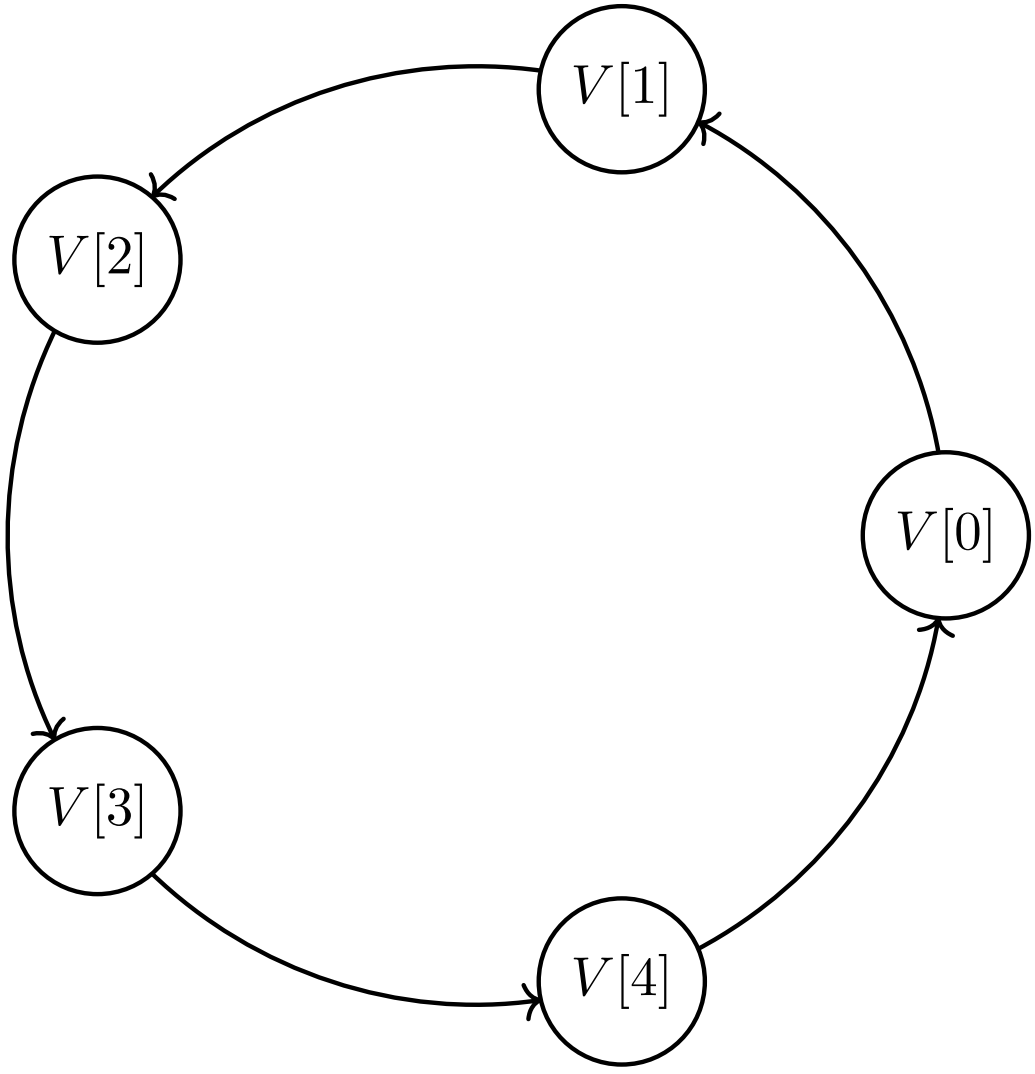
# Algorithme de Dijkstra

$$\text{DOMAINE} \triangleq 0..N$$
$$\text{IMAGE} \triangleq 0..M$$

$$\text{NToZero} \triangleq$$
$$\quad \land \quad V[0] = V[N]$$
$$\quad \land \quad V' = [V \ \textbf{EXCEPT} \ ![0] = (V[0]+1)\%(M+1)]$$

$$\text{Others(I)} \triangleq$$
$$\quad \land \quad I \in 0..N$$
$$\quad \land \quad I \neq N$$
$$\quad \land \quad V[I+1] \neq V[I]$$
$$\quad \land \quad V' = [V \ \textbf{EXCEPT} \ ![I+1] = V[I]]$$

# Réseau en anneau



# Propriétés de stabilisation

```
Init == V = [i \in 0..N |-> (IF i # N THEN i  ELSE 0)]
Next == NToZero \/ (\E I \in 0..N-1:Others(I))

Prop1 ==    (V[0] = V[N])
              => (\A i \in 1..N-1: V[i+1] = V[i])

Prop2 ==
        \/   (\E i,j  \in 0..N-1: i # j
               /\ V[i+1] #  V[i] /\ V[j+1] #  V[j])
        \/ (V[0] = V[N] /\
              (\E i  \in 0..N-1:  V[i+1] #  V[i]))
```

