Fisher, Lynch et Paterson : Impossibilité du consensus déterministe dans les systèmes asynchrones

Stéphane Devismes

Université Joseph Fourier, Grenoble I

17 septembre 2016

Plan

Plan

Le résultat

≪ Il est impossible de résoudre de manière déterministe le consensus dans un système asynchrone où **au plus un** processus peut être défaillant ≫.

Fisher, Lynch et Paterson (1985)

Le résultat

≪ Il est impossible de résoudre de manière déterministe le consensus dans un système asynchrone où **au plus un** processus peut être défaillant ≫.

Fisher, Lynch et Paterson (1985)

N.b., Pas d'information sur l'éventuelle panne (*e.g.*, pas de détecteurs de pannes).

Le résultat

≪ Il est impossible de résoudre de manière déterministe le consensus dans un système asynchrone où **au plus un** processus peut être défaillant ≫.

Fisher, Lynch et Paterson (1985)

N.b., Pas d'information sur l'éventuelle panne (*e.g.*, pas de détecteurs de pannes). De plus, si la panne arrive, elle peut arriver n'importe quand durant l'exécution.

Le consensus

Multi-initiateurs

Le consensus

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$

Sortie : $d_p \in \{\perp,0,1\}$ initialisée à $\perp$;

Le consensus

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$

Sortie : $d_p \in \{\perp,0,1\}$ initialisée à $\perp$;
p doit décider (*i.e.*, affecter) une valeur booléenne dans d_p en respectant les conditions suivantes :

Le consensus

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$

Sortie : $d_p \in \{\perp,0,1\}$ initialisée à $\perp$;
p doit décider (*i.e.*, affecter) une valeur booléenne dans d_p en respectant les conditions suivantes :

Accord (uniforme) : Si deux processus p et q décident, alors ils décident la même valeur, $d_p = d_q$.

Le consensus

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0, 1\}$

Sortie : $d_p \in \{\perp, 0, 1\}$ initialisée à $\perp$;
p doit décider (*i.e.*, affecter) une valeur booléenne dans d_p en respectant les conditions suivantes :

Accord (uniforme) : Si deux processus p et q décident, alors ils décident la même valeur, $d_p = d_q$.

Validité : Toute valeur décidée est l'une des valeurs initiales.

Le consensus

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$

Sortie : $d_p \in \{\perp,0,1\}$ initialisée à $\perp$;
p doit décider (*i.e.*, affecter) une valeur booléenne dans d_p en respectant les conditions suivantes :

Accord (uniforme) : Si deux processus p et q décident, alors ils décident la même valeur, $d_p = d_q$.

Validité : Toute valeur décidée est l'une des valeurs initiales.

Terminaison : Tout processus correct décidera un jour.

Le consensus

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$

Sortie : $d_p \in \{\perp,0,1\}$ initialisée à $\perp$;
p doit décider (*i.e.*, affecter) une valeur booléenne dans d_p en respectant les conditions suivantes :

Accord (uniforme) : Si deux processus p et q décident, alors ils décident la même valeur, $d_p = d_q$.

Validité : Toute valeur décidée est l'une des valeurs initiales.

Terminaison : Tout processus correct décidera un jour.

Intégrité : Tout processus décide au plus une fois.

Le consensus

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$

Sortie : $d_p \in \{\perp,0,1\}$ initialisée à $\perp$;
p doit décider (*i.e.*, affecter) une valeur booléenne dans d_p en respectant les conditions suivantes :

Accord (uniforme) : Si deux processus p et q décident, alors ils décident la même valeur, $d_p = d_q$.

Validité : Toute valeur décidée est l'une des valeurs initiales.

Terminaison : Tout processus correct décidera un jour.

Intégrité : Tout processus décide au plus une fois.

ATTENTION :

Si pour tout processus p, $v_p = 0$ (resp. $v_p = 1$) alors la valeur décidée doit être 0 (resp. 1).

Le consensus dans les systèmes sans pannes

Avec un réseau de communications complet : Facile !

Le consensus dans les systèmes sans pannes

- Les initiateurs : réveil + diffusion de leur valeur initiale.

Le consensus dans les systèmes sans pannes

- Les initiateurs : réveil $+$ diffusion de leur valeur initiale.
- Lorsqu'un processus reçoit une proposition, il la sauvegarde.

  De plus, s'il n'était pas (déjà) réveillé :

  réveil $+$ diffusion de sa valeur initiale.

Le consensus dans les systèmes sans pannes

- Les initiateurs : réveil + diffusion de leur valeur initiale.
- Lorsqu'un processus reçoit une proposition, il la sauvegarde.

  De plus, s'il n'était pas (déjà) réveillé :

  $$\text{réveil} + \text{diffusion de sa valeur initiale.}$$
- Lorsqu'un processus a reçu des propositions de tous les autres, il décide.

  Toute règle d'intégrité déterministe vérifiant la validité est valable, à partir du moment où elle est commune à tous.

  E.g., si un processus a reçu au moins $\lceil \frac{n}{2} \rceil$ valeurs 0, il décide 0, sinon il décide 1.

Aspect fondamental du résultat

1. Le consensus est le **problème d'accord** le plus simple.

   (accord sur deux valeurs Booléennes)

   Autres problèmes d'accord : le registre partagé, la diffusion atomique, la duplication de machine d'état, la synchronisation, ...

   Les problèmes d'accord sont omniprésents en système distribué. (Election, allocation de resource, ...)

Aspect fondamental du résultat

1. Le consensus est le **problème d'accord** le plus simple.

   (accord sur deux valeurs Booléennes)

   Autres problèmes d'accord : le registre partagé, la diffusion atomique, la duplication de machine d'état, la synchronisation, ...

   Les problèmes d'accord sont omniprésents en système distribué. (Election, allocation de resource, ...)

2. L'impossibilité est obtenue malgré des **hypothèses très fortes** sur le système (canaux fiables, au plus une panne, ...).

Plan

Remarque

Le modèle doit être le plus général possible afin d'obtenir le résultat le plus général possible.

Liens

- Asynchrones

Liens

- Asynchrones
- Fiables : chaque message finit par être livré, exactement une fois et seulement s'il a été envoyé.

  Ainsi, chaque message finit par être reçu **à condition que** le processus destinataire essaie de le recevoir infiniment souvent.

Liens

- Asynchrones
- Fiables : chaque message finit par être livré, exactement une fois et seulement s'il a été envoyé.

  Ainsi, chaque message finit par être reçu **à condition que** le processus destinataire essaie de le recevoir infiniment souvent.

- Ordre d'arrivée : les messages peuvent être retardés arbitrairement longtemps et sont livrés dans n'importe quel ordre.

Processus

- Il y a $n \geq 2$ processus et au moins $n - 1$ sont corrects.

  Au plus un processus peut tomber en panne.
  Dans ce cas, c'est une panne **crash** définitive, le processus cesse définitivement de faire des pas de calculs.

Processus

- Il y a $n \geq 2$ processus et au moins $n - 1$ sont corrects.

  Au plus un processus peut tomber en panne.
  Dans ce cas, c'est une panne **crash** définitive, le processus cesse définitivement de faire des pas de calculs.
- Les processus n'ont aucun accès à une horloge globale.

Processus

- Il y a $\boldsymbol{n} \geq \mathbf{2}$ processus et au moins $\boldsymbol{n} - \mathbf{1}$ sont corrects.

  Au plus un processus peut tomber en panne.
  Dans ce cas, c'est une panne **crash** définitive, le processus cesse définitivement de faire des pas de calculs.
- Les processus n'ont aucun accès à une horloge globale.
- Les processus n'ont pas d'information sur les pannes arrivées ou à venir (*e.g.*, détecteur de pannes).

Processus

- Il y a $\boldsymbol{n} \geq \mathbf{2}$ processus et au moins $\boldsymbol{n} - \mathbf{1}$ sont corrects.

  Au plus un processus peut tomber en panne.
  Dans ce cas, c'est une panne **crash** définitive, le processus cesse définitivement de faire des pas de calculs.
- Les processus n'ont aucun accès à une horloge globale.
- Les processus n'ont pas d'information sur les pannes arrivées ou à venir (*e.g.*, détecteur de pannes).
- Processus :

Processus

- Il y a $\boldsymbol{n} \geq \mathbf{2}$ processus et au moins $\boldsymbol{n} - \mathbf{1}$ sont corrects.

  Au plus un processus peut tomber en panne.
  Dans ce cas, c'est une panne **crash** définitive, le processus cesse définitivement de faire des pas de calculs.
- Les processus n'ont aucun accès à une horloge globale.
- Les processus n'ont pas d'information sur les pannes arrivées ou à venir (*e.g.*, détecteur de pannes).
- Processus :
  - Automate déterministe

Processus

- Il y a $\boldsymbol{n} \geq \mathbf{2}$ processus et au moins $\boldsymbol{n} - \mathbf{1}$ sont corrects.

  Au plus un processus peut tomber en panne.
  Dans ce cas, c'est une panne **crash** définitive, le processus cesse définitivement de faire des pas de calculs.
- Les processus n'ont aucun accès à une horloge globale.
- Les processus n'ont pas d'information sur les pannes arrivées ou à venir (*e.g.*, détecteur de pannes).
- Processus :
  - Automate déterministe
  - Mémoire locale (potentiellement infinie)

Processus

- Il y a $\boldsymbol{n} \geq \mathbf{2}$ processus et au moins $\boldsymbol{n} - \mathbf{1}$ sont corrects.

  Au plus un processus peut tomber en panne.
  Dans ce cas, c'est une panne **crash** définitive, le processus cesse définitivement de faire des pas de calculs.
- Les processus n'ont aucun accès à une horloge globale.
- Les processus n'ont pas d'information sur les pannes arrivées ou à venir (*e.g.*, détecteur de pannes).
- Processus :
  - Automate déterministe
  - Mémoire locale (potentiellement infinie)
  - Capable de communiquer avec tous les autres processus **par envoi de message**.

