Détecteurs de défaillances

Stéphane Devismes

Université Joseph Fourier, Grenoble I

17 septembre 2016

Plan

Plan

Conséquence du FLP

L'impossibilité du consensus vient du fait qu'il est impossible de différencier un processus **lent**, d'un processus **en panne**.

Conséquence du FLP

L'impossibilité du consensus vient du fait qu'il est impossible de différencier un processus **lent**, d'un processus **en panne**.

Les hypothèses de synchronie partielle
(quelles sont les informations connues sur la qualité — synchrone, asynchrone, finalement synchrone *etc* — de tel ou tel lien ?)
permettent d'obtenir des informations sur les pannes.

Conséquence du FLP

L'impossibilité du consensus vient du fait qu'il est impossible de différencier un processus **lent**, d'un processus **en panne**.

Les hypothèses de synchronie partielle
(quelles sont les informations connues sur la qualité — synchrone, asynchrone, finalement synchrone *etc* — de tel ou tel lien ?)
permettent d'obtenir des informations sur les pannes.

Ces informations permettent ensuite de résoudre le consensus (ou un autre problème).

Détecteurs de défaillances

L'idée est de séparer

- les connaissances nécessaires sur les pannes pour résoudre un problème,
- de l'implémentation permettant d'obtenir ces connaissances (en particulier, les hypothèses nécessaires sur le système).

Détecteur de défaillances distribué

Chaque processus a accès à un module local de détection de défaillances.

Détecteur de défaillances distribué

Chaque processus a accès à un module local de détection de défaillances.

Pour le détecteur de défaillances distribué $\mathcal{D}$, nous noterons $\mathcal{D}_p$ le module local de détection de défaillances du processus p.

Détecteur de défaillances distribué

Chaque processus a accès à un module local de détection de défaillances.

Pour le détecteur de défaillances distribué $\mathcal{D}$, nous noterons $\mathcal{D}_p$ le module local de détection de défaillances du processus p.

Chaque module surveille un (sous-)ensemble de processus du système, et maintient une liste contenant ceux parmi les processus surveillés qu'il **suspecte** d'être en pannes.

Détecteur de défaillances distribué

Chaque processus a accès à un module local de détection de défaillances.

Pour le détecteur de défaillances distribué $\mathcal{D}$, nous noterons $\mathcal{D}_p$ le module local de détection de défaillances du processus p.

Chaque module surveille un (sous-)ensemble de processus du système, et maintient une liste contenant ceux parmi les processus surveillés qu'il **suspecte** d'être en pannes.

Chaque module peut faire des erreurs :

- en suspectant à tort des processus corrects ou
- en oubliant de suspecter des processus en panne.

Plan

Définition d'une classe

Une classe de détecteurs de défaillances se caractérise par deux propriétés :

- La complétude, qui limite la capacité des détecteurs à oublier des processus crashés.
- L'exactitude, qui limite la capacité des détecteurs à suspecter (à tort) des processus corrects.

Exemple de propriétés de complétude

Complétude forte : Tout processus qui tombe en panne finit par être suspecté en permanence par tous les processus corrects.

Exemple de propriétés de complétude

Complétude forte : Tout processus qui tombe en panne finit par être suspecté en permanence par tous les processus corrects.

Complétude faible : Tout processus qui tombe en panne finit par être suspecté en permanence par certains processus corrects.

Exemple de propriétés d'exactitude

Exactitude forte : Aucun processus n'est suspecté avant qu'il ne tombe ne panne.

Exemple de propriétés d'exactitude

Exactitude forte : Aucun processus n'est suspecté avant qu'il ne tombe ne panne.

Exactitude faible : Certains processus corrects ne sont jamais suspectés.

Exemple de propriétés d'exactitude

Exactitude forte : Aucun processus n'est suspecté avant qu'il ne tombe ne panne.

Exactitude faible : Certains processus corrects ne sont jamais suspectés.

Exactitude finalement forte : Il existe un temps à partir duquel plus aucun processus correct n'est suspecté par un processus correct.

Exemple de propriétés d'exactitude

Exactitude forte : Aucun processus n'est suspecté avant qu'il ne tombe ne panne.

Exactitude faible : Certains processus corrects ne sont jamais suspectés.

Exactitude finalement forte : Il existe un temps à partir duquel plus aucun processus correct n'est suspecté par un processus correct.