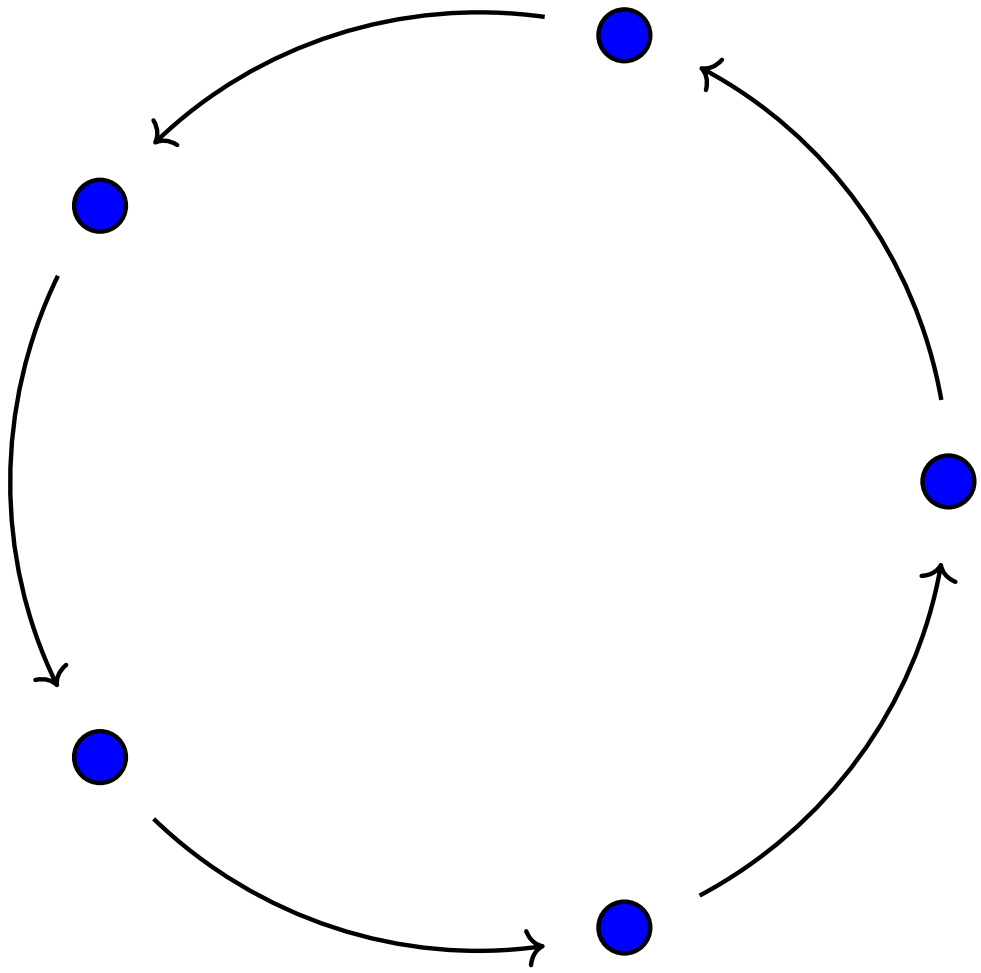
# Etat courant

# Coloriage d'un anneau

Le problème du coloriage d'un anneau concerne l'existence d'une configuration dans laquelle trois nœuds consécutifs n'ont pas la même couleur et pose la question de définir un algorithme réparti permettant de l'atteindre à parrtir de toute configuration intiale de l'anneau.

- Modéliser un processus de coloration d'un anneau à partir d'une situation quelconque initiale pour atteindre une situation stable caractérisée par le fait que deux nœuds voisins n'ont pas la même couleur.
- La règle de changement de couleur : Si un nœud a la même couleur que l'un de ses vosins, il chosit une autre couleur que celle de ses voisins.
- La règle est applicable tant que deux nœuds ont la même couleur.
- La règle est non-applicable dès que les nœuds adjacents ont tous une couleur différente : le cas d'un anneau à un nœud est exclus en considérant qu'il y a au moins deux nœuds dans l'anneau.

# Exemple d'anneau



**CONTEXT** $ring$
**SETS**
$N$
**CONSTANTS**
$n$
**AXIOMS**
$axm1 : n \in N \rightarrowtail\!\!\!\!\!\!\rightarrow N$
$axm2 : \forall s \cdot s \subseteq n[s] \Rightarrow N \subseteq s$
$axm3 : finite(N)$
**END**
**MACHINE** $one{-}shot$
**SEES** $ring, color$
**VARIABLES**
$col$
**INVARIANTS**
$inv1 : col \in N \rightarrow color$

**EVENTS**
EVENT INITIALISATION
$act1 : col :\in N \rightarrow color$
EVENT stable
**ANY**
$f$
**WHERE**
$grd1 : f \in N \rightarrow color$
$grd2 : \forall x \cdot f(x) \notin f[(n \cup n^{-1})[\{x\}]]$
**THEN**
$act1 : col := f$
**END**

Le modèle définit la solution à trouver par un événement abstrait exprimant le lien entre la configuration initiale et une configuration stable. Les régles données explique comment le calcul est effectué par les différents processus du réseau. Chaque règle suppose que leur application est possible quand trois nœuds consécutifs vérifient une condition donnée.