Exécution d'un pas de calcul

Etape atomique.

Exécution d'un pas de calcul

Etape atomique.

En une étape, un processus peut :

- Essayer de recevoir un message,
- Faire un calcul local

  (basé sur la réception ou non d'un message)
  (en cas de réception d'un message, le calcul pourra être basé sur le contenu du message)
- Envoyer un nombre quelconque mais fini de messages aux autres processus.

Intuition

Dans ce modèle, il est impossible pour un processus de détecter si un autre processus est **en panne ou s'il est simplement très lent**.

Consensus faible

Pour tout processus p

Entrée : $v_p \in \{0,1\}$

Sortie : $d_p \in \{\perp,0,1\}$ initialisée à $\perp$;
p doit décider une valeur booléenne dans d_p en respectant les conditions suivantes :

Accord (uniforme) : Si deux processus p et q décide, alors ils décident la même valeur, $d_p = d_q$.

Intégrité : Tout processus décide au plus une fois.

Consensus faible

Pour tout processus p

Entrée : $v_p \in \{0,1\}$

Sortie : $d_p \in \{\perp,0,1\}$ initialisée à $\perp$;
p doit décider une valeur booléenne dans d_p en respectant les conditions suivantes :

Accord (uniforme) : Si deux processus p et q décide, alors ils décident la même valeur, $d_p = d_q$.

Validité faible : Chacune des deux valeurs (0 ou 1) doit pouvoir être décidée (peut-être, à partir de configurations initiales différentes)

Intégrité : Tout processus décide au plus une fois.

Consensus faible

Pour tout processus p

Entrée : $v_p \in \{0, 1\}$

Sortie : $d_p \in \{\perp, 0, 1\}$ initialisée à $\perp$;
p doit décider une valeur booléenne dans d_p en respectant les conditions suivantes :

Accord (uniforme) : Si deux processus p et q décide, alors ils décident la même valeur, $d_p = d_q$.

Validité faible : Chacune des deux valeurs (0 ou 1) doit pouvoir être décidée (peut-être, à partir de configurations initiales différentes)

Terminaison faible : Au moins un processus doit finir par décider

Intégrité : Tout processus décide au plus une fois.

Algorithme de consensus

Soit $\mathcal{P}$ un algorithme de consensus.
Chaque processus p a

- un bit d'entrée $v_p \in \{0, 1\}$,
- une variable de sortie d_p qui peut prendre les valeurs $\{\perp, 0, 1\}$,
- et un espace de stockage interne non borné.

Etats

Etat interne : Les valeurs des variables d'entrée, sortie, internes et le compteur de programme.

Etats

Etat interne : Les valeurs des variables d'entrée, sortie, internes et le compteur de programme.

Etats internes initiaux : donnent une valeur fixée à chaque variable sauf au bit d'entrée v_p.

Etats

Etat interne : Les valeurs des variables d'entrée, sortie, internes et le compteur de programme.

Etats internes initiaux : donnent une valeur fixée à chaque variable sauf au bit d'entrée v_p.
Il y a **deux états initiaux** possibles par processus, l'un où l'entrée vaut 0 et l'autre où l'entrée vaut 1.

Etats

Etat interne : Les valeurs des variables d'entrée, sortie, internes et le compteur de programme.

Etats internes initiaux : donnent une valeur fixée à chaque variable sauf au bit d'entrée v_p.
Il y a **deux états initiaux** possibles par processus, l'un où l'entrée vaut 0 et l'autre où l'entrée vaut 1.
En particulier, la variable de sortie d_p a pour valeur initiale $\perp$ (le processus n'a pas encore décidé).

Etats

Etat interne : Les valeurs des variables d'entrée, sortie, internes et le compteur de programme.

Etats internes initiaux : donnent une valeur fixée à chaque variable sauf au bit d'entrée v_p.
Il y a **deux états initiaux** possibles par processus, l'un où l'entrée vaut 0 et l'autre où l'entrée vaut 1.
En particulier, la variable de sortie d_p a pour valeur initiale $\perp$ (le processus n'a pas encore décidé).

Etats de décision : Les états dans lesquels la valeur de la variable de sortie du processus est 0 ou 1.

Etats : exemple

Supposons qune protocole où chaque processus p a une seule variable **entière** interne x initialisée à 0.

Etats : exemple

Supposons qune protocole où chaque processus p a une seule variable **entière** interne x initialisée à 0.

Exemple d'état interne : $\langle v_p = 0, d_p = \perp, x_p = 10, CP_p = 0x4040 \rangle$

Etats : exemple

Supposons qune protocole où chaque processus p a une seule variable **entière** interne x initialisée à 0.

Exemple d'état interne : $\langle v_p = 0, d_p = \perp, x_p = 10, CP_p = 0x4040 \rangle$

Exemple d'état interne initial : $\langle v_p = 0, d_p = \perp, x_p = 0, CP_p = 0x4000 \rangle$

Etats : exemple

Supposons qune protocole où chaque processus p a une seule variable **entière** interne x initialisée à 0.

Exemple d'état interne : $\langle v_p = 0, d_p = \perp, x_p = 10, CP_p = 0x4040 \rangle$

Exemple d'état interne initial : $\langle v_p = 0, d_p = \perp, x_p = 0, CP_p = 0x4000 \rangle$

Exemple d'états de décision :

$$\langle v_p = 0, d_p = 1, x_p = 30, CP_p = 0x4048 \rangle$$

Fonction de transition $\approx$ algorithme local

Chaque processus agit de manière déterministe en fonction de sa fonction de transition.

Fonction de transition $\approx$ algorithme local

Chaque processus agit de manière déterministe en fonction de sa fonction de transition.

La fonction de transition ne peut pas changer la valeur de la variable de sortie dans un état de décision : **Cette variable ne peut être écrite qu'une fois !** (Intégrité)

Messages

Message : (p, m) où p l'identité du processus destinataire et m la valeur du message, $m \in M$.

Messages

Message : (p, m) où p l'identité du processus destinataire et m la valeur du message, $m \in M$.

Réseau : multi-ensemble de messages (car communications non-FIFO), appelé tampon-mémoire, où sont gardés les messages envoyés non encore reçus.

Messages

Message : (p, m) où p l'identité du processus destinataire et m la valeur du message, $m \in M$.

Réseau : multi-ensemble de messages (car communications non-FIFO), appelé tampon-mémoire, où sont gardés les messages envoyés non encore reçus.

`envoi(`p`,`m`)` : Met (p, m) dans le tampon-mémoire.

`reçoit(`p`)` : Supprime un message (p, m) du tampon-mémoire et retourne m (dans ce cas, (p, m) est reçu)
ou
retourne $\emptyset$ et laisse le tampon-mémoire inchangé (en particulier, s'il n'existe pas de message pour p).

Non-déterminisme

Le système de messages se comporte de manière non-déterministe.

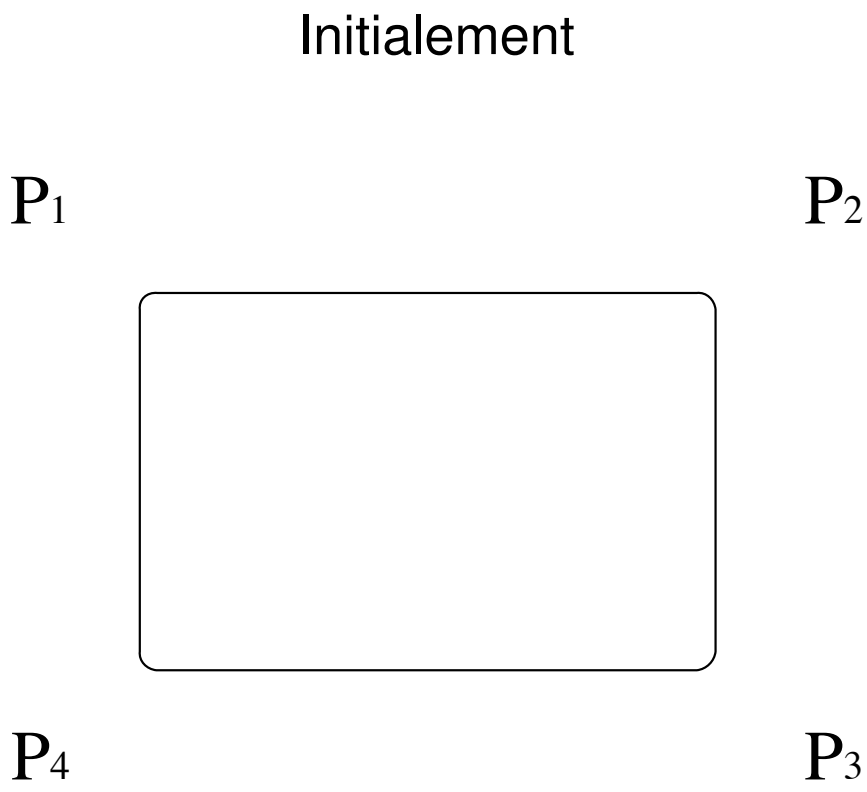
Non-déterminisme

Le système de messages se comporte de manière non-déterministe.

Seule condition : si `reçoit(`p`)` est exécutée infiniment souvent, alors tous les messages $(p, \_)$ finissent par être reçus.