Exactitude finalement faible : Il existe un temps à partir duquel certains processus corrects ne sont plus jamais suspectés par les processus corrects.

Différentes classes de détecteurs de défaillances

Complétude	**Exactitude**			
	Forte	Faible	Finalement forte	Finalement faible
Forte	*Parfait* $\mathcal{P}$	*Fort* $\mathcal{S}$	*Finalement parfait* $\diamond\mathcal{P}$	*Finalement fort* $\diamond\mathcal{S}$
Faible	$\mathcal{Q}$	*Faible* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Finalement faible* $\diamond\mathcal{W}$

Plan

Outil de comparaison (1/2)

Le consensus déterministe est prouvé impossible à résoudre dans un système purement asynchrone (avec possibilité d'une panne).

Outil de comparaison (1/2)

Le consensus déterministe est prouvé impossible à résoudre dans un système purement asynchrone (avec possibilité d'une panne).

Cependant, ce problème a des solutions dans plusieurs types de réseau partiellement synchrone.

Outil de comparaison (1/2)

Le consensus déterministe est prouvé impossible à résoudre dans un système purement asynchrone (avec possibilité d'une panne).

Cependant, ce problème a des solutions dans plusieurs types de réseau partiellement synchrone.

Ces différents types de système sont difficiles à comparer.

Outil de comparaison (1/2)

Le consensus déterministe est prouvé impossible à résoudre dans un système purement asynchrone (avec possibilité d'une panne).

Cependant, ce problème a des solutions dans plusieurs types de réseau partiellement synchrone.

Ces différents types de système sont difficiles à comparer.

Par exemple, lequel de ces deux systèmes est le plus faible ?

- Soit $\mathcal{S}_{source}$ un système où au moins un processus (appelé *source*), *a priori* inconnu, est capable d'envoyer de manière fiable des informations à tous processus en temps borné, cette borne étant connu de tous.
- Soit $\mathcal{S}_{fsync}$ un système où :
  - Tous les processus sont finalement synchrone (borne connue) et
  - tous liens sont finalement synchrones (borne connue) et finalement fiables.

    C'est-à-dire le système se comporte de manière fiable et synchrone (borne connue) à partir d'un temps t inconnu des processus *a priori*. Avant ce temps t les messages peuvent, par exemple, être arbitrairement ralentis ou perdus.

Outil de comparaison (2/2)

Ainsi, de manière générale, il est difficile de déterminer quel est le système partiellement asynchrone le plus faible permettant de résoudre le consensus.

Outil de comparaison (2/2)

Ainsi, de manière générale, il est difficile de déterminer quel est le système partiellement asynchrone le plus faible permettant de résoudre le consensus.

L'approche par ≪ détecteur de défaillances ≫ **permet de faire des comparaisons** et ainsi de définir (par exemple) l'hypothèse la plus faible permettant de résoudre le consensus.

Outil de comparaison (2/2)

Ainsi, de manière générale, il est difficile de déterminer quel est le système partiellement asynchrone le plus faible permettant de résoudre le consensus.

L'approche par ≪ détecteur de défaillances ≫ **permet de faire des comparaisons** et ainsi de définir (par exemple) l'hypothèse la plus faible permettant de résoudre le consensus.

Les comparaisons sont basées sur la notion de réduction.

Définition

L'opération de réduction permet de comparer les détecteurs de défaillances entre-eux.

Définition

L'opération de réduction permet de comparer les détecteurs de défaillances entre-eux.

Soient $\mathcal{D}$ et $\mathcal{D}'$ deux détecteurs de défaillances.

Définition

L'opération de réduction permet de comparer les détecteurs de défaillances entre-eux.

Soient $\mathcal{D}$ et $\mathcal{D}'$ deux détecteurs de défaillances.

L'algorithme $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ est un algorithme de réduction de $\mathcal{D}$ vers $\mathcal{D}'$ s'il émule la sortie de $\mathcal{D}'$ en utilisant uniquement $\mathcal{D}$.

Définition

L'opération de réduction permet de comparer les détecteurs de défaillances entre-eux.

Soient $\mathcal{D}$ et $\mathcal{D}'$ deux détecteurs de défaillances.

L'algorithme $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ est un algorithme de réduction de $\mathcal{D}$ vers $\mathcal{D}'$ s'il émule la sortie de $\mathcal{D}'$ en utilisant uniquement $\mathcal{D}$.