Chaque règle constitue un élément de calcul réparti et cet élément est appliqué sur la *boule* centrée sur un nœud donné et tient compte des couleurs des nœpériphériques.

**MACHINE** $rule$ **REFINES** $one{-}shot$
**SEES** $ring, color$
**VARIABLES** $col, c$
**INVARIANTS**
$inv1 : c \in N \to color$
**THEOREMS**
$thm1 :$

$$\left( \begin{array}{l} \forall x \cdot c(x) \neq c(n(x)) \\ \lor \\ \exists x, clr \cdot c(x) = c(n(x)) \land clr \neq c(x) \land clr \neq c(n(x))) \end{array} \right)$$

**EVENTS**

EVENT INITIALISATION

$$act1 : col, c : | \left( \begin{array}{l} col' \in N \to color \\ \wedge \\ c' \in N \to color \\ \wedge \\ col' = c' \end{array} \right)$$

EVENT stable REFINES stable

**WHEN**

$grd1 : \forall x \cdot c(x) \neq c(n(x))$

**WITNESSES**

$f : f = c$

**THEN**

$act1 : col := c$

**END**

EVENT rule

**STATUS convergent**

**ANY**

$x, clr$

**WHERE**

$grd1 : c(x) = c(n(x)) \vee c(x) = c(n^{-1}(x))$

$grd2 : clr \neq c(n(x))$

$grd3 : clr \neq c(n^{-1}(x))$

**THEN**

$act1 : c(x) := clr$

**END**

La terminaison de ce processus dépend de la condition appelée variant et qui doit décroître par application des règles.

**VARIANT** $card(\{x|c(x) = c(n(x))\})$

Les graphes VISIDIA sont symétriques ; le graphe $g$ est défini à partir du graphe $n$ et la règle $rule$ est raffinée en $visidia{-}rule$

**CONTEXT** $ringgr$
**EXTENDS** $ring1$
**CONSTANTS**
$g$
**AXIOMS**
$axm1 : g = n \cup n^{-1}$
**END**

EVENT INITIALISATION

$act1 : col, c : |(col' \in N \to color \wedge c' \in N \to color \wedge col' = c')$

EVENT stable REFINES stable

**WHEN**

$grd1 : \forall x \cdot c(x) \notin c[g[\{x\}]]$

**THEN**

$act1 : col := c$

**END**

EVENT rule

**STATUS convergent**

REFINES rule

**ANY**

$x, clr$

**WHERE**

$grd1 : c(x) \in c[g[\{x\}]]$

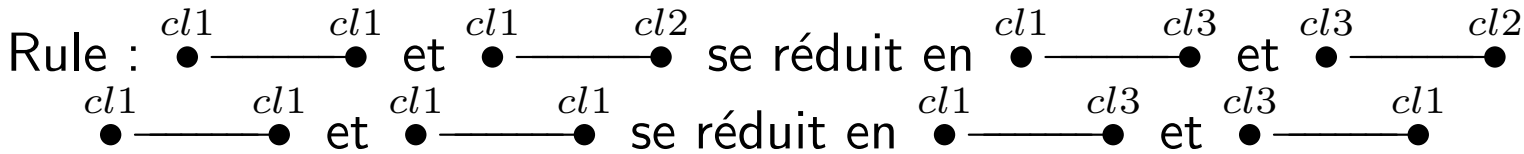
$grd2 : clr \notin c[g[\{x\}]]$

**THEN**

$act1 : c(x) := clr$

**END**

# Dérivation d'un programme VISIDIA



# Sommaire sur l'autostabilisation

- Définition d'algorithmes indépendants des états initiaux
- Algorithmes très complexes à vérifier en particulier *la terminaison*
- Version autostabilisante de beaucoup d'algorithmes

# Terminaison de l'algorithme

- stabilité : un seul processus a la main ou un seul $i \in 0..N \wedge v[i] = v[i +_N 1]$.
- propriété 1 : il y a au moins un processus avec une garde franchissable :
  - tous les processus de 1 à N ont la même valeur (hypothèse)
  - nécessairement 0 et N ont la même valeur et 0 a la main.
- propriété 2 : toute configuration légale satisfait la propriété de clôture :
  - les processus de 0 à i-1 ont la même valeur
  - les processus de i à N ont la même valeur
  - $v[i] \neq v[i-1]$
- propriété 3 : à partir de toute configuration illégale, l'anneau atteint une configuration légale (le nombre de processus ayant la main ne peut pas croître et au mieux il peut stagner)
  - $0 \ldots i-1$ : non franchissable
  - chaque processus maintient le nombre de processus franchissables : on construit à partir de p, une suite w1 ... wN différent deux à deux
  - cette suite ne peut pas rester indéfiniment stable pr propagation de la valeur et respect de la constnace.