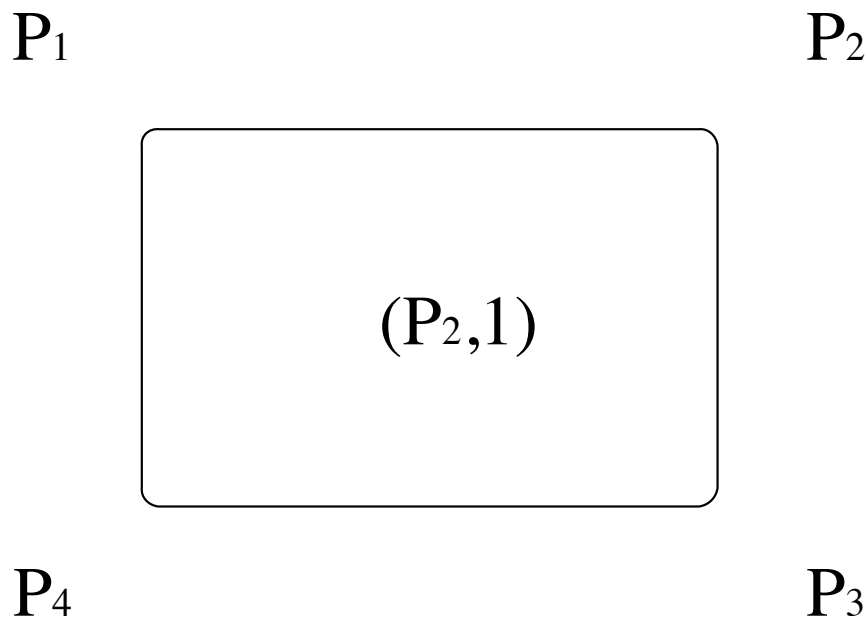
En particulier, le système de messages peut retourner $\emptyset$ un nombre fini de fois en réponse de l'appel `reçoit(`p`)`, bien qu'un message (p, m) soit présent dans le tampon-mémoire. (Asynchronisme)

