Introduction à la tolérance aux fautes

Alain Cournier Stéphane Devismes

Université de Picardie Jules Verne

17 janvier 2025

Préambule

Trois objectifs :

1. Définir les fautes dans les systèmes distribués
2. Définir la tolérance aux fautes
3. Comprendre que la tolérance aux fautes c'est dur !

Plan

Les fautes

Plan

Qu'est-ce qu'une faute ?

« une faute provoque une erreur qui entraîne une défaillance ».

Défaillances/pannes

On parle de la défaillance d'un composant (ou d'un système) lorsque son comportement n'est plus conforme à sa spécification.

Composant du réseau = lien de communication ou nœud

(nœud = processus, machine . . .)

Défaillances/pannes

On parle de la défaillance d'un composant (ou d'un système) lorsque son comportement n'est plus conforme à sa spécification.

Composant du réseau = lien de communication ou nœud

(nœud = processus, machine . . .)

On parlera de nœud correct (resp. lien fiable) lorsque le nœud (resp. le lien) ne subit pas de défaillance.

Défaillances/pannes

On parle de la défaillance d'un composant (ou d'un système) lorsque son comportement n'est plus conforme à sa spécification.

Composant du réseau = lien de communication ou nœud

(nœud = processus, machine . . .)

On parlera de nœud correct (resp. lien fiable) lorsque le nœud (resp. le lien) ne subit pas de défaillance.

Exemples :

- Un **processus** arrête d'exécuter un programme.
- Un **lien de communication** perd un message.

Lien fiable

La spécification d'un lien fiable consiste en la conjonction des trois propriétés suivantes :

Pas de création : Tout message reçu par un processus p venant du processus q a été envoyé au préalable par q à p.

Pas de duplication : Tout message est reçu au plus une fois.

Pas de perte : Tout message envoyé est livré au récepteur en temps fini.

Erreurs

Une erreur est un état du système à partir duquel la poursuite de l'exécution est susceptible de conduire à une défaillance.

Erreurs

Une erreur est un état du système à partir duquel la poursuite de l'exécution est susceptible de conduire à une défaillance.

Des exemples de telles situations sont :

- un chaînage d'une liste chainée corrompu ou un pointeur non initialisé : il s'agit ici d'erreurs logicielles ;
- un câble du réseau déconnecté ou une unité disque éteinte : il s'agit ici d'erreurs matérielles.

Fautes

Une faute est un événement ayant entrainé une erreur.

Fautes

Une faute est un événement ayant entrainé une erreur.

Il peut s'agir

- d'une faute de programmation (pour les erreurs logicielles)

Fautes

Une faute est un événement ayant entrainé une erreur.

Il peut s'agir

- d'une faute de programmation (pour les erreurs logicielles) ou
- d'événements physiques, *e.g.*, usure, malveillance, catastrophe, . . .(dans le cas d'erreurs matérielles).

Fautes

Une faute est un événement ayant entrainé une erreur.

Il peut s'agir

- d'une faute de programmation (pour les erreurs logicielles) ou
- d'événements physiques, *e.g.*, usure, malveillance, catastrophe, . . .(dans le cas d'erreurs matérielles).

Dans le cadre de ce cours, nous ne considérerons que des fautes matérielles, *e.g.*, coupure d'un lien, perturbation électro-magnétique du signal dans un lien . . .

Faute, erreur, défaillance

La différence est subtile !

Faute, erreur, défaillance

La différence est subtile !

Dans la suite de ce cours, ces trois termes seront considérés comme synonymes.

Fautes dans les réseaux

Les réseaux sont naturellement sujets aux fautes.

Fautes dans les réseaux

Les réseaux sont naturellement sujets aux fautes.

Plus le nombre de processus est important, plus la probabilité qu'un composant du réseau subisse une faute durant l'exécution d'un protocole est importante.

Fautes dans les réseaux

Les réseaux sont naturellement sujets aux fautes.

Plus le nombre de processus est important, plus la probabilité qu'un composant du réseau subisse une faute durant l'exécution d'un protocole est importante.

Par exemple, si on considère le réseau internet (350 millions de serveurs en 2006), il est impossible d'imaginer qu'un tel réseau puisse fonctionner ne serait-ce qu'une heure sans subir la moindre faute !

Fautes dans les réseaux

Les réseaux sont naturellement sujets aux fautes.

Plus le nombre de processus est important, plus la probabilité qu'un composant du réseau subisse une faute durant l'exécution d'un protocole est importante.

Par exemple, si on considère le réseau internet (350 millions de serveurs en 2006), il est impossible d'imaginer qu'un tel réseau puisse fonctionner ne serait-ce qu'une heure sans subir la moindre faute !

Il faut aussi noter que la plupart des réseaux actuels sont constitués de machines « grand public » produites en grand nombre à prix réduit : d'où, un risque de défaut plus important.

Plan

Critères pour la classification des fautes

Les fautes sont généralement classées suivant différents critères :

- L'origine de la faute.
- La cause de la faute.
- La durée de la faute.
- La détectabilité de la faute.

L'origine de la faute

Le type de composant qui est responsable de la faute : lien de communication ou nœud.

La cause de la faute

La faute peut être

- bénigne : non volontaire, *e.g.*, due à un problème matériel, ou
- maligne : due à une intention (malveillante ou malicieuse) extérieure au système.

La durée de la faute

- Si la durée d'une faute est supérieure au temps restant de l'exécution du protocole, elle est dite définitive ou franche,
- Sinon elle est dite transitoire ou intermittente.

La durée de la faute

- Si la durée d'une faute est supérieure au temps restant de l'exécution du protocole, elle est dite définitive ou franche,
- Sinon elle est dite transitoire ou intermittente.

La différence entre transitoire et intermittente est définie par la fréquence.

- Dans le premier cas, elle se produit de manière isolée, c'est-à-dire rarement (en moyenne une fois pendant le temps d'exécution du protocole).
- Dans le second cas, elle se produit régulièrement

La détectabilité de la faute

Une faute est détectable si son incidence sur la cohérence de l'état d'un processus permet à celui-ci de s'en apercevoir.

Plan

Panne crash

Panne franche d'un processus.

Le processus cesse définitivement de faire des pas de calculs.

Perte intermittente de messages

Régulièrement un lien perd des messages.

Perte intermittente de messages

Régulièrement un lien perd des messages.

Deux hypothèses sont généralement utilisées dans ce cas :

- Soit on suppose que les pertes sont équitables,
- Soit on suppose qu'il y a un taux de pertes de message (connu ou inconnu des processus).

Perte intermittente de messages

Régulièrement un lien perd des messages.

Deux hypothèses sont généralement utilisées dans ce cas :

- Soit on suppose que les pertes sont équitables,
- Soit on suppose qu'il y a un taux de pertes de message (connu ou inconnu des processus).

Lorsque les pertes sont équitables, le lien de commmunication vérifie l'hypothèse suivante : si des messages sont envoyés infiniment souvent, alors une infinité de messages est livrée.

Perte intermittente de messages

Régulièrement un lien perd des messages.

Deux hypothèses sont généralement utilisées dans ce cas :

- Soit on suppose que les pertes sont équitables,
- Soit on suppose qu'il y a un taux de pertes de message (connu ou inconnu des processus).

Lorsque les pertes sont équitables, le lien de commmunication vérifie l'hypothèse suivante : si des messages sont envoyés infiniment souvent, alors une infinité de messages est livrée.

On parle aussi de fautes par omission : à divers instants de l'exécution, un composant du réseau omet de communiquer avec un autre en réception ou en émission.

Faute transitoire

C'est un comportement erroné d'un à plusieurs composants du réseau durant une certaine période (finie).

Une fois que cette période est passée, les composants reprennent un comportement correct. Cependant, l'état du système est perturbé.

Faute transitoire

C'est un comportement erroné d'un à plusieurs composants du réseau durant une certaine période (finie).

Une fois que cette période est passée, les composants reprennent un comportement correct. Cependant, l'état du système est perturbé.