# Conclusion

- Algorithmes répartis : problème de l'expression locale de la globalité
- Algorithmes très complexes à vérifier
- Prise en compte de nombreux aspects de l'environnement comme les fautes, les erreurs, ...

# Section Courante



]

# Etat courant

# système réparti

- un programme réparti est constitué d'un ensemble de processus s'exécutant en parallèle et communiquant seulement par envoi de messages via des canaux de communication.
- Il n'y a pas d'horloge globale commune pour ces processus
- Les temps de transfert des messages ne sont pas bornés : sous hypothèse de communication fiable, ces temps sont finis.
- **Conséquence** : pas d'observation *simultanée* globale possible de l'état
- **Problème** : déterminer un état global consistant
- Le snapshot est une copie ou une photographie instantanée de l'état d'un système.

# Exemples

- Calcul de la topologie d'un réseau
- Compter le nombre de processus dans un système réparti
- Détecter la terminaison
- Détecter le blocage
- Détecter la perte de coordination

# Reconstituer l'état global

- Les algorithmes de snapshots permettent de donner des clichés répartis de l'état global
- Comment rassembler ces *clichés répartis* ou *snapshots* afin de donner une vue consistante et globale à chaque processus du système ?
- Proposer des algorithmes rassemblant les *fragments* en un état global.

# Applications de ce problème

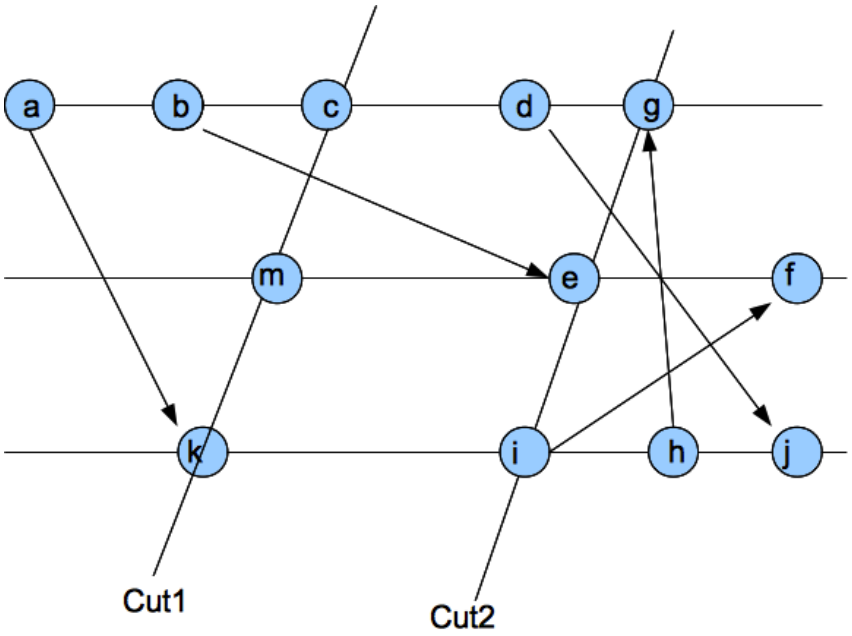
- *détection du blocage* : tout processus bloqué momentanément souhaite connare l'état global du système, afin de savoir s'il a atteint une configuration de blocage.
- *détection de la terminaison* : dans le cas de calculs répartis, les processus exécutent des actions au cours de phases de calculs mais le passage à une phase suivante peut exiger que le processus en cours obtienne la connaisance de l'état des autres processus : nécessité de connaître l'état global.
- *détection de la conservation de jetons dans un réseau*
- *réinitialisation du réseau* : un système réparti peut entrer dans une configuration nécessitant de relancer ou de réinitilaiser l'état global cournat et cette relance peut être faite à partir des snapshots conservés qui vont donc constituer des points de reprise (*checkpoints*)

# Etat courant

# Modélisation d'un système réparti

- Un système réparti est modélisé comme un ensemble fini de processus séquentiels
- Chaque processus $i$ est caractérisé par un état local noté $LS_i$
- Chaque canal de communication $C_{ij}$ a un état noté $SC_{ij}$.
- Chaque processus séquentiel peut réaliser trois types d'actions :
  - Une action interne modifiant l'état local
  - Une envoi sur un canal
  - Une réception sur un canal