Exemple



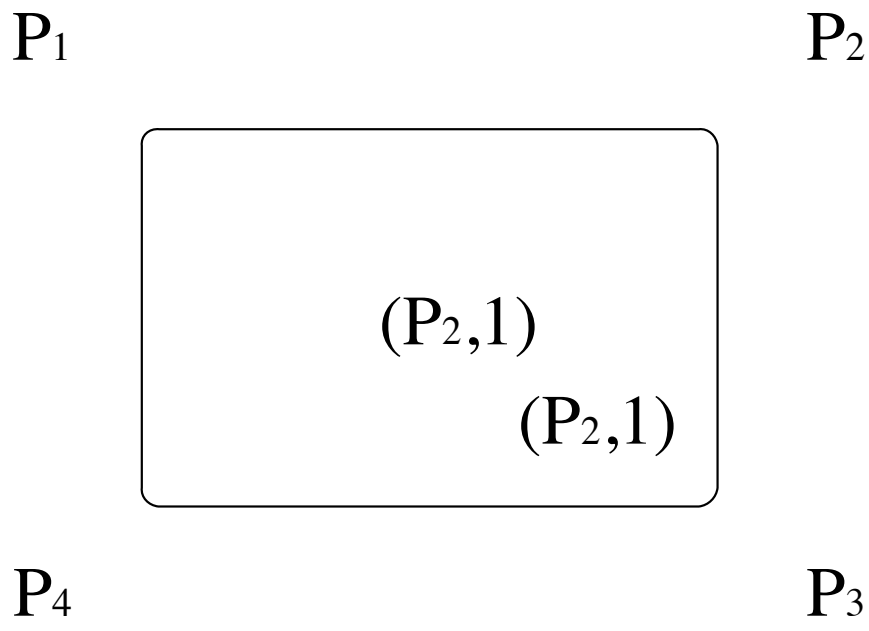
Exemple

p_1 exécute `envoi(`p_2`,1)`



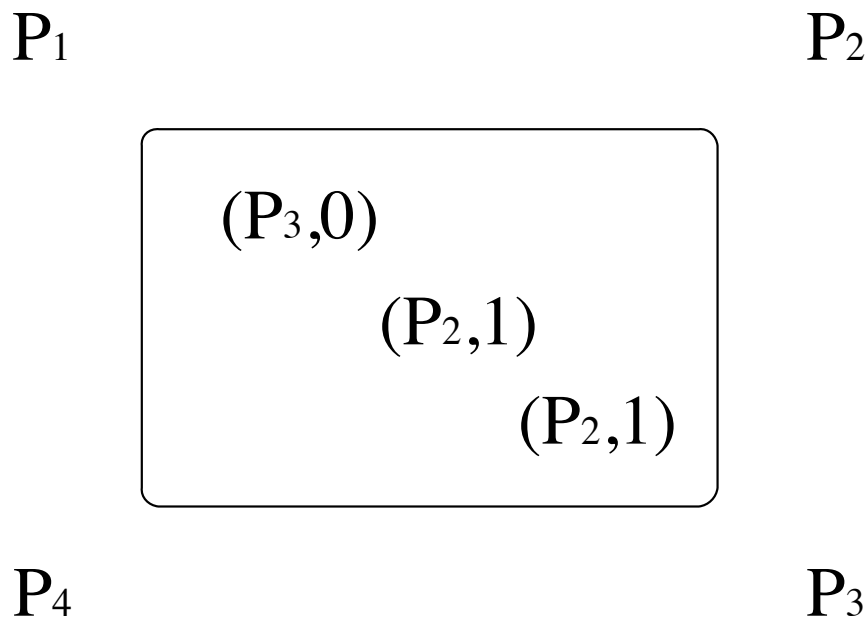
Exemple

p_3 exécute `envoi(`p_2`,1)`



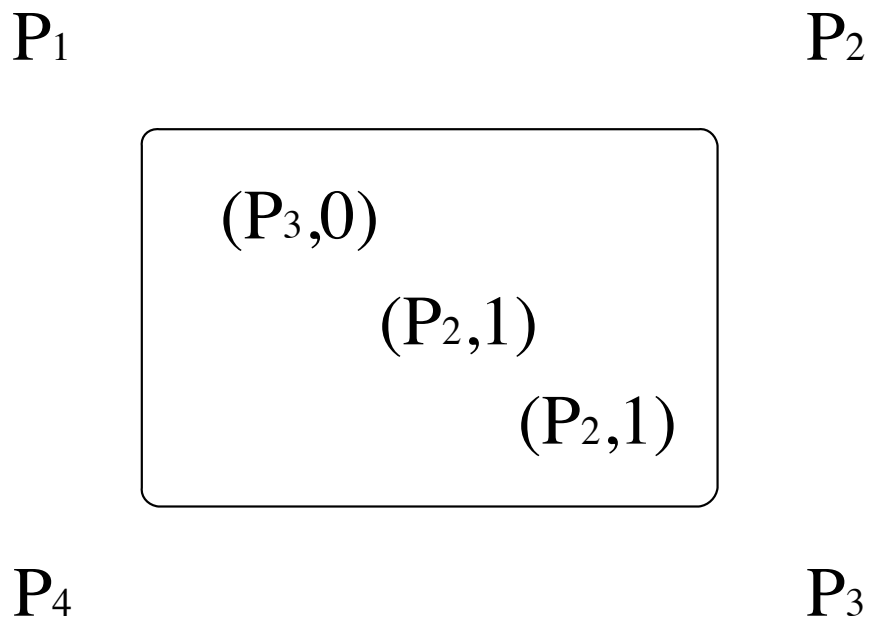
Exemple

p_1 exécute `envoi(`p_3`,0)`



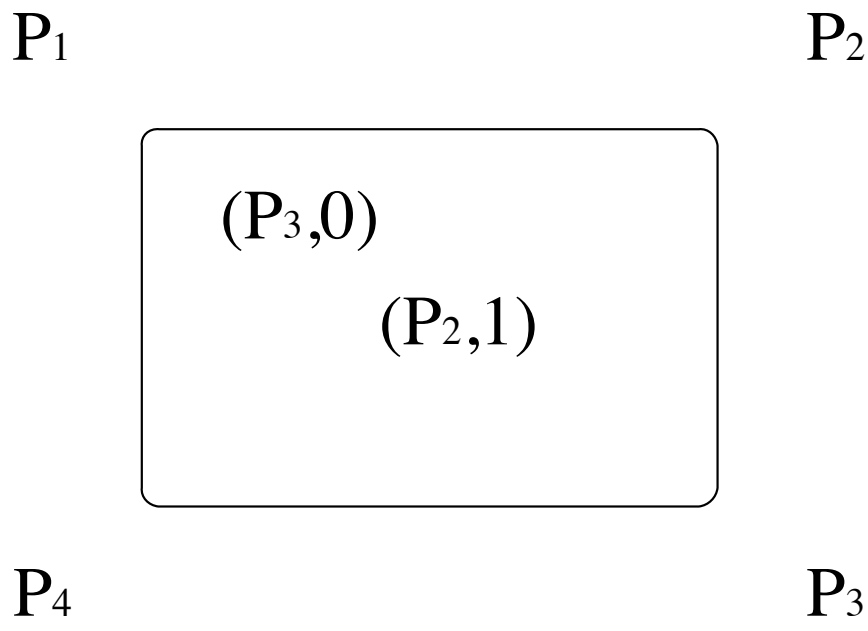
Exemple

`reçoit(`p_1`)` retourne $\emptyset$ à p_1



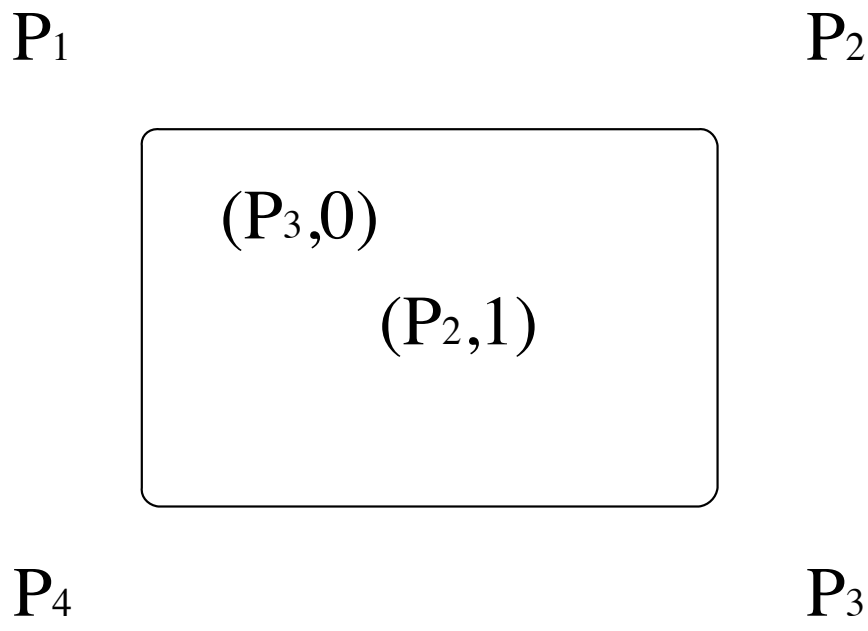
Exemple

`reçoit(`p_2`)` retourne 1 à p_2



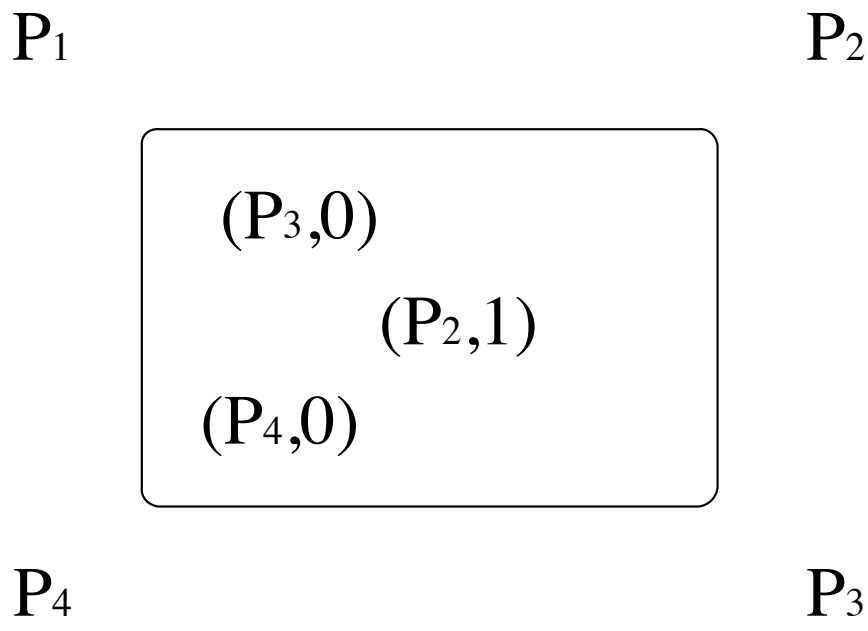
Exemple

`reçoit(`p_2`)` retourne $\emptyset$ à p_2



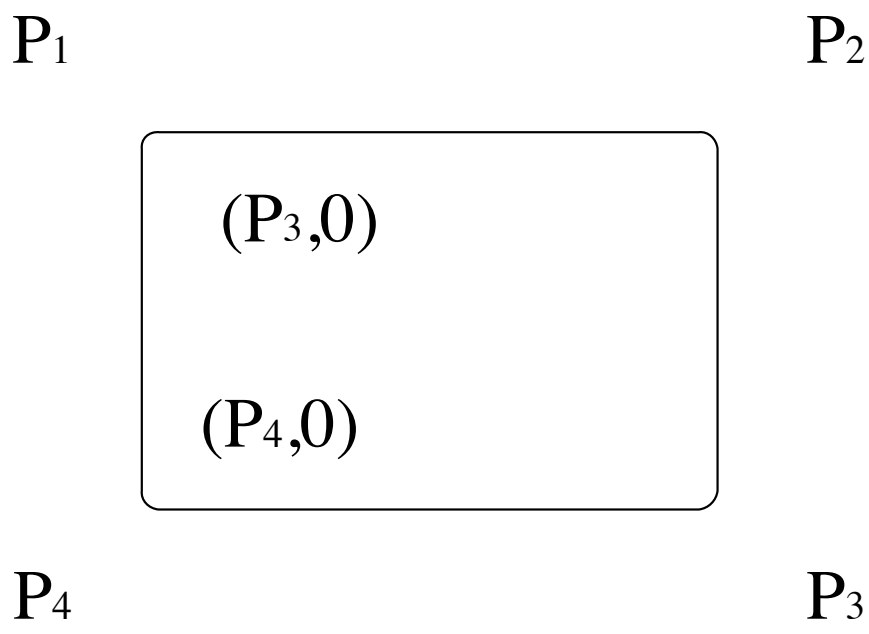
Exemple

p_3 exécute `envoi(`p_4`,0)`



Exemple

`reçoit(`p_2`)` retourne 1 à p_2



Configuration

Configuration : état interne de chaque processus

+

contenu du tampon-mémoire de message.

Configuration

Configuration : état interne de chaque processus

$+$

contenu du tampon-mémoire de message.

Configuration initiale :

- Tous les processus sont dans **un état initial** et
- le tampon-mémoire de message est **vide**.

Configuration : exemple

Avec un réseau à 3 processus p_1, p_2, p_3 où chaque processus a une seul variable interne entière x initialisée à 0

Configuration : exemple

Avec un réseau à 3 processus p_1, p_2, p_3 où chaque processus a une seul variable interne entière x initialisée à 0

Configuration :

$[\langle v_{p_1} = 0, d_{p_1} = \perp, x_{p_1} = 10, CP_{p_1} = 0x4040\rangle,$
$\langle v_{p_2} = 0, d_{p_2} = 1, x_{p_2} = 32, CP_{p_2} = 0x4048\rangle,$
$\langle v_{p_3} = 1, d_{p_3} = \perp, x_{p_3} = 14, CP_{p_3} = 0x4044\rangle,$
$\{(p_2, m_a), (p_3, m_a), (p_3, m_b), (p_3, m_b)\}]$

Configuration : exemple

Avec un réseau à 3 processus p_1, p_2, p_3 où chaque processus a une seul variable interne entière x initialisée à 0

Configuration :

$[\langle v_{p_1} = 0, d_{p_1} = \perp, x_{p_1} = 10, CP_{p_1} = 0x4040\rangle,$
$\langle v_{p_2} = 0, d_{p_2} = 1, x_{p_2} = 32, CP_{p_2} = 0x4048\rangle,$
$\langle v_{p_3} = 1, d_{p_3} = \perp, x_{p_3} = 14, CP_{p_3} = 0x4044\rangle,$
$\{(p_2, m_a), (p_3, m_a), (p_3, m_b), (p_3, m_b)\}]$

Configuration initiale :

$[\langle v_{p_1} = 0, d_{p_1} = \perp, x_{p_1} = 0, CP_{p_1} = 0x4000\rangle,$
$\langle v_{p_2} = 0, d_{p_2} = \perp, x_{p_2} = 0, CP_{p_2} = 0x4020\rangle,$
$\langle v_{p_3} = 1, d_{p_3} = \perp, x_{p_3} = 0, CP_{p_3} = 0x4010\rangle, \emptyset]$

Etape

Passage d'une configuration à une autre : **exécution atomique** de la fonction de transition d'un seul processus.

Etape

Passage d'une configuration à une autre : **exécution atomique** de la fonction de transition d'un seul processus.

Soit C une configuration. Une étape se déroule en deux phases :

1. p exécute `reçoit(`p`)` pour obtenir une valeur $m \in M \cup \{\emptyset\}$.

Etape

Passage d'une configuration à une autre : **exécution atomique** de la fonction de transition d'un seul processus.

Soit C une configuration. Une étape se déroule en deux phases :

1. p exécute `reçoit(`p`)` pour obtenir une valeur $m \in M \cup \{\emptyset\}$.
2. En fonction de l'état interne de p dans C et de m,
   - p passe dans un nouvel état interne et
   - envoie un nombre fini de messages aux autres processus.

Etape

Passage d'une configuration à une autre : **exécution atomique** de la fonction de transition d'un seul processus.

Soit C une configuration. Une étape se déroule en deux phases :

1. p exécute `reçoit(`p`)` pour obtenir une valeur $m \in M \cup \{\emptyset\}$.
2. En fonction de l'état interne de p dans C et de m,
   - p passe dans un nouvel état interne et
   - envoie un nombre fini de messages aux autres processus.

L'étape est entièrement déterminée par la paire $e = (p, m)$: l'évènement e peut-être vu comme « p reçoit m ».

Etape

Passage d'une configuration à une autre : **exécution atomique** de la fonction de transition d'un seul processus.

Soit C une configuration. Une étape se déroule en deux phases :

1. p exécute `reçoit(`p`)` pour obtenir une valeur $m \in M \cup \{\emptyset\}$.
2. En fonction de l'état interne de p dans C et de m,
   - p passe dans un nouvel état interne et
   - envoie un nombre fini de messages aux autres processus.

L'étape est entièrement déterminée par la paire $e = (p, m)$: l'évènement e peut-être vu comme « p reçoit m ».

$e(C)$: configuration résultant de l'application de e sur C.

Etape

Passage d'une configuration à une autre : **exécution atomique** de la fonction de transition d'un seul processus.

Soit C une configuration. Une étape se déroule en deux phases :

1. p exécute `reçoit(`p`)` pour obtenir une valeur $m \in M \cup \{\emptyset\}$.
2. En fonction de l'état interne de p dans C et de m,
   - p passe dans un nouvel état interne et
   - envoie un nombre fini de messages aux autres processus.

L'étape est entièrement déterminée par la paire $e = (p, m)$: l'évènement e peut-être vu comme « p reçoit m ».

$e(C)$: configuration résultant de l'application de e sur C.