Le système **subit les effets de la faute**, *e.g.*, certains messages ont été corrompus.

Faute transitoire

C'est un comportement erroné d'un à plusieurs composants du réseau durant une certaine période (finie).

Une fois que cette période est passée, les composants reprennent un comportement correct. Cependant, l'état du système est perturbé.

Le système **subit les effets de la faute**, *e.g.*, certains messages ont été corrompus.

On parlera alors de fautes d'état : des composants du système ont subi un changement d'état non prévu par l'algorithme.

Par exemple, cela peut être des corruptions de mémoires locales de processus ou de contenus de message, ou encore de la duplication de messages.

Fautes byzantines

Les fautes byzantines sont dues à des processus byzantins qui ont un comportement arbitraire, ne suivant plus (nécessairement) le code de leurs algorithmes locaux.

Fautes byzantines

Les fautes byzantines sont dues à des processus byzantins qui ont un comportement arbitraire, ne suivant plus (nécessairement) le code de leurs algorithmes locaux.

Cela peut être dû à une erreur matérielle, un virus ou la corruption du code de l'algorithme.

La tolérance aux fautes

Plan

Idée

Les réseaux modernes sont à grande-échelle et fait de machines hétérogènes et produites en masses à faible coût, *e.g.*

Idée

Les réseaux modernes sont à grande-échelle et fait de machines hétérogènes et produites en masses à faible coût, *e.g.*

- **Internet**
  - 17,6 milliard d'objets connectés en 2016
  - Internet des objets

Idée

Les réseaux modernes sont à grande-échelle et fait de machines hétérogènes et produites en masses à faible coût, *e.g.*

- **Internet**
  - 17,6 milliard d'objets connectés en 2016
  - Internet des objets
- **Réseaux sans fils**
  - Communication radio : beaucoup de pertes de messages
  - Crash de machines à cause des batteries limitées

Idée

Les réseaux modernes sont à grande-échelle et fait de machines hétérogènes et produites en masses à faible coût, *e.g.*

- **Internet**
  - 17,6 milliard d'objets connectés en 2016
  - Internet des objets
- **Réseaux sans fils**
  - Communication radio : beaucoup de pertes de messages
  - Crash de machines à cause des batteries limitées

$\Rightarrow$ Forte probabilité de pannes

$\Rightarrow$ Intervention humain impossible ou au moins non-souhaitable

Idée

Les réseaux modernes sont à grande-échelle et fait de machines hétérogènes et produites en masses à faible coût, *e.g.*

- **Internet**
  - 17,6 milliard d'objets connectés en 2016
  - Internet des objets
- **Réseaux sans fils**
  - Communication radio : beaucoup de pertes de messages
  - Crash de machines à cause des batteries limitées

$\Rightarrow$ Forte probabilité de pannes

$\Rightarrow$ Intervention humain impossible ou au moins non-souhaitable

$\Rightarrow$ Besoin de tolérance aux fautes, *i.e.*, une prise en compte automatique de la possibilité de l'arrivée de fautes au niveau algorithmique.

Objectif

L'objectif principal de la tolérance aux fautes est d'éviter de réinitialiser le réseau après chaque panne.

Ainsi, la tolérance aux fautes qualifie l'aptitude d'un système à résister à ou récupérer des fautes sans intervention extérieure (humaine par exemple).

Plan

Approches pour la tolérance aux fautes

Deux catégories principales :

- Les algorithmes robustes.
- Les algorithmes (auto)stabilisants.

Algorithmes robustes

Ils abordent le problème de tolérance aux fautes selon une approche pessimiste où les processus suspectent toutes les informations qu'ils recoivent.

Algorithmes robustes

Ils abordent le problème de tolérance aux fautes selon une approche pessimiste où les processus suspectent toutes les informations qu'ils recoivent.

Le but est de « masquer » l'effet des pannes à l'utilisateur : on garantit toujours la spécification de l'algorithme en dépit des pannes.

Les algorithmes (auto)stabilisants

Ils abordent le problème selon une approche optimiste, on fait confiance au système mais si on détecte un dysfonctionnement, on le corrige.

Les algorithmes (auto)stabilisants

Ils abordent le problème selon une approche optimiste, on fait confiance au système mais si on détecte un dysfonctionnement, on le corrige.

Le système (en particulier l'utilisateur) subit l'effet des pannes : cela cause un comportement anormal de processus, parfois même non-défaillants, mais garantit le retour vers un comportement global normal en un temps fini après que les fautes ont cessé, c'est une approche « non-masquante ».

Les algorithmes (auto)stabilisants

Ils abordent le problème selon une approche optimiste, on fait confiance au système mais si on détecte un dysfonctionnement, on le corrige.

Le système (en particulier l'utilisateur) subit l'effet des pannes : cela cause un comportement anormal de processus, parfois même non-défaillants, mais garantit le retour vers un comportement global normal en un temps fini après que les fautes ont cessé, c'est une approche « non-masquante ».

L'autostabilisation est considérée comme lightweight (*i.e.*, à surcoût faible) par rapport à l'approche robuste.

Impossibilité du consensus asynchrone avec 0 ou 1 crash

Plan

Le résultat (FLP'85)

« Il est impossible de résoudre de manière déterministe le consensus (binaire) dans un système asynchrone où **au plus un** processus peut être défaillant (un crash) ».

Fisher, Lynch et Paterson (1985)

Le résultat (FLP'85)

« Il est impossible de résoudre de manière déterministe le consensus (binaire) dans un système asynchrone où **au plus un** processus peut être défaillant (un crash) ».

Fisher, Lynch et Paterson (1985)

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

Le résultat (FLP'85)

« Il est impossible de résoudre de manière déterministe le consensus (binaire) dans un système asynchrone où **au plus un** processus peut être défaillant (un crash) ».

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 résultat (FLP'85)

« Il est impossible de résoudre de manière déterministe le consensus (binaire) dans un système asynchrone où **au plus un** processus peut être défaillant (un crash) ».

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.

N.b., ce résultat concerne uniquement l'approche **robuste**.

Le consensus (binaire)

Multi-initiateurs

Le consensus (binaire)

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$, **une constante**

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

Le consensus (binaire)

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$, **une constante**

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 (binaire)

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$, **une constante**

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 :

Intégrité : Tout processus décide au plus une fois.

Le consensus (binaire)

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$, **une constante**

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 :

Intégrité : Tout processus décide au plus une fois.

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

Le consensus (binaire)

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$, **une constante**

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 :

Intégrité : Tout processus décide au plus une fois.

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 (binaire)

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$, **une constante**

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 :

Intégrité : Tout processus décide au plus une fois.

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 (binaire)

Multi-initiateurs

Pour tout processus p

Entrée : $v_p \in \{0,1\}$, **une constante**

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 :

Intégrité : Tout processus décide au plus une fois.

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.

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 **connexe** : facile !

Le consensus dans les systèmes sans pannes

- Élection de leader (multi-initiateurs).

Le consensus dans les systèmes sans pannes

- Élection de leader (multi-initiateurs).
- Le leader collecte les propositions en utilisant une propagation d'information avec retour.

Le consensus dans les systèmes sans pannes

- Élection de leader (multi-initiateurs).
- Le leader collecte les propositions en utilisant une propagation d'information avec retour.
- Le leader décide en fonction de toutes les propositions et diffuse sa décision (diffusion simple).
- Les autres processus décident comme le leader à la réception de sa décision.

Le consensus dans les systèmes sans pannes

- Élection de leader (multi-initiateurs).
- Le leader collecte les propositions en utilisant une propagation d'information avec retour.
- Le leader décide en fonction de toutes les propositions et diffuse sa décision (diffusion simple).
- Les autres processus décident comme le leader à la réception de sa décision.

Toute règle d'intégrité déterministe vérifiant la validité est valable.

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 (binaire) est le **problème d'accord** le plus simple.

   (accord sur deux valeurs booléennes)

Autres problèmes d'accord : le consensus multi-valué, l'élection, le registre partagé, la diffusion atomique, la duplication de machine d'état, la synchronisation, ...