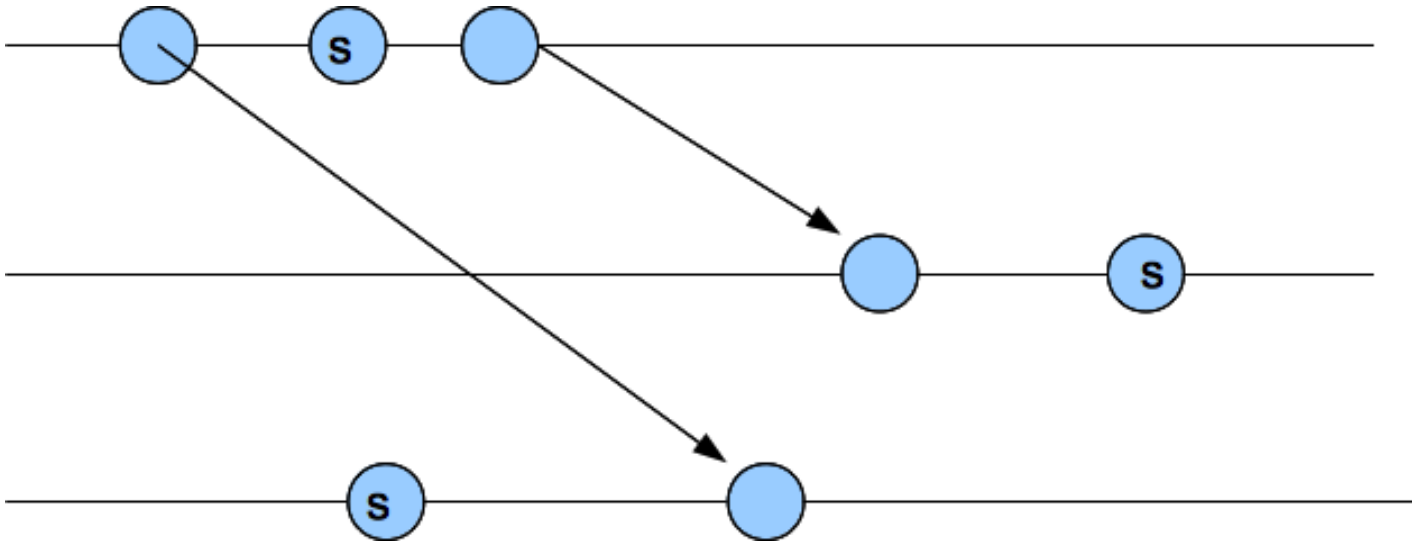
# Coupures



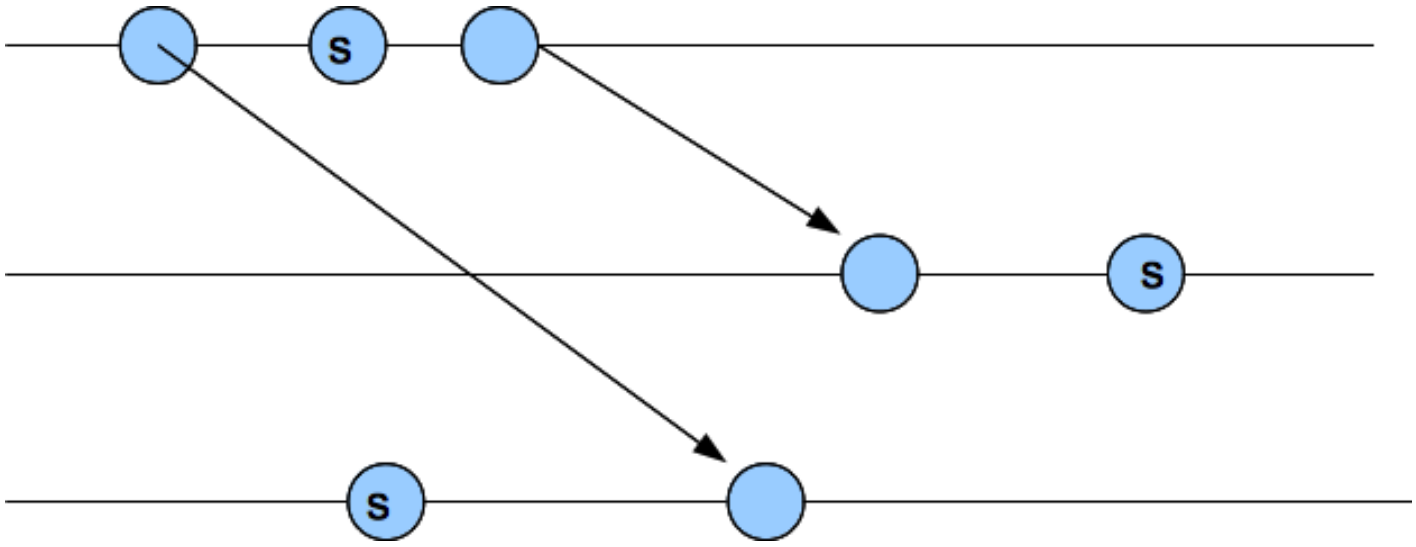

- $Cut1 = \{a, b, c, m, k\}$
- $Cut2 = \{a, b, c, d, g, m, e, k, i\}$

- Une coupure $C$ est un ensemble d'événements
- Une coupure consistante $C$ est une coupure satisfaisant la propriétét suivante : $\forall a, b. a \in C \wedge b < a \Rightarrow b \in C$
- $Cut1$ est consistante mais $Cut2$ n'est pas consistante ($h < g$ mais $h \notin Cut2$).