$(p, \emptyset)$ peut toujours être appliqué sur n'importe C (asynchronisme) : il est toujours possible pour un processus d'exécuter une nouvelle étape.

Ordonnancement

Un ordonnancement depuis C est une suite **finie ou infinie** σ d'**évènements** qui peuvent être appliqués séquentiellement depuis C.

Ordonnancement

Un ordonnancement depuis C est une suite **finie ou infinie** σ d'**évènements** qui peuvent être appliqués séquentiellement depuis C.

Ex : $(p_3, \emptyset), (p_1, \emptyset), (p_1, m_a), (p_1, \emptyset), (p_2, m_a), (p_1, m_a), (p_3, m_b)$

Ordonnancement

Un ordonnancement depuis C est une suite **finie ou infinie** σ d'**évènements** qui peuvent être appliqués séquentiellement depuis C.

Ex : $(p_3, \emptyset), (p_1, \emptyset), (p_1, m_a), (p_1, \emptyset), (p_2, m_a), (p_1, m_a), (p_3, m_b)$

La suite de configurations associée est appelée exécution.

Ordonnancement

Un ordonnancement depuis C est une suite **finie ou infinie** σ d'**évènements** qui peuvent être appliqués séquentiellement depuis C.

Ex : $(p_3, \emptyset), (p_1, \emptyset), (p_1, m_a), (p_1, \emptyset), (p_2, m_a), (p_1, m_a), (p_3, m_b)$

La suite de configurations associée est appelée exécution.

Si σ est finie, alors nous notons $\sigma(C)$ la configuration obtenue en exécutant σ à partir de C, cette configuration est dite atteignable depuis C.

Ordonnancement

Un ordonnancement depuis C est une suite **finie ou infinie** σ d'**évènements** qui peuvent être appliqués séquentiellement depuis C.

Ex : $(p_3, \emptyset), (p_1, \emptyset), (p_1, m_a), (p_1, \emptyset), (p_2, m_a), (p_1, m_a), (p_3, m_b)$

La suite de configurations associée est appelée exécution.

Si σ est finie, alors nous notons $\sigma(C)$ la configuration obtenue en exécutant σ à partir de C, cette configuration est dite atteignable depuis C.

Une configuration atteignable depuis une configuration initiale est dite accessible.

Ordonnancement

Un ordonnancement depuis C est une suite **finie ou infinie** σ d'**évènements** qui peuvent être appliqués séquentiellement depuis C.

Ex : $(p_3, \emptyset), (p_1, \emptyset), (p_1, m_a), (p_1, \emptyset), (p_2, m_a), (p_1, m_a), (p_3, m_b)$

La suite de configurations associée est appelée exécution.

Si σ est finie, alors nous notons $\sigma(C)$ la configuration obtenue en exécutant σ à partir de C, cette configuration est dite atteignable depuis C.

Une configuration atteignable depuis une configuration initiale est dite accessible.

Dans la suite, nous ne considérerons que des configurations accessibles.

Vocabulaire

Une configuration C a une valeur de décision v si au moins un processus p est dans un état de décision avec $d_p = v$.

Vocabulaire

Une configuration C a une valeur de décision v si au moins un processus p est dans un état de décision avec $d_p = v$.

Un algorithme de consensus est partiellement correct s'il vérifie les deux conditions suivantes :

1. Aucune configuration accessible a **plus d'une valeur de décision**. (Accord et intégrité)
2. Pour chaque valeur $v \in \{0, 1\}$, **au moins une configuration accessible** a une valeur de décision v. (Validité faible)

Vocabulaire

Une configuration C a une valeur de décision v si au moins un processus p est dans un état de décision avec $d_p = v$.

Un algorithme de consensus est partiellement correct s'il vérifie les deux conditions suivantes :

1. Aucune configuration accessible a **plus d'une valeur de décision**. (Accord et intégrité)
2. Pour chaque valeur $v \in \{0, 1\}$, **au moins une configuration accessible** a une valeur de décision v. (Validité faible)

Une exécution est admissible si **au plus un processus est défaillant** (il fait un nombre fini de pas de calcul) et **tous les messages envoyés vers des processus corrects finissent par être livrés**.

Vocabulaire

Une configuration C a une valeur de décision v si au moins un processus p est dans un état de décision avec $d_p = v$.

Un algorithme de consensus est partiellement correct s'il vérifie les deux conditions suivantes :

1. Aucune configuration accessible a **plus d'une valeur de décision**. (Accord et intégrité)
2. Pour chaque valeur $v \in \{0, 1\}$, **au moins une configuration accessible** a une valeur de décision v. (Validité faible)

Une exécution est admissible si **au plus un processus est défaillant** (il fait un nombre fini de pas de calcul) et **tous les messages envoyés vers des processus corrects finissent par être livrés**.

Une exécution est une exécution décidante si **au moins un** processus atteint un état de décision durant l'exécution. (terminaison faible)

Vocabulaire

Une configuration C a une valeur de décision v si au moins un processus p est dans un état de décision avec $d_p = v$.

Un algorithme de consensus est partiellement correct s'il vérifie les deux conditions suivantes :

1. Aucune configuration accessible a **plus d'une valeur de décision**. (Accord et intégrité)
2. Pour chaque valeur $v \in \{0, 1\}$, **au moins une configuration accessible** a une valeur de décision v. (Validité faible)

Une exécution est admissible si **au plus un processus est défaillant** (il fait un nombre fini de pas de calcul) et **tous les messages envoyés vers des processus corrects finissent par être livrés**.

Une exécution est une exécution décidante si **au moins un** processus atteint un état de décision durant l'exécution. (terminaison faible)

Un algorithme de consensus $\mathcal{P}$ est correct en dépit d'une faute s'il est **partiellement correct** et toutes ses exécutions **admissibles sont décidantes**.

Plan

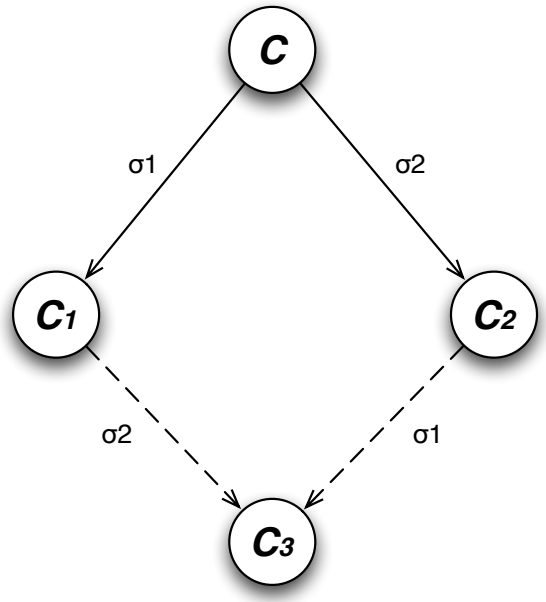
Premier résultat : commutativité

Lemme 1

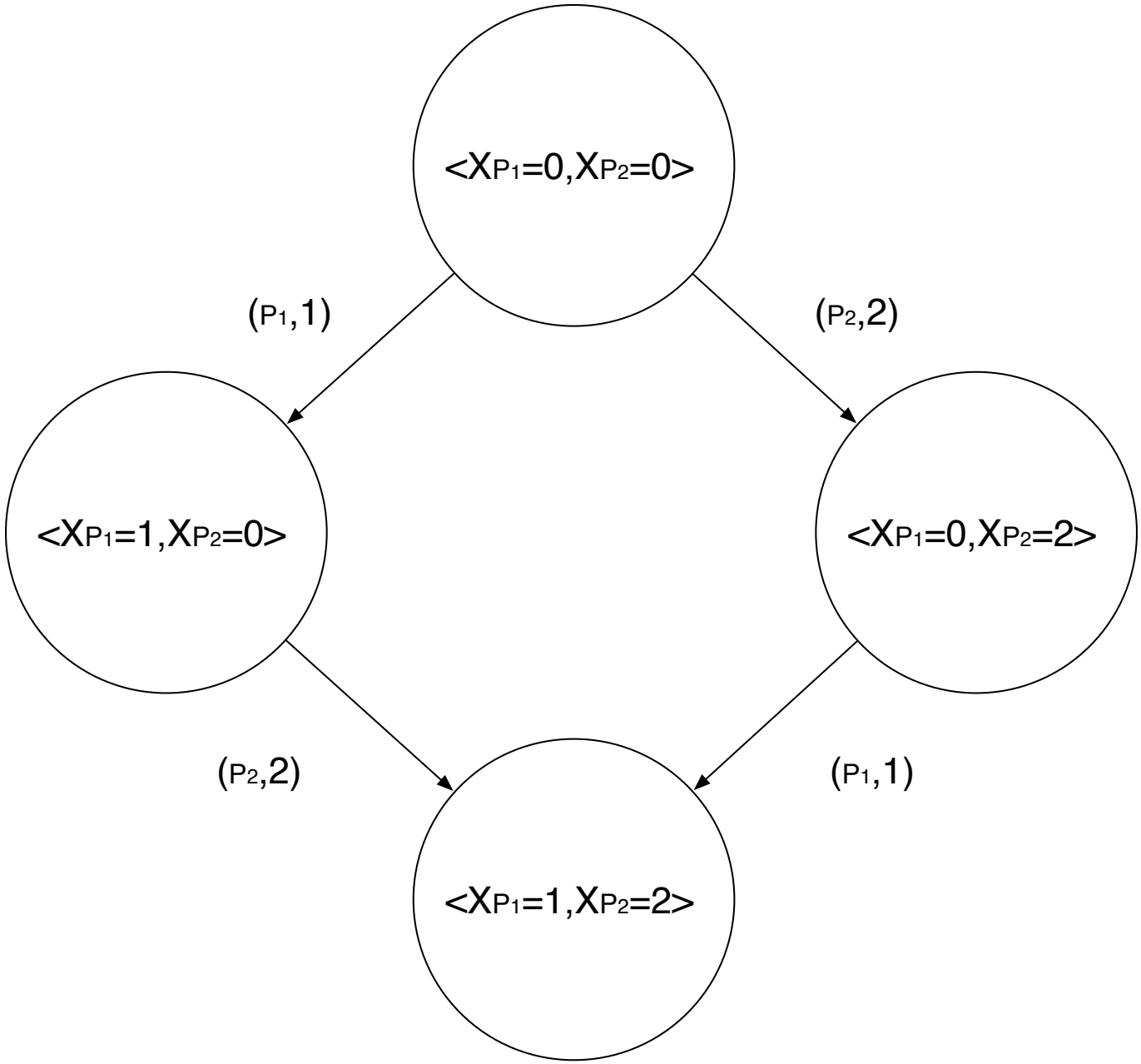
Soit C une configuration. Soient σ_1 et σ_2 deux ordonnancements qui mènent respectivement à C_1 et C_2 à partir de C.