Dans ce cas, nous disons que $\mathcal{D}'$ peut être réductible à $\mathcal{D}$ ou $\mathcal{D}'$ est plus faible que $\mathcal{D}$, noté $\mathcal{D}' \preceq \mathcal{D}$.

Définition

L'opération de réduction permet de comparer les détecteurs de défaillances entre-eux.

Soient $\mathcal{D}$ et $\mathcal{D}'$ deux détecteurs de défaillances.

L'algorithme $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ est un algorithme de réduction de $\mathcal{D}$ vers $\mathcal{D}'$ s'il émule la sortie de $\mathcal{D}'$ en utilisant uniquement $\mathcal{D}$.

Dans ce cas, nous disons que $\mathcal{D}'$ peut être réductible à $\mathcal{D}$ ou $\mathcal{D}'$ est plus faible que $\mathcal{D}$, noté $\mathcal{D}' \preceq \mathcal{D}$.

Dans ce cas, tout problème solvable avec $\mathcal{D}'$ sera aussi solvable avec $\mathcal{D}$.

$\prec$

S'il existe un algorithme de réduction de $\mathcal{D}$ vers $\mathcal{D}'$, mais pas le contraire, alors $\mathcal{D}'$ est strictement plus faible que $\mathcal{D}$, noté $\mathcal{D}' \prec \mathcal{D}$.

$\simeq$

S'il existe à la fois un algorithme de réduction de $\mathcal{D}$ vers $\mathcal{D}'$ et de $\mathcal{D}'$ vers $\mathcal{D}$, alors $\mathcal{D}$ et $\mathcal{D}'$ sont dits équivalents, noté $\mathcal{D}' \cong \mathcal{D}$.

Taxinomie

Théorème 1

- $\mathcal{P} \cong \mathcal{Q}$,
- $\mathcal{S} \cong \mathcal{W}$,
- $\diamond\mathcal{P} \cong \diamond\mathcal{Q}$,
- $\diamond\mathcal{S} \cong \diamond\mathcal{W}$,
- $\mathcal{S} \prec \mathcal{P}$,
- $\diamond\mathcal{S} \prec \diamond\mathcal{P}$,
- $\diamond\mathcal{P} \prec \mathcal{P}$,
- $\diamond\mathcal{S} \prec \mathcal{S}$, et
- $\mathcal{S}$ et $\diamond\mathcal{P}$ sont **incomparable**.

Détecteur le plus faible

Le détecteur de défaillances le plus faible pour résoudre un problème P est le détecteur de défaillances $\mathcal{D}$ qui est à la fois **nécessaire et suffisant** pour résoudre P, c'est-à-dire qu'il est plus faible que n'importe quel autre détecteur de défaillances permettant de résoudre P.

Détecteur le plus faible

Le détecteur de défaillances le plus faible pour résoudre un problème P est le détecteur de défaillances $\mathcal{D}$ qui est à la fois **nécessaire et suffisant** pour résoudre P, c'est-à-dire qu'il est plus faible que n'importe quel autre détecteur de défaillances permettant de résoudre P.

Pour cela, il suffit de

- montrer l'existence d'un algorithme utilisant $\mathcal{D}$ résolvant P (suffisant) et

Détecteur le plus faible

Le détecteur de défaillances le plus faible pour résoudre un problème P est le détecteur de défaillances $\mathcal{D}$ qui est à la fois **nécessaire et suffisant** pour résoudre P, c'est-à-dire qu'il est plus faible que n'importe quel autre détecteur de défaillances permettant de résoudre P.

Pour cela, il suffit de

- montrer l'existence d'un algorithme utilisant $\mathcal{D}$ résolvant P (suffisant) et
- montrer qu'il est possible d'émuler $\mathcal{D}$ avec tout algorithme utilisant un détecteur de défaillances $\mathcal{D}'$ et résolvant P (nécessaire).

Plan

Pour résoudre le consensus

Nous allons voir en TD comment résoudre le **consensus** avec le détecteur $\mathcal{S}$ et sans information sur le nombre de pannes.

Pour résoudre le consensus

Nous allons voir en TD comment résoudre le **consensus** avec le détecteur $\mathcal{S}$ et sans information sur le nombre de pannes.