# Cas pathologique

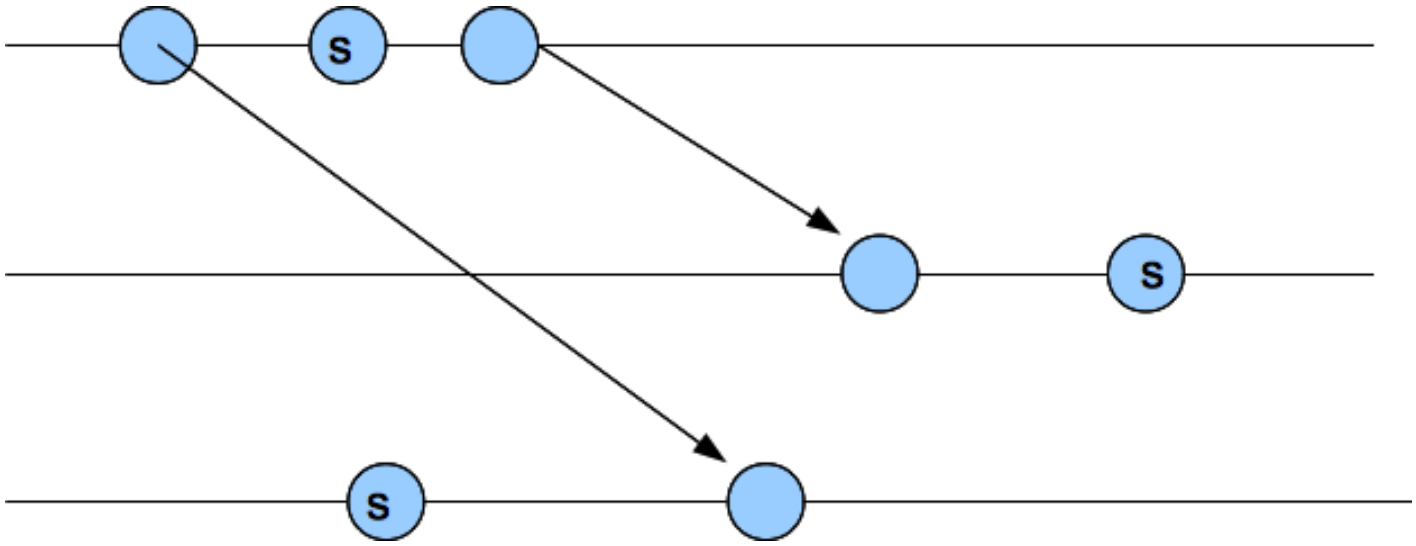


# Cas pathologique




- Un message est reçu parce qu'il a été envoyé

# Cas pathologique




- Un message est reçu parce qu'il a été envoyé
- *Pour tout canal de communication, tout message reçu a été envoyé*

# Des coupures aux snapshots

- Deux coupures peuvent être équivalentes dans la mesure où l'échange de deux actions indépendantes c'est-à-dire causalement indépendantes.
- Pour une coupure donnée, on peut définir l'ensemble des événements les plus récents : ces événements constituent une frontière du calcul en cours.
- Une coupure induit donc un snapshot : une photographie du système et de son état global.
- Toute coupure consistante induit un snapshot consistant et le snapshot courant est l'ensemble des états locaux suivants les événements récemment enregistrés.
- Si $C1$ et $C2$ sont deux coupures telles que $C1 \subseteq C2$, alors $S2$ est plus récent que $S1$

# Calculs répartis

- Une exécution (ou un calcul) $C$ est un ordre total des événements
- Une exécution $C$ est *consistante*, si elle est construite en satisfaisant la condition suivante :

  Pour toute action a et b de $C$, si $a < b$, alors $a\ precedes\ b$.
- Deux exécutions sont équivalentes modulu la relation d'indépendance : deux actions a et b sont indépendantes si ni $a\ precedes\ b$ et ni $b\ precedes\ a$
- Les algorithmes vont construire des snapshots consistants mais vont faire des choix selon les techniques utilisées.

# Etat courant

# Hypothèses de l'algorithme

- $P$ est un ensemble de processus connectés via un ensemble de canaux $C$
- Pas d'horloge commune ni de mémoire partagée
- Toute processus peut enregistrer son état local, envoyer des messages ou recevoir des messages
- Les canaux sont
  - unidirectionnels
  - gérés selon la politique FIFO
  - de capacité infinie
  - sans erreur
  - équitables
- *Construire un snapshot consistant*

# Principe de cet algorithme

- On se donne deux ensembles : $P$ est l'ensemble des processus et $M$ est l'ensemble des messages
- On définit un message particulier appelé marqueur et noté ✠
- Un ensemble de processus $P$ connectés par des canaux caractérisés par deux variables
  - $sent \in P \times P \pfun (\mathbb{N} \pfun M)$
  - $rec \in P \times P \pfun (\mathbb{N} \pfun M)$
  - $dom(sent) = dom(rec)$
- On se donne un initiateur du calcul $i \in P$.
- On se donne une variable $s$ qui contient tous les processus au début de l'algorithme
- Deux actions sont possibles :
  - *CL1* : le processus $i$ réalise les actions suivantes en un pas de calcul, si $i \in s$ : soustraire $i$ de $s$ ; enregistrer l'état local ; envoyer le marqueur ✠ à tous ses canaux.
  - *CL2* : le processus $j \in P$ reçoit le marqueur ✠ sur un canal incident et réalise les actions suivantes : soustraire $j$ de $s$ ; enregistrer l'état local ; envoyer le marqueur ✠ à tous ses canaux.