Les problèmes d'accord sont omniprésents en système distribué. (base de donnée, allocation de ressource, ...)

Aspect fondamental du résultat

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

   (accord sur deux valeurs booléennes)

   Autres problèmes d'accord : le consensus multi-valué, l'élection, le registre partagé, la diffusion atomique, la duplication de machine d'état, la synchronisation, ...

   Les problèmes d'accord sont omniprésents en système distribué. (base de donnée, allocation de ressource, ...)

2. L'impossibilité est obtenue malgré des **hypothèses très fortes** sur le système : canaux fiables, au plus une panne, réseau complet, les processus ont des identifiants uniques, ...

Plan

Remarque préliminaire

Le modèle doit être plutôt fort afin de rendre la preuve d'impossibilité aussi largement applicable que 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 envoyé 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 envoyé 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 identifiés et au moins $n - 1$ sont corrects.

  Un processus est correct s'il exécute une infinité de pas de calculs, sinon il est défaillant, c'est-à-dire qu'il va tomber en panne. Dans ce cas, il s'agit d'une panne **crash** définitive.

  Un processus défaillant n'exécute qu'un nombre fini de pas de calculs.

  Au plus un processus peut tomber en panne.

Processus

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

  Un processus est correct s'il exécute une infinité de pas de calculs, sinon il est défaillant, c'est-à-dire qu'il va tomber en panne. Dans ce cas, il s'agit d'une panne **crash** définitive.

  Un processus défaillant n'exécute qu'un nombre fini de pas de calculs.

  Au plus un processus peut tomber en panne.

- Les processus n'ont aucun accès à une horloge globale.

Processus

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

  Un processus est correct s'il exécute une infinité de pas de calculs, sinon il est défaillant, c'est-à-dire qu'il va tomber en panne. Dans ce cas, il s'agit d'une panne **crash** définitive.

  Un processus défaillant n'exécute qu'un nombre fini de pas de calculs.

  Au plus un processus peut tomber en panne.

- 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 $n \geq 2$ processus identifiés et au moins $n - 1$ sont corrects.

  Un processus est correct s'il exécute une infinité de pas de calculs, sinon il est défaillant, c'est-à-dire qu'il va tomber en panne. Dans ce cas, il s'agit d'une panne **crash** définitive.

  Un processus défaillant n'exécute qu'un nombre fini de pas de calculs.

  Au plus un processus peut tomber en panne.

- 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 $n \geq 2$ processus identifiés et au moins $n - 1$ sont corrects.

  Un processus est correct s'il exécute une infinité de pas de calculs, sinon il est défaillant, c'est-à-dire qu'il va tomber en panne. Dans ce cas, il s'agit d'une panne **crash** définitive.

  Un processus défaillant n'exécute qu'un nombre fini de pas de calculs.

  Au plus un processus peut tomber en panne.

- 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 $n \geq 2$ processus identifiés et au moins $n - 1$ sont corrects.

  Un processus est correct s'il exécute une infinité de pas de calculs, sinon il est défaillant, c'est-à-dire qu'il va tomber en panne. Dans ce cas, il s'agit d'une panne **crash** définitive.

  Un processus défaillant n'exécute qu'un nombre fini de pas de calculs.

  Au plus un processus peut tomber en panne.

- 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 $n \geq 2$ processus identifiés et au moins $n-1$ sont corrects.

  Un processus est correct s'il exécute une infinité de pas de calculs, sinon il est défaillant, c'est-à-dire qu'il va tomber en panne. Dans ce cas, il s'agit d'une panne **crash** définitive.

  Un processus défaillant n'exécute qu'un nombre fini de pas de calculs.

  Au plus un processus peut tomber en panne.

- 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 messages** (le réseau est complet).

Exécution d'un pas de calcul

Étape atomique.

Exécution d'un pas de calcul

Étape atomique.

En une étape, un processus :

- Essaie de recevoir un message,
- Fait 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)
- Envoie un nombre quelconque mais fini de messages aux autres processus.

Exécution d'un pas de calcul

Étape atomique.

En une étape, un processus :

- Essaie de recevoir un message,
- Fait 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)
- Envoie un nombre quelconque mais fini de messages aux autres processus.

Rappel : par définition, tout processus correct exécute une infinité d'étapes atomiques [1].

1. Ainsi, tout message envoyé à un processus correct finit par être reçu par ce dernier.

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**.

Encore plus général : le consensus faible

Pour tout processus p

Entrée : $v_p \in \{0,1\}$, **une constante**

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

p doit décider une valeur booléenne dans d_p en respectant les conditions suivantes :

Intégrité : Tout processus décide au plus une fois.

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

Encore plus général : le consensus faible

Pour tout processus p

Entrée : $v_p \in \{0,1\}$, **une constante**

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

p doit décider une valeur booléenne dans d_p en respectant les conditions suivantes :

Intégrité : Tout processus décide au plus une fois.

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)

Encore plus général : le consensus faible

Pour tout processus p

Entrée : $v_p \in \{0,1\}$, **une constante**

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

p doit décider une valeur booléenne dans d_p en respectant les conditions suivantes :

Intégrité : Tout processus décide au plus une fois.

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

Algorithme de consensus

Soit $\mathcal{P}$ un algorithme de consensus faible.
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**.

États

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

États

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

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

États

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

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

Il y a **deux états internes initiaux** possibles par processus, l'un où l'entrée vaut 0 et l'autre où l'entrée vaut 1.

États

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

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

Il y a **deux états internes 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).

États

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

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

Il y a **deux états internes 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).

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

États : exemple

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

États : exemple

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

Exemple d'état interne :

$$\langle v_p = 0, d_p = \perp, x_p = 10, CP_p = 0x4040 \rangle$$

États : exemple

Supposons un 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 :

$$\langle v_p = 0, d_p = \perp, x_p = 0, CP_p = 0x4000 \rangle$$

États : exemple

Supposons un 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 :

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

Algorithme local = Fonction de transition

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

mêmes entrées (état, et message ou absence de message reçu) $\Rightarrow$ **mêmes sorties** (état et envoi de messages éventuel).

Algorithme local = Fonction de transition

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

mêmes entrées (état, et message ou absence de message reçu) $\Rightarrow$ **mêmes sorties** (état et envoi de messages éventuel).

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 [2], appelé tampon-mémoire, où sont gardés les messages envoyés non encore reçus.

2. Pas d'ordre car communications ne sont pas FIFO !

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 [2], 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).

2. Pas d'ordre car communications ne sont pas FIFO !

Non-déterminisme

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

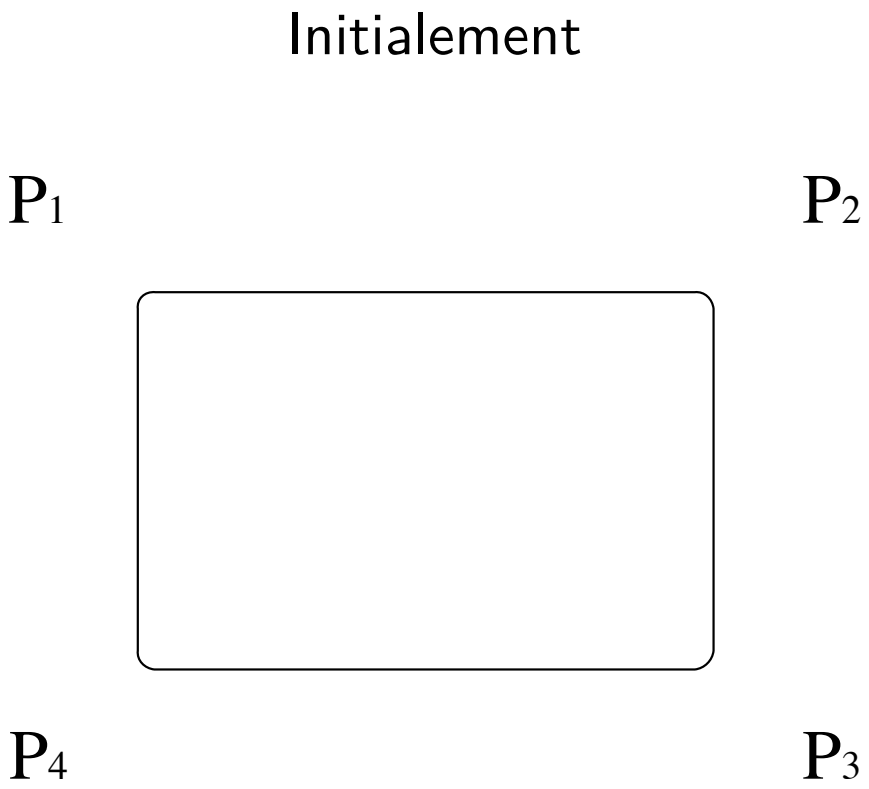
Non-déterminisme

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.