Si les ensembles de processus exécutant respectivement des étapes dans σ_1 et σ_2 sont disjoints, alors

- σ_1 peut être appliqué à C_2 et σ_2 peut être appliqué à C_1,
- et tous deux mènent à la même configuration, C_3.



Exemple



Idée intuitive de la preuve

Supposons l'existence d'un protocole de consensus $\mathcal{P}$ qui est correct en dépit d'une faute.

Idée intuitive de la preuve

Supposons l'existence d'un protocole de consensus $\mathcal{P}$ qui est correct en dépit d'une faute.

L'idée est de montrer certaines circonstances sous lesquelles $\mathcal{P}$ ne peut jamais décider.

Idée intuitive de la preuve

Supposons l'existence d'un protocole de consensus $\mathcal{P}$ qui est correct en dépit d'une faute.

L'idée est de montrer certaines circonstances sous lesquelles $\mathcal{P}$ ne peut jamais décider.

Nous prouvons cela en deux étapes.

1. Nous montrons qu'il existe des configurations initiales où la valeur de décision n'est pas déjà déterminée.
2. Ensuite, nous construisons une **exécution admissible** qui évite en permanence d'exécuter des étapes qui engagent le système vers une décision particulière.

Configurations bivalentes et univalentes

Soit C une configuration et soit V l'ensemble des valeurs de décision des configurations atteignables depuis C.

Configurations bivalentes et univalentes

Soit C une configuration et soit V l'ensemble des valeurs de décision des configurations atteignables depuis C.

C est bivalente si $|V| = 2$.

Configurations bivalentes et univalentes

Soit C une configuration et soit V l'ensemble des valeurs de décision des configurations atteignables depuis C.

C est bivalente si $|V| = 2$.

C est univalente si $|V| = 1$.

Configurations bivalentes et univalentes

Soit C une configuration et soit V l'ensemble des valeurs de décision des configurations atteignables depuis C.

C est bivalente si $|V| = 2$.

C est univalente si $|V| = 1$.

Dans le cas d'une configuration univalente, nous parlerons d'0-valente ou 1-valente en fonction de la valeur de décision correspondante.

Configurations bivalentes et univalentes

Soit C une configuration et soit V l'ensemble des valeurs de décision des configurations atteignables depuis C.

C est bivalente si $|V| = 2$.

C est univalente si $|V| = 1$.

Dans le cas d'une configuration univalente, nous parlerons d'0-valente ou 1-valente en fonction de la valeur de décision correspondante.

Puisque $\mathcal{P}$ est correct et puisqu'il y a toujours des exécutions admissibles, on a toujours $V \neq \emptyset$.

Résultat 2

Lemme 2

$\mathcal{P}$ a (au moins) une configuration initiale bivalente.

Résultat 2

Lemme 2

$\mathcal{P}$ a (au moins) une configuration initiale bivalente.

Idée de la preuve : quelle que soit la procédure pour décider une valeur, il existe toujours une configuration initiale bivalente, c'est-à-dire une configuration à partir de laquelle les deux valeurs peuvent être décidées.

Résultat 2

Lemme 2

$\mathcal{P}$ a (au moins) une configuration initiale bivalente.

Idée de la preuve : quelle que soit la procédure pour décider une valeur, il existe toujours une configuration initiale bivalente, c'est-à-dire une configuration à partir de laquelle les deux valeurs peuvent être décidées.

Par exemple, si on décide 0 lorsque le nombre de 0 proposés est supérieur ou égal au nombre de 1 proposés, une configuration où il y a un 1 proposé de plus que de 0, est bivalente.

En effet, un processus proposant 1 ne fait aucun pas de calcul (c'est possible s'il tombe en panne), alors 0 sera décidé, sinon 1 peut-être décidé.

Preuve du lemme 2

Supposons que $\mathcal{P}$ n'a pas de configuration initiale bivalente.

Preuve du lemme 2

Supposons que $\mathcal{P}$ n'a pas de configuration initiale bivalente.

De part la correction partielle, $\mathcal{P}$ doit avoir à la fois des configurations initiales 0-valentes et 1-valentes.

Preuve du lemme 2

Supposons que $\mathcal{P}$ n'a pas de configuration initiale bivalente.

De part la correction partielle, $\mathcal{P}$ doit avoir à la fois des configurations initiales 0-valentes et 1-valentes.

Nous disons que deux configurations initiales sont adjacentes si elles diffèrent par exactement une valeur initiale v_p d'un seul processus p.

Preuve du lemme 2

Supposons que $\mathcal{P}$ n'a pas de configuration initiale bivalente.

De part la correction partielle, $\mathcal{P}$ doit avoir à la fois des configurations initiales 0-valentes et 1-valentes.

Nous disons que deux configurations initiales sont adjacentes si elles diffèrent par exactement une valeur initiale v_p d'un seul processus p.

Toute paire de configurations initiales sont jointes par une chaîne de configurations initiales, chacune adjacente à la suivante : par exemple de $(0,0,0)$ à $(1,1,1)$ on a, entre autres :
$(0,0,0) \rightarrow (0,0,1) \rightarrow (0,1,1) \rightarrow (1,1,1)$

Preuve du lemme 2

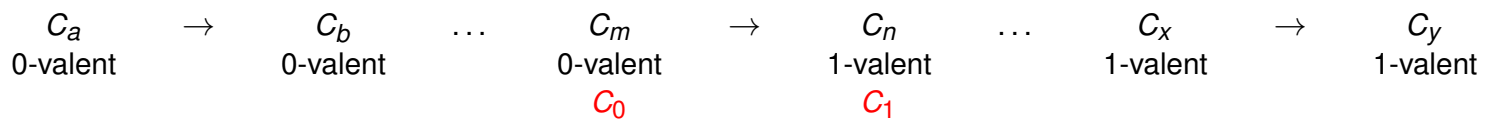
Supposons que $\mathcal{P}$ n'a pas de configuration initiale bivalente.

De part la correction partielle, $\mathcal{P}$ doit avoir à la fois des configurations initiales 0-valentes et 1-valentes.

Nous disons que deux configurations initiales sont adjacentes si elles diffèrent par exactement une valeur initiale v_p d'un seul processus p.

Toute paire de configurations initiales sont jointes par une chaîne de configurations initiales, chacune adjacente à la suivante : par exemple de $(0,0,0)$ à $(1,1,1)$ on a, entre autres :
$(0,0,0) \to (0,0,1) \to (0,1,1) \to (1,1,1)$

il existe nécessairement une configuration initiale 0-valente C_0 adjacente à une configuration initiale 1-valente C_1.



Preuve du lemme 2

Supposons que $\mathcal{P}$ n'a pas de configuration initiale bivalente.

De part la correction partielle, $\mathcal{P}$ doit avoir à la fois des configurations initiales 0-valentes et 1-valentes.

Nous disons que deux configurations initiales sont adjacentes si elles diffèrent par exactement une valeur initiale v_p d'un seul processus p.

Toute paire de configurations initiales sont jointes par une chaîne de configurations initiales, chacune adjacente à la suivante : par exemple de $(0,0,0)$ à $(1,1,1)$ on a, entre autres :
$(0,0,0) \rightarrow (0,0,1) \rightarrow (0,1,1) \rightarrow (1,1,1)$

il existe nécessairement une configuration initiale 0-valente C_0 adjacente à une configuration initiale 1-valente C_1.

$$\underset{\text{0-valent}}{C_a} \rightarrow \underset{\text{0-valent}}{C_b} \ldots \underset{\substack{\text{0-valent}\\ C_0}}{C_m} \rightarrow \underset{\substack{\text{1-valent}\\ C_1}}{C_n} \ldots \underset{\text{1-valent}}{C_x} \rightarrow \underset{\text{1-valent}}{C_y}$$

Soit p le processus dont la valeur initiale diffère dans C_0 et C_1.

Preuve du lemme 2 (suite)

Considérons maintenant une **exécution admissible décidante** depuis C_0 où p n'exécute aucune étape, et soit σ l'ordonnancement associé.

Preuve du lemme 2 (suite)

Considérons maintenant une **exécution admissible décidante** depuis C_0 où p n'exécute aucune étape, et soit σ l'ordonnancement associé.

σ peut aussi être appliqué à C_1.

Preuve du lemme 2 (suite)

Considérons maintenant une **exécution admissible décidante** depuis C_0 où p n'exécute aucune étape, et soit σ l'ordonnancement associé.

σ peut aussi être appliqué à C_1.

Les configurations correspondantes dans les deux exécutions sont identiques sauf pour l'état interne de p.

Preuve du lemme 2 (suite)

Considérons maintenant une **exécution admissible décidante** depuis C_0 où p n'exécute aucune étape, et soit σ l'ordonnancement associé.

σ peut aussi être appliqué à C_1.

Les configurations correspondantes dans les deux exécutions sont identiques sauf pour l'état interne de p.