# Algorithme de Chandy et Lamport CL

## Initialisation de CL

Initialisation du processus de calcul par un processus p tel que $taken_p = false$.

**BEGIN**
**record(state(p))**
$taken_p := true$
**send** $m$ **to** $Neighbors(p)$
**For each port i of p, record messages arriving on i**
**END**

# Algorithme de Chandy et Lamport CL

## Initialisation de CL

Initialisation du processus de calcul par un processus p tel que $taken_p = false$.

**BEGIN**
**record(state(p))**
$taken_p := true$
**send** $m$ **to** $Neighbors(p)$
**For each port i of p, record messages arriving on i**
**END**

- Applicable par n'importe quel nœud qui souhaite commencer

# Algorithme de Chandy et Lamport CL

## Initialisation de CL

Initialisation du processus de calcul par un processus p tel que $taken_p = false$.

**BEGIN**
**record(state(p))**
$taken_p := true$
**send** $m$ **to** $Neighbors(p)$
**For each port i of p, record messages arriving on i**
**END**

- Applicable par n'importe quel nœud qui souhaite commencer
- Action effectuée localement

# Algorithme de Chandy et Lamport

## R-CL

Une action est activée si un marqueur m arrive sur un port j de p.

**BEGIN**

**receive** $m$ **on** $j$ **of** $p$

**mark**$(port\ j)$

**IF** $not\ taken_p$ **THEN**

$taken_p := true$

**record**(**state**(**p**))

**send** $m$ **to** $Neighbors(p)$

**For each port** $i \neq j$ **of p, record messages arriving on i via** $M_{p,i}$

**ELSE**

**Stop recording messages incoming from port j of p**

**record**$(M_{p,j})$

**FI**

**IF p has received a marker via all incoming channels THEN**

$local\_snapshot_p := true$

**FI**

**END**

# Propriétés de l'algorithme

- Pourquoi l'algorithme termine :
  - les canaux sont supposés fiables
  - les messages sont transmis FIFO
  - $s$ est vide à la terminaison
  - Quand $s$ est vide, tous les processus ont reçu le marqueur sur leurs canaux incidents.

# Applications de l'algorithme

- Calculer le nombre de jetons dans un réseau
- Détermination de point de reprise de calcul

# un réseau

- On considère un réseau de type anneau de 1 à 3 dans ce sens

# un réseau

- On considère un réseau de type anneau de 1 à 3 dans ce sens
- CL1 : le processus 1 envoie le jeton à 2 ; le processus devient local $(n(1) = 0)$ ; il positionne les variables associées aux canaux : $sent(1, 2) = 1$ et $rec(3, 1) = 0)$ ; il envoie le marqueur sur le canal $(1, 2)$.

# un réseau

- On considère un réseau de type anneau de 1 à 3 dans ce sens
- CL1 : le processus 1 envoie le jeton à 2 ; le processus devient local $(n(1) = 0)$ ; il positionne les variables associées aux canaux : $sent(1, 2) = 1$ et $rec(3, 1) = 0)$ ; il envoie le marqueur sur le canal $(1, 2)$.
- E : le processus i reçoit le jeton et le renvoie à son voisin.

# un réseau

- On considère un réseau de type anneau de 1 à 3 dans ce sens
- CL1 : le processus 1 envoie le jeton à 2 ; le processus devient local $(n(1) = 0)$ ; il positionne les variables associées aux canaux : $sent(1, 2) = 1$ et $rec(3, 1) = 0)$ ; il envoie le marqueur sur le canal $(1, 2)$.
- E : le processus i reçoit le jeton et le renvoie à son voisin.
- CL2 : le processus i reçoit le marqueur ; le processus i devient local$(n(i) = 0)$ ; il positionne $rec(i-1, i) = 1$ et $sent(i, i+1) = 1$ ; il envoie le marqueur sur le canal $(i, i+1)$.

# un réseau

- On considère un réseau de type anneau de 1 à 3 dans ce sens
- CL1 : le processus 1 envoie le jeton à 2 ; le processus devient local $(n(1) = 0)$ ; il positionne les variables associées aux canaux : $sent(1, 2) = 1$ et $rec(3, 1) = 0)$ ; il envoie le marqueur sur le canal $(1, 2)$.
- E : le processus i reçoit le jeton et le renvoie à son voisin.
- CL2 : le processus i reçoit le marqueur ; le processus i devient local$(n(i) = 0)$ ; il positionne $rec(i-1, i) = 1$ et $sent(i, i+1) = 1$ ; il envoie le marqueur sur le canal $(i, i+1)$.
- L'algorithme termine et le nombre de jetons enregistrés est : $n_1+n_2+n_3+sent(1, 2)-rec(1, 2)+sent(2, 3)-rec(2, 3)+sent(3, 1)-rec(3, 1$

# Algorithme de Lay et Yang

# Algorithme de Lay et Yang

- LY1 : L'initialisateur enregistre son état et quand il doit envoyer un message à un autre processus, il envoie le message (m,r) où r désigne une couleur.