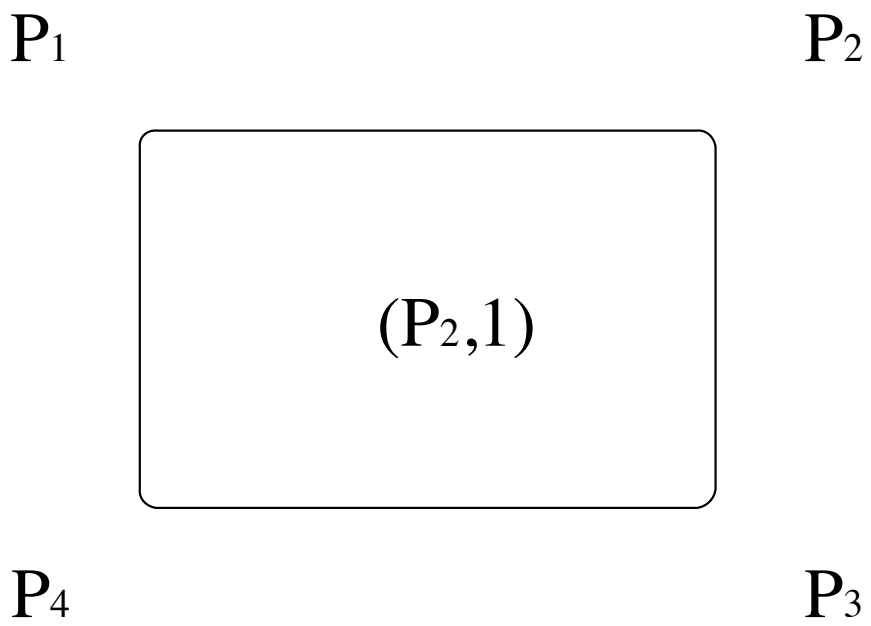
En particulier, le système de messages peut retourner $\emptyset$ **un nombre fini de fois en réponse de l'appel** `reçoit(`p`)`**, bien qu'un message** (p, m) **soit présent dans le tampon-mémoire**. (Asynchronisme)