Ces deux exécutions finissent par atteindre la même valeur de décision (p n'est pas impliqué dans la décision).

Preuve du lemme 2 (suite)

Considérons maintenant une **exécution admissible décidante** depuis C_0 où p n'exécute aucune étape, et soit σ l'ordonnancement associé.

σ peut aussi être appliqué à C_1.

Les configurations correspondantes dans les deux exécutions sont identiques sauf pour l'état interne de p.

Ces deux exécutions finissent par atteindre la même valeur de décision (p n'est pas impliqué dans la décision).

Si la valeur est 1, alors C_0 est bivalente, sinon C_1 est bivalente, contradiction. □

Résultat 3

Lemme 3

Soit C une configuration bivalente de $\mathcal{P}$, et soit $e = (p, m)$ un évènement applicable sur C.
Soit $\mathcal{C}$ l'ensemble des configurations atteignables depuis C sans appliquer e, et soit

$$\mathcal{D} = e(\mathcal{C}) = \{e(E) \mid E \in \mathcal{C} \text{ et } e \text{ est applicable sur } E\}.$$

Alors, $\mathcal{D}$ contient une configuration bivalente.

BUT : Montrer qu'il est toujours possible d'atteindre une configuration bivalente à partir d'une configuration bivalente, en retardant un évènement particulier.

Preuve du lemme 3

e est applicable sur C. Donc, par définition de $\mathcal{C}$ et le fait que les messages peuvent être retardés arbitrairement longtemps, e est applicable sur toutes les configurations E de $\mathcal{C}$. Donc,
$\mathcal{D} = e(\mathcal{C}) = \{e(E) \mid E \in \mathcal{C}\}$

Preuve du lemme 3

e est applicable sur C. Donc, par définition de $\mathcal{C}$ et le fait que les messages peuvent être retardés arbitrairement longtemps, e est applicable sur toutes les configurations E de $\mathcal{C}$. Donc, $\mathcal{D} = e(\mathcal{C}) = \{e(E) \mid E \in \mathcal{C}\}$

Supposons maintenant, par contradiction, que $\mathcal{D}$ ne contient aucune configuration bivalente. Donc, chaque configuration de $\mathcal{D}$ est univalente.

$$\begin{array}{lccccccc} \mathcal{C} = & C & \xrightarrow{e' \neq e} & C' = e'(C) & \xrightarrow{e'' \neq e} & C'' = e''(C') & \ldots & \\ & \downarrow e & & \downarrow e & & \downarrow e & & \\ \mathcal{D} = & e(C) & & e(C') & & e(C'') & \ldots & : \textbf{ univalentes} \end{array}$$

Preuve du lemme 3

Soit E_i une configuration i-valente atteignable depuis C, pour $i = 0, 1$ (à la fois E_0 et E_1 existent car C est bivalente).

Preuve du lemme 3

Soit E_i une configuration i-valente atteignable depuis C, pour $i = 0, 1$ (à la fois E_0 et E_1 existent car C est bivalente).

- Si $E_i \in \mathcal{C}$, posons $F_i = e(E_i) \in \mathcal{D}$.

$$\begin{array}{lccccccc} \mathcal{C} = & C & \xrightarrow{e' \neq e} & C' & \xrightarrow{e'' \neq e} & E_i = C'' & \dots & \\ & \downarrow e & & \downarrow e & & \downarrow e & & \\ \mathcal{D} = & e(C) & & e(C') & & F_i = e(C'') & \dots & \text{: univalentes} \end{array}$$

- Sinon, e a été appliqué pour atteindre E_i et donc il existe une configuration $F_i \in \mathcal{D}$ à partir de laquelle E_i est atteignable. Ex :

$$C \to C' \to C'' \to F_i = e(C'') \to \dots \to E_i$$

Preuve du lemme 3

Soit E_i une configuration i-valente atteignable depuis C, pour $i = 0, 1$ (à la fois E_0 et E_1 existent car C est bivalente).

- Si $E_i \in \mathcal{C}$, posons $F_i = e(E_i) \in \mathcal{D}$.

$$\begin{array}{lccccccc} \mathcal{C} = & C & \xrightarrow{e' \neq e} & C' & \xrightarrow{e'' \neq e} & E_i = C'' & \dots & \\ & \downarrow e & & \downarrow e & & \downarrow e & & \\ \mathcal{D} = & e(C) & & e(C') & & F_i = e(C'') & \dots & : \textbf{univalentes} \end{array}$$

- Sinon, e a été appliqué pour atteindre E_i et donc il existe une configuration $F_i \in \mathcal{D}$ à partir de laquelle E_i est atteignable. Ex :

$$C \to C' \to C'' \to F_i = e(C'') \to \dots \to E_i$$

Dans tous les cas, F_i est i-valente puisque F_i n'est pas bivalente ($F_i \in \mathcal{D}$ et $\mathcal{D}$ ne contient aucune configuration bivalente). Puisque $F_i \in \mathcal{D}$, pour $i = 0, 1$, $\mathcal{D}$ contient à la fois des configurations 0-valentes et 1-valentes.

Preuve du lemme 3

Nous disons maintenant que deux configurations sont voisines si l'une est résultante de l'autre en une seule étape.

Preuve du lemme 3

Nous disons maintenant que deux configurations sont voisines si l'une est résultante de l'autre en une seule étape.

Puisque par hypothèse, $\mathcal{D}$ ne contient que des configurations univalentes, si on applique e à C, on obtient une configuration univalente.

$$
\begin{array}{lccccccc}
\mathcal{C} = & C & \xrightarrow{e' \neq e} & C' & \xrightarrow{e'' \neq e} & C'' & \dots & \\
 & \downarrow e & & \downarrow e & & \downarrow e & & \\
\mathcal{D} = & e(C) & & e(C') & & e(C'') & \dots & : \text{ univalentes} \\
 & i\text{-valent} & & & & & &
\end{array}
$$

Preuve du lemme 3

Quelque soit la valeur de décision à laquelle amène $e(C)$, il existe une configuration C_x atteignable depuis C, depuis laquelle on obtient une configuration univalente pour l'autre valeur en appliquant e, d'après le point précédent.

$$
\begin{array}{lccccccccc}
\mathcal{C} = & C & \xrightarrow{e' \neq e} & C' & \xrightarrow{e'' \neq e} & C'' & \xrightarrow{e''' \neq e} & C_x = C''' & \dots & \\
 & \downarrow e & & \downarrow e & & \downarrow e & & \downarrow e & & \\
\mathcal{D} = & e(C) & & e(C') & & e(C'') & & e(C''') & \dots & : \quad \textbf{univalentes} \\
 & \textbf{i-valent} & & & & & & \textbf{j-valent} \ (j \neq i) & &
\end{array}
$$

Preuve du lemme 3

Considérons maintenant l'ordonnancement σ amenant de C à C_x.

Preuve du lemme 3

Considérons maintenant l'ordonnancement σ amenant de C à C_x.

Soit C_y la première configuration atteinte avec σ telle que $e(C_y)$ est univalente pour la même valeur que $e(C_x)$.

Preuve du lemme 3

Considérons maintenant l'ordonnancement σ amenant de C à C_x.

Soit C_y la première configuration atteinte avec σ telle que $e(C_y)$ est univalente pour la même valeur que $e(C_x)$.

Soit C_z la configuration qui précède C_y avec σ.

Preuve du lemme 3

Considérons maintenant l'ordonnancement σ amenant de C à C_x.

Soit C_y la première configuration atteinte avec σ telle que $e(C_y)$ est univalente pour la même valeur que $e(C_x)$.

Soit C_z la configuration qui précède C_y avec σ.

Donc C_y **et** C_z **sont voisines et** $e(C_y)$ **et** $e(C_z)$ **sont univalentes pour des valeurs différentes**.

$$
\begin{array}{lccccccccc}
\mathcal{C} = & C & \xrightarrow{e' \neq e} & C_z = C' & \xrightarrow{e'' \neq e} & C_y = C'' & \xrightarrow{e''' \neq e} & C_x = C''' & \dots & \\
 & \downarrow e & & \downarrow e & & \downarrow e & & \downarrow e & & \\
\mathcal{D} = & e(C) & & e(C') & & e(C'') & & e(C''') & \dots & : \quad \textbf{univalentes} \\
 & i\textbf{-valent} & & i\textbf{-valent} & & j\textbf{-valent} & & j\textbf{-valent } (j \neq i) & &
\end{array}
$$

Preuve du lemme 3

Posons $C_0, C_1 \in \mathcal{C}$ deux configurations voisines telles que $D_i = e(C_i)$ est i-valente pour $i = 0, 1$ (ces deux configurations existent d'après le point précédent).

Preuve du lemme 3

Posons $C_0, C_1 \in \mathcal{C}$ deux configurations voisines telles que $D_i = e(C_i)$ est i-valente pour $i = 0, 1$ (ces deux configurations existent d'après le point précédent).

Sans perte de générалité, posons $C_1 = e'(C_0)$, où $e' = (p', m')$.

$$\begin{array}{ccc} C_0 & \xrightarrow{e'} & C_1 \\ \downarrow e & & \downarrow e \\ D_0 & & D_1 \\ \text{0-valent} & & \text{1-valent} \end{array}$$

Preuve du lemme 3

Posons $C_0, C_1 \in \mathcal{C}$ deux configurations voisines telles que $D_i = e(C_i)$ est i-valente pour $i = 0, 1$ (ces deux configurations existent d'après le point précédent).

Sans perte de générалité, posons $C_1 = e'(C_0)$, où $e' = (p', m')$.