Cependant, notez que si le nombre maximum de pannes est inférieur à la majorité, **le détecteur de défaillances le plus faible permettant de résoudre le consensus** est Ω

(Les processus corrects finissent par être d'accord sur l'identité d'un processus correct).

Pour résoudre le consensus

Nous allons voir en TD comment résoudre le **consensus** avec le détecteur $\mathcal{S}$ et sans information sur le nombre de pannes.

Cependant, notez que si le nombre maximum de pannes est inférieur à la majorité, **le détecteur de défaillances le plus faible permettant de résoudre le consensus** est Ω

(Les processus corrects finissent par être d'accord sur l'identité d'un processus correct).

(L'algorithme est une adaptation de l'algorithme de Ben-Or)

Pour résoudre le consensus

Nous allons voir en TD comment résoudre le **consensus** avec le détecteur $\mathcal{S}$ et sans information sur le nombre de pannes.

Cependant, notez que si le nombre maximum de pannes est inférieur à la majorité, **le détecteur de défaillances le plus faible permettant de résoudre le consensus** est Ω

(Les processus corrects finissent par être d'accord sur l'identité d'un processus correct).

(L'algorithme est une adaptation de l'algorithme de Ben-Or)

Si le nombre maximum de pannes est inconnu, alors **le détecteur de défaillances le plus faible permettant de résoudre le consensus** est $\Sigma \times \Omega$.

(Quorum $+ \Omega$)

(L'algorithme est aussi une adaptation de l'algorithme de Ben-Or)

Plan

Un exemple simple : $\diamond\mathcal{P}$

Tout détecteur de la classe $\diamond\mathcal{P}$ vérifie la complétude forte et l'exactitude finalement forte :

Complétude forte : Tout processus qui tombe en panne finit par être suspecté en permanence par tous les processus corrects.

Exactitude finalement forte : Il existe un temps à partir duquel plus aucun processus correct n'est suspecté par un processus correct.

Système partiellement synchrone : $\mathcal{S}_{\diamond b}$ (1/2)

Le FLP nous impose de faire des hypothèses !

Système partiellement synchrone : $\mathcal{S}_{\diamond b}$ (1/2)

Le FLP nous impose de faire des hypothèses !

Système partiellement synchrone $\mathcal{S}_{\diamond b}$:

Système partiellement synchrone : $\mathcal{S}_{\diamond b}$ (1/2)

Le FLP nous impose de faire des hypothèses !

Système partiellement synchrone $\mathcal{S}_{\diamond b}$:

1. Le réseau de communication est complet.

Système partiellement synchrone : $\mathcal{S}_{\diamond b}$ (1/2)

Le FLP nous impose de faire des hypothèses !

Système partiellement synchrone $\mathcal{S}_{\diamond b}$:

1. Le réseau de communication est complet.
2. Les pannes de processus sont **uniquement** des pannes ≪ crash ≫.

Système partiellement synchrone : $\mathcal{S}_{\diamond b}$ (2/2)

3. Les processus corrects sont finalement synchrones : pour tout processus correct p, il existe un temps t_p (inconnu de tous les processus) à partir duquel p exécute chacune de ses instructions en un temps compris entre α_p et β_p avec $0 < \alpha_p \leq \beta_p$.

Système partiellement synchrone : $\mathcal{S}_{\diamond b}$ (2/2)

3. Les processus corrects sont finalement synchrones : pour tout processus correct p, il existe un temps t_p (inconnu de tous les processus) à partir duquel p exécute chacune de ses instructions en un temps compris entre α_p et β_p avec $0 < \alpha_p \leq \beta_p$.
4. Il existe (au moins) un processus correct $\diamond b$ dont l'ensemble des liens entrants et sortants sont **finalement synchrone et finalement fiable** : c'est-à-dire qu'il existe un temps $t_{\diamond b}$ à partir duquel tout message envoyé par ou vers $\diamond b$ est livré en au plus $\delta_{\diamond b}$ unités de temps.

Système partiellement synchrone : $\mathcal{S}_{\diamond b}$ (2/2)

3. Les processus corrects sont finalement synchrones : pour tout processus correct p, il existe un temps t_p (inconnu de tous les processus) à partir duquel p exécute chacune de ses instructions en un temps compris entre α_p et β_p avec $0 < \alpha_p \leq \beta_p$.
4. Il existe (au moins) un processus correct $\diamond b$ dont l'ensemble des liens entrants et sortants sont **finalement synchrone et finalement fiable** : c'est-à-dire qu'il existe un temps $t_{\diamond b}$ à partir duquel tout message envoyé par ou vers $\diamond b$ est livré en au plus $\delta_{\diamond b}$ unités de temps.