Exemple



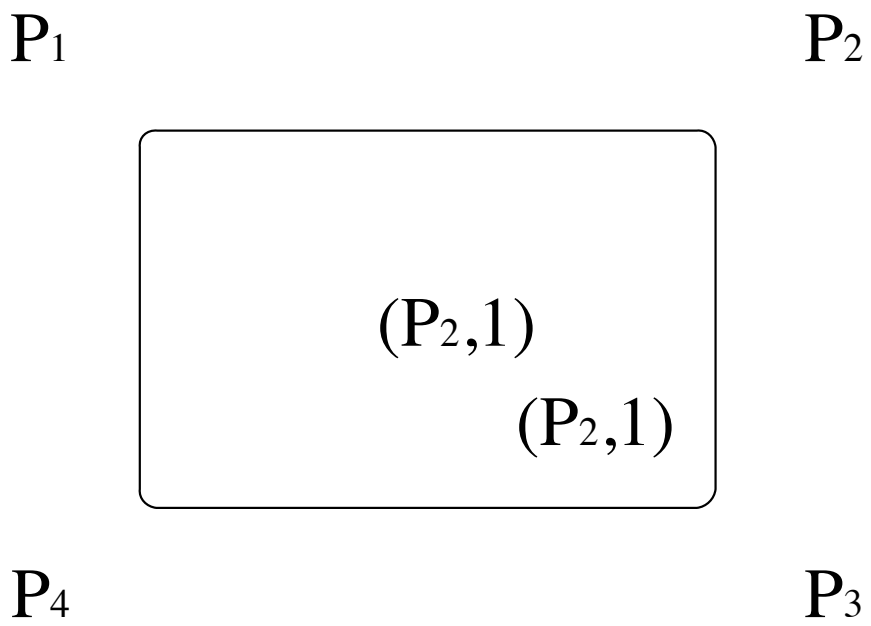
Exemple

P_1 exécute `envoi(`P_2`,1)`



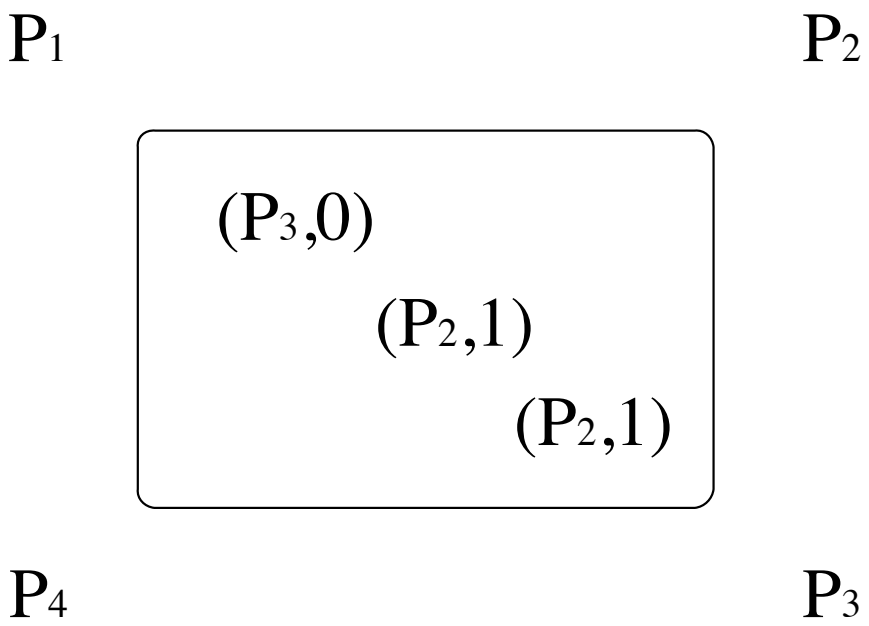
Exemple

P_3 exécute `envoi(`P_2`,1)`



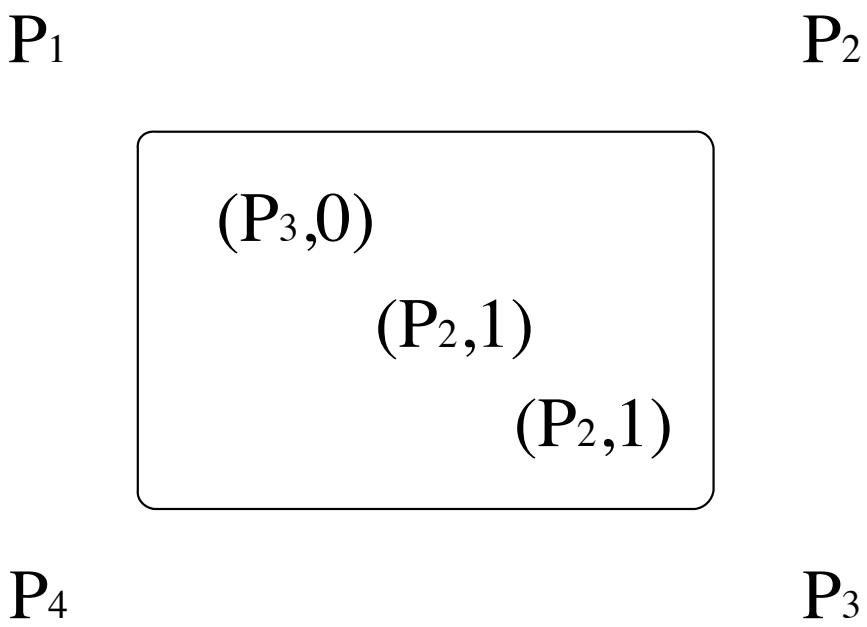
Exemple

P_1 exécute `envoi(`P_3`,0)`



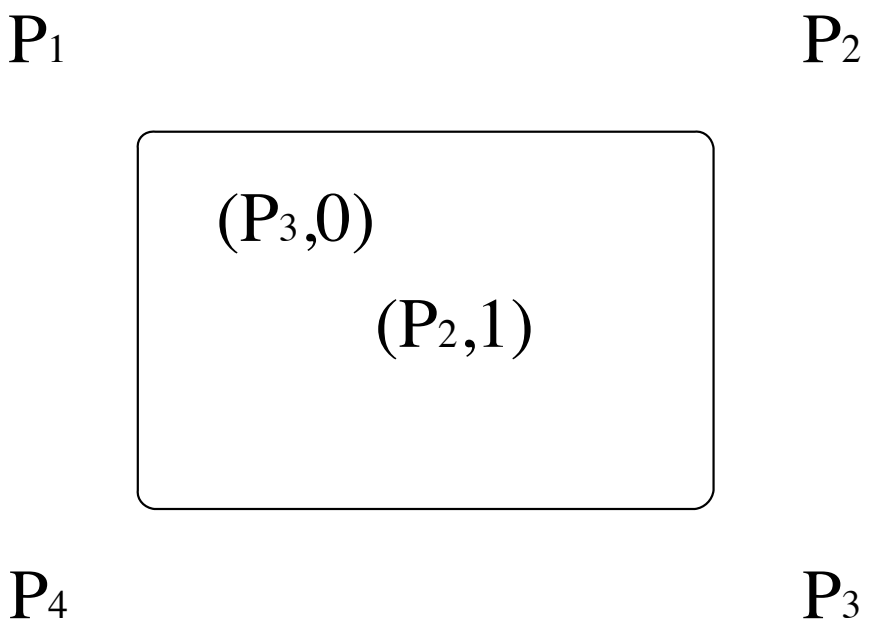
Exemple

`reçoit(`P_1`)` retourne $\emptyset$ à P_1



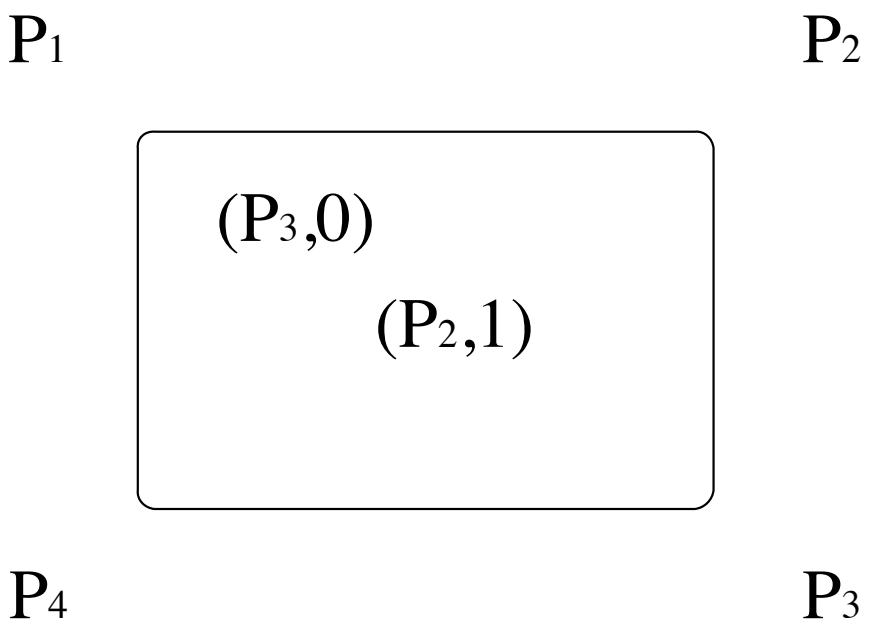
Exemple

`reçoit(`P_2`)` retourne 1 à P_2



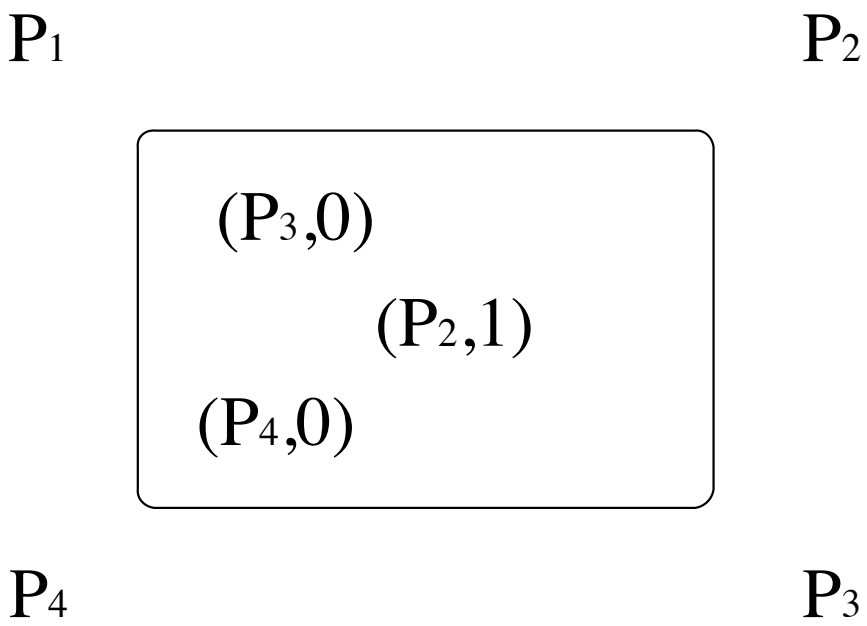
Exemple

`reçoit(`P_2`)` retourne $\emptyset$ à P_2



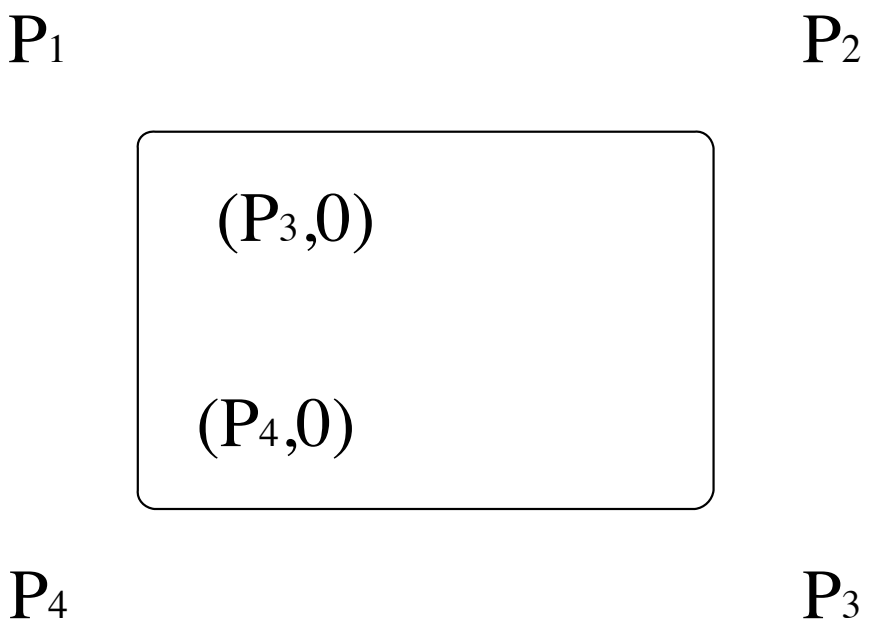
Exemple

P_3 exécute `envoi(`P_4`,0)`



Exemple

`reçoit(`P_2`)` retourne 1 à P_2



Configurations

Configuration : état interne de chaque processus
+
Contenu du tampon-mémoire de message.