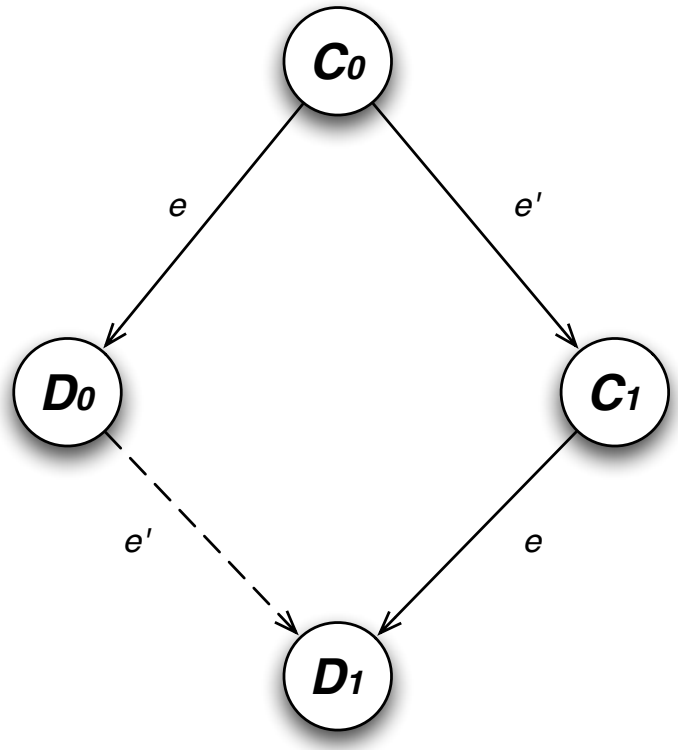
$$\begin{array}{ccc} C_0 & \xrightarrow{e'} & C_1 \\ \downarrow e & & \downarrow e \\ D_0 & & D_1 \\ \text{0-valent} & & \text{1-valent} \end{array}$$

Cas 1 : $p' \neq p$.

Cas 2 : $p' = p$.

Preuve du lemme 3, Cas 1 : $p' \neq p$

Si $p' \neq p$, alors $D_1 = e'(D_0)$ d'après le lemme 1. C'est impossible, car tout successeur d'une configuration 0-valente est par définition 0-valente.



Preuve du lemme 3, Cas 2 : $p' = p$

Considérons une exécution **décidante** finie depuis C_0 dans laquelle p n'exécute aucune étape. Soit σ l'ordonnancement correspondant, et soit $A = \sigma(C_0)$.

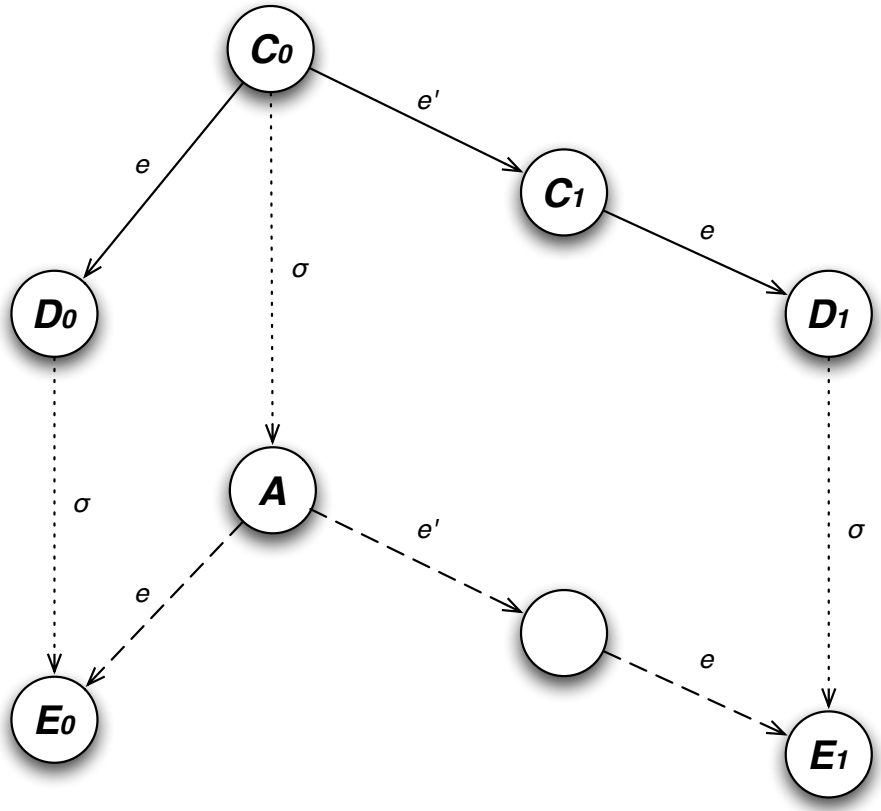
Preuve du lemme 3, Cas 2 : $p' = p$

Considérons une exécution **décidante** finie depuis C_0 dans laquelle p n'exécute aucune étape. Soit σ l'ordonnancement correspondant, et soit $A = \sigma(C_0)$.

Lemme 1 : σ est applicable à D_i, et mène à une configuration i-valente $E_i = \sigma(D_i), i = 0, 1$.

De plus, d'après le lemme 1, $e(A) = E_0$ et $e(e'(A)) = E_1$. D'où, A est bivalente.

Contradiction : l'exécution amenant à A est décidante, donc A doit être univalente.



Preuve du lemme 3

Dans chaque cas, nous obtenons une contradiction, donc $\mathcal{D}$ contient une configuration bivalente. □

Contradiction finale

Chaque exécution décidante démarrant d'une configuration bivalente mène vers une configuration univalente.

Contradiction finale

Chaque exécution décidante démarrant d'une configuration bivalente mène vers une configuration univalente.

Donc, il doit exister une certaine étape pour aller d'une configuration bivalente à une configuration univalente. Une telle étape détermine la valeur qui sera décidée.

Contradiction finale

Chaque exécution décidante démarrant d'une configuration bivalente mène vers une configuration univalente.

Donc, il doit exister une certaine étape pour aller d'une configuration bivalente à une configuration univalente. Une telle étape détermine la valeur qui sera décidée.

Nous montrons maintenant qu'il est toujours possible que le système s'exécute de telle manière qu'il évite toujours ce type d'étape, menant ainsi à une **exécution admissible non-décidante**.

Construction

Cette exécution se construit par blocs, à partir de la configuration initiale.

Construction

Cette exécution se construit par blocs, à partir de la configuration initiale.

Nous assurons que l'exécution est **admissible** de la manière suivante :

- une file d'attente de processus est maintenue (elle est initialement dans un ordre quelconque), et
- le tampon-mémoire de messages dans une configuration est ordonné en fonction des temps auxquels les messages ont été envoyés, les plus anciens en premier.

Construction

Chaque bloc consiste en une à plusieurs étapes.

Construction

Chaque bloc consiste en une à plusieurs étapes.

Le bloc courant termine lorsque le processus en tête de la file de processus exécute une étape de calcul dans laquelle si il y avait des messages pour lui dans le tampon-mémoire de messages au début du bloc, le plus ancien a été reçu.

Le processus est alors déplacé en queue de file.

Construction

Dans toute suite infinie de tels blocs, chaque processus exécute une infinité d'étape et reçoit tous les messages qu'on lui a envoyés. Ainsi, on obtient une exécution admissible.

Construction

Dans toute suite infinie de tels blocs, chaque processus exécute une infinité d'étape et reçoit tous les messages qu'on lui a envoyés. Ainsi, on obtient une exécution admissible.

Le problème, bien sûr, est de construire une telle exécution en évitant toujours qu'une décision soit prise.

Construction

Soit C_0 une configuration initiale bivalente (une telle configuration existe, d'après le lemme 2).

Construction

Soit C_0 une configuration initiale bivalente (une telle configuration existe, d'après le lemme 2).

L'exécution commence en C_0, et nous assurons que tout bloc commence dans une configuration bivalente.

Construction

Soit C_0 une configuration initiale bivalente (une telle configuration existe, d'après le lemme 2).

L'exécution commence en C_0, et nous assurons que tout bloc commence dans une configuration bivalente.

Considérons une configuration bivalente C où le processus p est en tête de la file de priorités.

Construction

Soit C_0 une configuration initiale bivalente (une telle configuration existe, d'après le lemme 2).

L'exécution commence en C_0, et nous assurons que tout bloc commence dans une configuration bivalente.

Considérons une configuration bivalente C où le processus p est en tête de la file de priorités.

Soit m le message le plus ancien de p dans le tampon-mémoire de message, si un tel message existe, sinon posons $m = \emptyset$.

Construction

Soit C_0 une configuration initiale bivalente (une telle configuration existe, d'après le lemme 2).

L'exécution commence en C_0, et nous assurons que tout bloc commence dans une configuration bivalente.

Considérons une configuration bivalente C où le processus p est en tête de la file de priorités.

Soit m le message le plus ancien de p dans le tampon-mémoire de message, si un tel message existe, sinon posons $m = \emptyset$.

Soit $e = (p, m)$. D'après le lemme 3, il existe une configuration bivalente C' atteignable depuis C avec un ordonnancement où e est le dernier évènement appliqué. La suite de configurations correspondante définit le bloc.

Conclusion

Puisque chaque bloc finit dans une configuration bivalente, nous pouvons construire un ordonnancement infini.

Conclusion

Puisque chaque bloc finit dans une configuration bivalente, nous pouvons construire un ordonnancement infini.

De plus, par construction, l'exécution résultant de cet ordonnancement est **admissible** et **aucune décision n'est jamais atteinte**.

Conclusion

Puisque chaque bloc finit dans une configuration bivalente, nous pouvons construire un ordonnancement infini.

De plus, par construction, l'exécution résultant de cet ordonnancement est **admissible** et **aucune décision n'est jamais atteinte**.

Ainsi, nous obtenons la contradiction et nous concluons avec le théorème suivant :

Théorème 1

Il n'existe pas d'algorithme déterministe de consensus qui est correct en dépit d'une faute.