   ($\delta_{\diamond b}$ étant **a priori** inconnu des processus, $\diamond b$ en particulier).

Système partiellement synchrone : $\mathcal{S}_{\diamond b}$ (2/2)

3. Les processus corrects sont finalement synchrones : pour tout processus correct p, il existe un temps t_p (inconnu de tous les processus) à partir duquel p exécute chacune de ses instructions en un temps compris entre α_p et β_p avec $0 < \alpha_p \leq \beta_p$.
4. Il existe (au moins) un processus correct $\diamond b$ dont l'ensemble des liens entrants et sortants sont **finalement synchrone et finalement fiable** : c'est-à-dire qu'il existe un temps $t_{\diamond b}$ à partir duquel tout message envoyé par ou vers $\diamond b$ est livré en au plus $\delta_{\diamond b}$ unités de temps.

   ($\delta_{\diamond b}$ étant **a priori** inconnu des processus, $\diamond b$ en particulier).

   Tout autre lien peut être arbitrairement lent et peut perdre des messages.

   Rappel : tout message non perdu est livré en temps fini.

Système partiellement synchrone : $\mathcal{S}_{\diamond b}$ (2/2)

3. Les processus corrects sont finalement synchrones : pour tout processus correct p, il existe un temps t_p (inconnu de tous les processus) à partir duquel p exécute chacune de ses instructions en un temps compris entre α_p et β_p avec $0 < \alpha_p \leq \beta_p$.
4. Il existe (au moins) un processus correct $\diamond b$ dont l'ensemble des liens entrants et sortants sont **finalement synchrone et finalement fiable** : c'est-à-dire qu'il existe un temps $t_{\diamond b}$ à partir duquel tout message envoyé par ou vers $\diamond b$ est livré en au plus $\delta_{\diamond b}$ unités de temps.

   ($\delta_{\diamond b}$ étant **a priori** inconnu des processus, $\diamond b$ en particulier).

   Tout autre lien peut être arbitrairement lent et peut perdre des messages.

   Rappel : tout message non perdu est livré en temps fini.

   $\diamond b$ est **a priori** inconnu (même de $\diamond b$!).
   $\diamond b$ est appelé une finalement bi-source.

Principes de l'algorithme

Principes de l'algorithme

1. Chaque processus envoie régulièrement des messages ALIVE contenant son identité.

   Tout message est relayé une fois.

   De cette manière, $\diamond b$ agit comme relai et finit par assurer la réception de messages contenant les identités des processus corrects en temps borné.

Principes de l'algorithme

1. Chaque processus envoie régulièrement des messages ALIVE contenant son identité.

   Tout message est relayé une fois.

   De cette manière, $\diamond b$ agit comme relai et finit par assurer la réception de messages contenant les identités des processus corrects en temps borné.
2. Chaque processus maintient un minuteur dédié pour chaque autre processus.

   A l'expiration du minuteur, il suspecte le processus.

   Si plus tard, il reçoit un message contenant l'identité d'un processus suspecté, il arrête de le suspecter et augmente le temps d'attente de son minuteur.

L'algorithme

Algorithme 1 Algorithme $\diamond\mathcal{P}$ pour tout processus p, sortie : *Suspected*

1: $Alive \leftarrow V$; $Suspected \leftarrow \emptyset$
2: **Pour tout** $q \in V \setminus \{p\}$ **faire**
3: $Timer[q] \leftarrow k$; $ElapseTime[q] \leftarrow k$ où k est une constante positive
4: **Fin Pour**
5: **Tant que** *vrai* **faire**
6: **envoie** $\langle ALIVE, p\rangle$ **à tous les processus** $q \neq p$
7: **Pour tout** $q \in V \setminus \{p\}$ **faire**
8: **Si réception** $\langle ALIVE, r\rangle$ **depuis** q **alors**
9: **Si** $ElapseTime[r] \leq 0$ **alors** $Timer[r]++$ **Fin Si**
10: $ElapseTime[r] \leftarrow Timer[r]$; $Alive \leftarrow Alive \cup \{r\}$
11: **Si** $r = q$ **alors**
12: **envoie** $\langle ALIVE, r\rangle$ **à tous les processus** $s \neq p$ **et** $s \neq r$
13: **Fin Si**
14: **Fin Si**
15: **Fin Pour**
16: **Pour tout** $q \in V \setminus \{p\}$ **faire**
17: **Si** $ElapseTime[q] = 0$ **alors** $Alive \leftarrow Alive \setminus \{q\}$ **Sinon** $ElapseTime[q]--$ **Fin Si**
18: **Fin Pour**
19: $Suspected \leftarrow V \setminus Alive$
20: **Fin Tant que**