Configurations

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 seule 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 seule 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 seule 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 :

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

Étape

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

Étape

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 $m \in M \cup \{\emptyset\}$.

Étape

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.

Étape

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

$\rightarrow$ e peut-être vu comme « p reçoit m ».

Étape

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

$\rightarrow$ e peut-être vu comme « p reçoit m ».

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

Évènement applicable

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

Évènement applicable

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

L'évènement (p, m) avec $m \in M$ peut être appliqué sur configuration C seulement si dans la configuration C le tampon-mémoire contient (p, m).

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.

Soit $\sigma = e_0, e_1, e_2, e_3, e_4, e_5, e_6,$ où

- $e_0 = (p_3, \emptyset)$,
- $e_1 = (p_1, \emptyset)$,
- $e_2 = (p_1, m_a)$,
- $e_3 = (p_1, \emptyset)$,
- $e_4 = (p_2, m_a)$,
- $e_5 = (p_1, m_a)$ et
- $e_6 = (p_3, m_b)$.

Exemple : Si le tampon-mémoire de messages dans C contient (p_1, m_a), (p_2, m_a), (p_1, m_a) et p_2 envoie m_b à p_3 sur réception de m_a, alors σ est un ordonnancement possible depuis C.

Exécution

La suite de configurations associée à l'application séquentielle d'un ordonnancement depuis une configuration est appelée exécution.

Exécution

La suite de configurations associée à l'application séquentielle d'un ordonnancement depuis une configuration est appelée exécution.

Exemple : avec l'ordonnancement $\sigma = e_0, e_1, e_2, e_3, e_4, e_5, e_6$ depuis la configuration C_0, on obtient l'exécution :

$$C_0, C_1, C_2, C_3, C_4, C_5, C_6, C_7$$

où

- $C_1 = e_0(C_0)$,
- $C_2 = e_1(C_1)$,
- $C_3 = e_2(C_2)$,
- $C_4 = e_3(C_3)$,
- $C_5 = e_4(C_4)$,
- $C_6 = e_5(C_5)$ et
- $C_7 = e_6(C_6)$.

Atteignabilité

Si σ est fini, alors nous notons $\sigma(C_0)$ la configuration obtenue en appliquant séquentiellement σ depuis C_0.

La configuration $\sigma(C_0)$ est dite atteignable depuis C_0.

Atteignabilité

Si σ est fini, alors nous notons $\sigma(C_0)$ la configuration obtenue en appliquant séquentiellement σ depuis C_0.

La configuration $\sigma(C_0)$ est dite atteignable depuis C_0.

Dans l'exemple précédent, l'exécution de σ depuis C_0 mène à C_7 :

$$\sigma(C_0) = C_7$$

Donc, C_7 est atteignable depuis C_0.

Atteignabilité

Si σ est fini, alors nous notons $\sigma(C_0)$ la configuration obtenue en appliquant séquentiellement σ depuis C_0.

La configuration $\sigma(C_0)$ est dite atteignable depuis C_0.

Dans l'exemple précédent, l'exécution de σ depuis C_0 mène à C_7 :

$$\sigma(C_0) = C_7$$

Donc, C_7 est atteignable depuis C_0.

Une configuration atteignable depuis une configuration initiale est dite accessible.

Si C_0 est une configuration initiale, alors C_7 est accessible.

Atteignabilité

Si σ est fini, alors nous notons $\sigma(C_0)$ la configuration obtenue en appliquant séquentiellement σ depuis C_0.

La configuration $\sigma(C_0)$ est dite atteignable depuis C_0.

Dans l'exemple précédent, l'exécution de σ depuis C_0 mène à C_7 :

$$\sigma(C_0) = C_7$$

Donc, C_7 est atteignable depuis C_0.

Une configuration atteignable depuis une configuration initiale est dite accessible.

Si C_0 est une configuration initiale, alors C_7 est 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)
2. Pour chaque valeur $v \in \{0, 1\}$, **au moins une configuration accessible** a une valeur de décision v. (Validité faible)

(*N.b.*, l'intégrité est assurée par définition de la variable de sortie, qui ne peut être écrite qu'une seule fois)

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)
2. Pour chaque valeur $v \in \{0, 1\}$, **au moins une configuration accessible** a une valeur de décision v. (Validité faible)

(*N.b.*, l'intégrité est assurée par définition de la variable de sortie, qui ne peut être écrite qu'une seule fois)

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 reçus**.

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)
2. Pour chaque valeur $v \in \{0, 1\}$, **au moins une configuration accessible** a une valeur de décision v. (Validité faible)

(*N.b.*, l'intégrité est assurée par définition de la variable de sortie, qui ne peut être écrite qu'une seule fois)

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 reçus**.

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

Spécification

Un algorithme de consensus $\mathcal{P}$ est correct en dépit d'au plus 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 depuis 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.

Premier résultat : commutativité

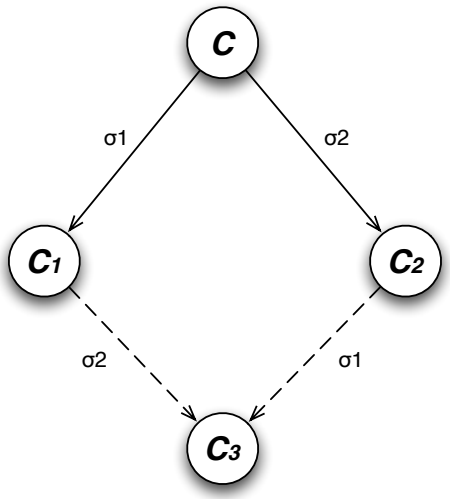
Lemme 1

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

Preuve. Par définition du système et car σ_1 et σ_2 n'ont pas de processus en commun.



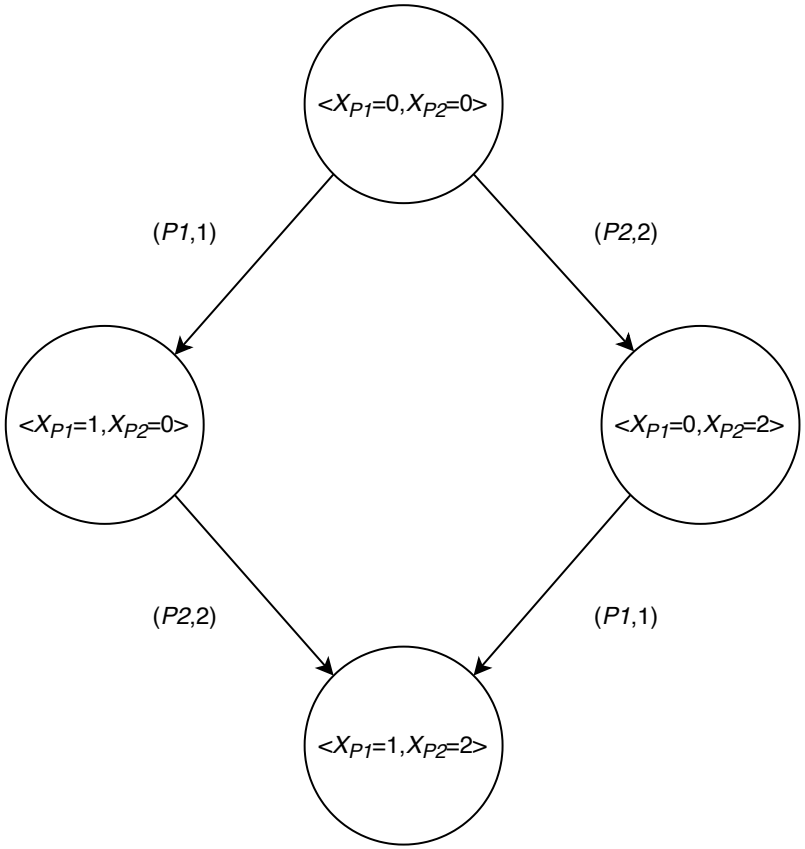
□

Exemple

$\sigma_1 = (P1, 1)$, c.-à-d., $P2$ a envoyé 1 à $P1$ précédemment, et

$\sigma_2 = (P2, 2)$, c.-à-d., $P1$ a envoyé 2 à $P2$ précédemment.