# Algorithme de Lay et Yang

- $\mathrm{LY1}$ : L'initialisateur enregistre son état et quand il doit envoyer un message à un autre processus, il envoie le message (m,r) où r désigne une couleur.B
- $\mathrm{LY2}$ : Quand un processus reçoit un message (m,r), il enregistre son état, si ce n'est pas déjà fait, et accepe le message m.

roa

# Algorithme de Lay et Yang

- LY1 : L'initialisateur enregistre son état et quand il doit envoyer un message à un autre processus, il envoie le message (m,r) où r désigne une couleur.B
- LY2 : Quand un processus reçoit un message (m,r), il enregistre son état, si ce n'est pas déjà fait, et accepe le message m.

roa

- L'accessibilité de l'état global n'est pas garanti

# Section Courante

# Diffusion fiable

Une diffusion est fiable, si

- elle est valide : quand un processus diffuse, tous les processus membres du groupe de diffusion reçoivent.
- elle satisfait la propriété d'accord : si un processus reçoit, alors tous les autres membres du groupe reçoivent.
- elle est intègre : chaque message n'arrive qu'une et une seule fois.

¡ Les mécanismes de diffusion fiable sont de type

- **FIFO** : les messages sont délivrés selon l'ordre d'envoi (`FBCAST`)
- **causalité** : les messages sont délivrés selon un ordre respectant la causalité (`CBCAST`)
- **Atomique** : les messages sont tous délivrés dans le même ordre (`ABCAST`)

# Conventions et notations

- $\mathbf{receive}_P(m)$ : événement de réception d'un message m par le processus P.
- $\mathbf{delivery}_P(m)$ : événement de délivrance du message m au processus P.

# Conventions et notations

- $\mathbf{receive}_P(m)$ : événement de réception d'un message m par le processus P.
- $\mathbf{delivery}_P(m)$ : événement de délivrance du message m au processus P.
- Observation : $\mathbf{receive}_P(m)$ précède $\mathbf{delivery}_P(m)$
- Idée : différer la délivrance d'un message quand il est reçu.

# Ordre FIFO avec FBCAST

## Principe

Si un processus diffuse un message m1 puis un message m2, alors aucun processus du groupe ne livre le message m2 à moins que m1 ait été livré.

- Si $\mathbf{send}_P(m1) \rightsquigarrow \mathbf{send}_P(m2)$, alors pour tout processus Q du groupe de diffusion D, $\mathbf{delivery}_Q(m1) \rightsquigarrow \mathbf{cdlivery}_Q(m2)$.
- Idée de l'algorithme : à la réception des messages, on les stocke et on compare les estampilles des messages pour la composante d'envoi P.

# Ordre causal avec CBCAST

## Principe

Si

- un processus envoie un message m1
- la délivrance de m1 est suivi causalement de l'envoi de m2

alors tous les processus délivrent le message m2 après le message m1.

- Si $\mathbf{delivery}_Q(m1) \rightsquigarrow \mathbf{send}_P(m2)$, alors pour tout processus Q du groupe de diffusion D, $\mathbf{delivery}_Q(m1) \rightsquigarrow \mathbf{cdlivery}_Q(m2)$.
- Idée de l'algorithme : à la réception des messages, on les stocke et on compare les estampilles des messages pour la composante d'envoi P.

# Ordre atomique avec ABCAST

## Principe

Les processus d'un groupe livre les messages dans le même ordre.

- Si $\mathbf{delivery}_P(m1) \rightsquigarrow \mathbf{send}_P(m2)$, alors pour tout processus Q du groupe de diffusion D, $\mathbf{delivery}_Q(m1) \rightsquigarrow \mathbf{cdlivery}_Q(m2)$.
- Idée de l'algorithme : à la réception des messages, on les stocke et on compare les estampilles des messages pour la composante d'envoi P.

# Protocole de diffusion CBCAST

## Initialisation

Pour chaque site $i \in 1..n$, positionner les valeurs des vecteurs $VC_i$ à $0$.

## Diffusion de $m$ sur le site $i$

$$\begin{cases} VC_i(i) := VC_i(i)+1 \\ \textbf{Pour tout site j de 1..n} : Send_i(m, VC_i(i), j) \end{cases}$$

## Réception

$$\begin{cases} Receive_i(m, VC_m, j) \\ Wait(VC_m = VC_i(j)+1) \\ Wait(\forall j.j \in 1..n \wedge j \neq VC_m \leq VC_i(j)) \\ Deliver_i(m) \\ VC_i(j) = VC_i(j)+1 \end{cases}$$

# Conclusion

- Rôle des estampilles pour les algorithmes d'exclusion mutuelle
- Rôle de horloges vectorielles dans la diffusion des messages et la propriété de causalité