Supposons que chaque processus stocke dans X la dernière valeur reçue.



Idée intuitive de la preuve

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

Idée intuitive de la preuve

Supposons l'existence d'un protocole de consensus $\mathcal{P}$ qui est correct en dépit d'au plus 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'au plus 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 $V \neq \emptyset$.
(toute exécution admissible est décidante, d'après la spécification)

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 $V \neq \emptyset$.
(toute exécution admissible est décidante, d'après la spécification)

Puisque $\mathcal{P}$ est correct, toute configuration qui a une valeur de décision est univalente.

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 $V \neq \emptyset$.
(toute exécution admissible est décidante, d'après la spécification)

Puisque $\mathcal{P}$ est correct, toute configuration qui a une valeur de décision est univalente.

Conséquence : des configurations 0-valentes et 1-valentes sont atteignables depuis n'importe quelle configuration bivalente, s'il y en a ...

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 à partir de laquelle les deux valeurs peuvent être décidées à cause du fait qu'un crash peut arriver.

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 à partir de laquelle les deux valeurs peuvent être décidées à cause du fait qu'un crash peut arriver.

Exemple : avec au plus un crash on peut collecter au plus $n-1$ valeurs proposées (sinon interblocage en cas de crash initial, par exemple)

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 à partir de laquelle les deux valeurs peuvent être décidées à cause du fait qu'un crash peut arriver.

Exemple : avec au plus un crash on peut collecter au plus $n-1$ valeurs proposées (sinon interblocage en cas de crash initial, par exemple)

Supposons que n est impair et qu'on décide lorsqu'on a obtenu $n-1$ propositions.

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 à partir de laquelle les deux valeurs peuvent être décidées à cause du fait qu'un crash peut arriver.

Exemple : avec au plus un crash on peut collecter au plus $n-1$ valeurs proposées (sinon interblocage en cas de crash initial, par exemple)

Supposons que n est impair et qu'on décide lorsqu'on a obtenu $n-1$ propositions.

Supposons qu'on décide **0** lorsqu'on a obtenu un nombre de « 0 » supérieur ou égal au nombre de « 1 ». Sinon on décide **1**.

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 à partir de laquelle les deux valeurs peuvent être décidées à cause du fait qu'un crash peut arriver.

Exemple : avec au plus un crash on peut collecter au plus $n-1$ valeurs proposées (sinon interblocage en cas de crash initial, par exemple)

Supposons que n est impair et qu'on décide lorsqu'on a obtenu $n-1$ propositions.

Supposons qu'on décide **0** lorsqu'on a obtenu un nombre de « 0 » supérieur ou égal au nombre de « 1 ». Sinon on décide **1**.

Une configuration initiale où il y a un « 1 » proposé de plus que de « 0 », est bivalente.

$\rightarrow$ si un processus proposant « 1 » ne fait aucun pas de calcul (c'est possible s'il tombe en panne), alors « 0 » sera décidé.

$\rightarrow$ si un processus proposant « 0 » ne fait aucun pas de calcul (c'est possible s'il tombe en panne), alors « 1 » sera décidé.

Preuve du lemme 2

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

Preuve du lemme 2

Supposons, par contradiction, 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, par contradiction, 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, par contradiction, 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 au moins 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, par contradiction, 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 au moins une chaîne de configurations initiales, chacune adjacente à la suivante : par exemple de $(0,0,0)$ à $(1,1,1)$ on a, entre autres : $(0,0,0) \to (0,0,1) \to (0,1,1) \to (1,1,1)$

On applique cette propriété à un couple de configurations initiales 0-valent/1-valent : il existe nécessairement une configuration initiale 0-valente C_0 adjacente à une configuration initiale 1-valente C_1.

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

Preuve du lemme 2

Supposons, par contradiction, 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 au moins une chaîne de configurations initiales, chacune adjacente à la suivante : par exemple de $(0,0,0)$ à $(1,1,1)$ on a, entre autres : $(0,0,0) \rightarrow (0,0,1) \rightarrow (0,1,1) \rightarrow (1,1,1)$

On applique cette propriété à un couple de configurations initiales 0-valent/1-valent : 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 décidée est 1, alors C_0 n'est pas 0-valente, sinon C_1 n'est pas 1-valente, contradiction. □

Résultat 3

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

Lemme 3

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

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

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

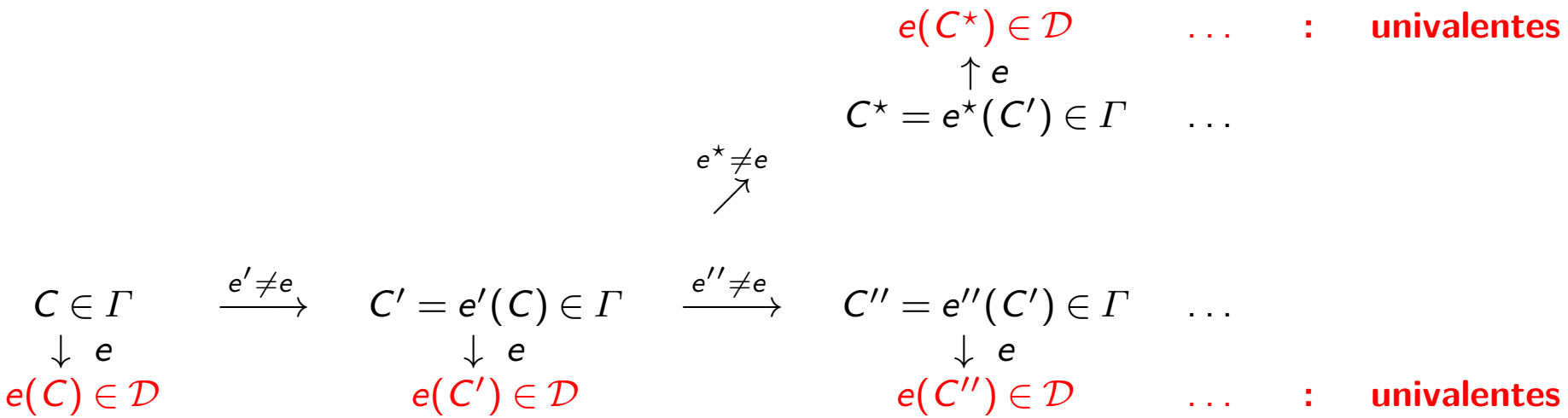
Preuve du lemme 3

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

Preuve du lemme 3

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

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



Preuve du lemme 3

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

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

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

- 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 :

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

Preuve du lemme 3

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

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

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

- 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 :

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

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.

$$\begin{array}{ccccccc}
 & & & & e(C^\star) \in \mathcal{D} & \ldots & : \textbf{univalentes} \\
 & & & & \uparrow e & & \\
 & & & & C^\star = e^\star(C') \in \Gamma & \ldots & \\
 & & & \overset{e^\star \neq e}{\nearrow} & & & \\
C \in \Gamma & \xrightarrow{e' \neq e} & C' = e'(C) \in \Gamma & \xrightarrow{e'' \neq e} & C'' = e''(C') \in \Gamma & \ldots & \\
\downarrow e & & \downarrow e & & \downarrow e & & \\
e(C) \in \mathcal{D} & & e(C') \in \mathcal{D} & & e(C'') \in \mathcal{D} & \ldots & : \textbf{univalentes} \\
\textit{i}\textbf{-valent} & & & & & &
\end{array}$$

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.

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

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**.

$$\begin{array}{ccccccccc} C \in \Gamma & \xrightarrow{e' \neq e} & C_z = C' \in \Gamma & \xrightarrow{e'' \neq e} & C_y = C'' \in \Gamma & \xrightarrow{e''' \neq e} & C_x = C''' \in \Gamma & \ldots & \\ \downarrow e & & \downarrow e & & \downarrow e & & \downarrow e & & \\ e(C) \in \mathcal{D} & & e(C') \in \mathcal{D} & & e(C'') \in \mathcal{D} & & e(C''') \in \mathcal{D} & \ldots & \text{: univalentes} \\ i\text{-valent} & & i\text{-valent} & & j\text{-valent} & & j\text{-valent } (j \neq i) & & \end{array}$$

Preuve du lemme 3

Posons $C_0, C_1 \in \Gamma$ 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 $C_0, C_1 \in \Gamma$ 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 généralité, 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 $C_0, C_1 \in \Gamma$ 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 généralité, posons $C_1 = e'(C_0)$, où $e' = (p', m')$.

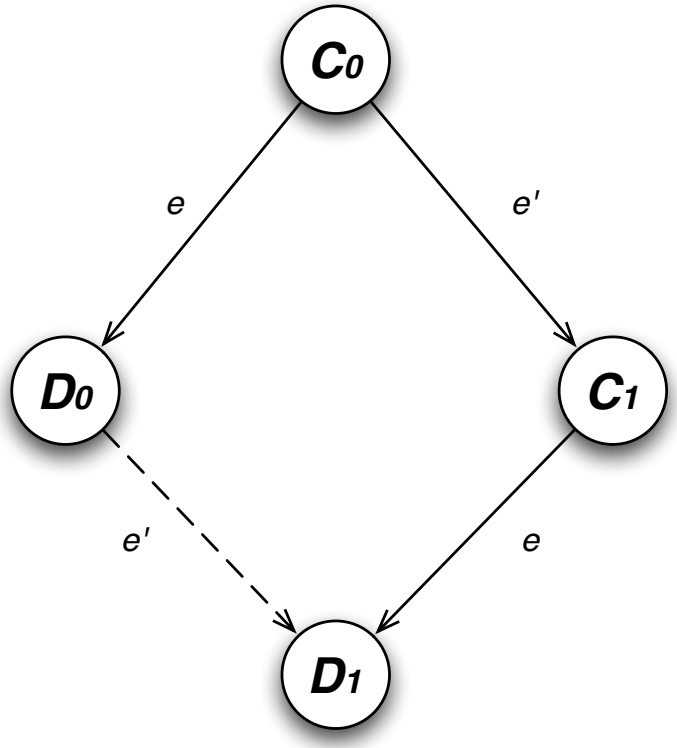
$$\begin{array}{ccc} C_0 & \xrightarrow{e'} & C_1 \\ \downarrow\ e & & \downarrow\ e \\ D_0 & & D_1 \\ \text{0-valent} & & \text{1-valent} \end{array}$$

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 $A = \sigma(C_0)$.

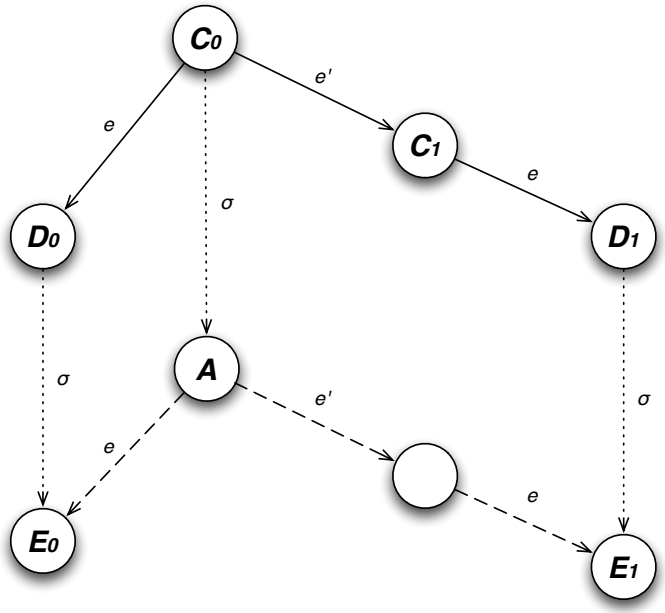
Preuve du lemme 3, Cas 2 : $p' = p$

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 pour 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 pour 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.

Résultat final

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

Résultat final

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**.

Résultat final

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 (faible) qui est correct en dépit d'au plus une faute.

Conclusion

Les sources de l'impossibilité

Ce résultat d'impossibilité fondamentale est principalement dû :

- aux fautes,
- l'asynchronisme,
- au déterminisme,
- et à la décision irrévoquable imposée par la spécification.

Qu'est-ce qu'on fait ?

Les sources de l'impossibilité

Ce résultat d'impossibilité fondamentale est principalement dû :

- aux fautes,
- l'asynchronisme,
- au déterminisme,
- et à la décision irrévoquable imposée par la spécification.

Qu'est-ce qu'on fait ?

Si on supprime ou affaiblit l'une de ses hypothèses, le problème devient soluble.

Les fautes

Sans fautes, le consensus a une solution triviale (déjà vue)

Les fautes

Sans fautes, le consensus a une solution triviale (déjà vue)

Si on fait des hypothèses plus faibles sur la nature des fautes, le consensus peut être solvable.

Par exemple, si on considère que les crashes sont des processus mort-nés (*initially dead*) [3] et qu'il y a une majorité de corrects.

3. Un processus « mort-né » n'exécute jamais le moindre pas de calcul durant l'exécution de l'algorithme.

Les fautes

Sans fautes, le consensus a une solution triviale (déjà vue)

Si on fait des hypothèses plus faibles sur la nature des fautes, le consensus peut être solvable.

Par exemple, si on considère que les crashes sont des processus mort-nés (*initially dead*) [3] et qu'il y a une majorité de corrects.

Plus généralement, la notion de détecteur de pannes a été introduite par **Chandra et Toueg** : un détecteur de panne est un oracle formalisant la connaissance sur les pannes nécessaires (et suffisante) pour résoudre un problème.

3. Un processus « mort-né » n'exécute jamais le moindre pas de calcul durant l'exécution de l'algorithme.

L'asynchronisme

Si le système est synchrone, le consensus est solvable quelque soit le nombre de pannes crash (l'algorithme « FloodSet »)

L'asynchronisme

Si le système est synchrone, le consensus est solvable quelque soit le nombre de pannes crash (l'algorithme « FloodSet »)

Plus généralement, des algorithmes de consensus existent dans des systèmes partiellement synchrones

(seule une partie des liens est synchrone)

Le déterminisme

Il existe des solutions probabilistes au problème du consensus.

Le déterminisme

Il existe des solutions probabilistes au problème du consensus.

Par exemple, l'algorithme de Ben-Hor assure une terminaison avec probabilité 1 (méthode de *Las Vegas*) s'il y a une majorité de corrects

Le déterminisme

Il existe des solutions probabilistes au problème du consensus.

Par exemple, l'algorithme de Ben-Hor assure une terminaison avec probabilité 1 (méthode de *Las Vegas*) s'il y a une majorité de corrects

Il existe aussi des solutions de type *Monte-Carlo* : terminaison déterministe mais probabilité faible d'avoir un conflit

Dans les deux cas (*Las Vegas/Monte Carlo*) la spécification assurée par les solutions est donc plus faible !

La décision

Une dernière approche consiste à affaiblir la spécification au niveau de la décision

Par exemple, le consensus ultime : les décisions ne sont plus irrévocables, mais le système converge vers une configuration à partir de laquelle il y a une unique valeur de décision qui ne change plus jamais

(proche du concept d'autostabilisation)

Conclusion

L'ensemble de ces approches est au programme de

« Systèmes Distribués II »

en Master 2