Correction de l'algorithme (1/6)

Lemme 1

Chaque processus vivant p finit par ne plus recevoir de message $\langle ALIVE, q \rangle$ où q est un processus finit par tomber en panne.

Preuve.

Correction de l'algorithme (1/6)

Lemme 1

Chaque processus vivant p finit par ne plus recevoir de message $\langle ALIVE, q \rangle$ où q est un processus finit par tomber en panne.

Preuve. Une fois qu'il est tombé en panne, q n'envoie plus de message $\langle ALIVE, q \rangle$.

Correction de l'algorithme (1/6)

Lemme 1

Chaque processus vivant p finit par ne plus recevoir de message $\langle ALIVE, q \rangle$ où q est un processus finit par tomber en panne.

Preuve. Une fois qu'il est tombé en panne, q n'envoie plus de message $\langle ALIVE, q \rangle$.

Donc, q envoie un nombre fini de messages.

Correction de l'algorithme (1/6)

Lemme 1

Chaque processus vivant p finit par ne plus recevoir de message $\langle ALIVE, q \rangle$ où q est un processus finit par tomber en panne.

Preuve. Une fois qu'il est tombé en panne, q n'envoie plus de message $\langle ALIVE, q \rangle$.

Donc, q envoie un nombre fini de messages.

Or, chaque message $\langle ALIVE, q \rangle$ envoyé par q est relayé au plus une fois par chacun des autres processus.

Correction de l'algorithme (1/6)

Lemme 1

Chaque processus vivant p finit par ne plus recevoir de message $\langle ALIVE, q \rangle$ où q est un processus finit par tomber en panne.

Preuve. Une fois qu'il est tombé en panne, q n'envoie plus de message $\langle ALIVE, q \rangle$.

Donc, q envoie un nombre fini de messages.

Or, chaque message $\langle ALIVE, q \rangle$ envoyé par q est relayé au plus une fois par chacun des autres processus.

Comme tout message envoyé est livré en temps fini ou perdu, le lemme est vérifié. □

Correction de l'algorithme (2/6)

Lemme 2

Soit p un processus correct. Il existe $t_p, K_p \in \mathbb{N}$ tels que après le temps t_p, p exécute un tour de boucle « tant que » au plus tous les K_p unités de temps.

Preuve.

Correction de l'algorithme (2/6)

Lemme 2

Soit p un processus correct. Il existe $t_p, K_p \in \mathbb{N}$ tels que après le temps t_p, p exécute un tour de boucle « tant que » au plus tous les K_p unités de temps.

Preuve. Par définition, il existe un temps t à partir duquel chaque instruction est exécutée par p en temps borné, car p est finalement synchrone.

Correction de l'algorithme (2/6)

Lemme 2

Soit p un processus correct. Il existe $t_p, K_p \in \mathbb{N}$ tels que après le temps t_p, p exécute un tour de boucle « tant que » au plus tous les K_p unités de temps.

Preuve. Par définition, il existe un temps t à partir duquel chaque instruction est exécutée par p en temps borné, car p est finalement synchrone.

Puisque les lignes 1-4 contiennent un nombre borné d'instructions, p commence à exécuter, dans le pire des cas, la boucle « tant que » en un temps borné t_p après t.

Correction de l'algorithme (2/6)

Lemme 2

Soit p un processus correct. Il existe $t_p, K_p \in \mathbb{N}$ tels que après le temps t_p, p exécute un tour de boucle « tant que » au plus tous les K_p unités de temps.

Preuve. Par définition, il existe un temps t à partir duquel chaque instruction est exécutée par p en temps borné, car p est finalement synchrone.

Puisque les lignes 1-4 contiennent un nombre borné d'instructions, p commence à exécuter, dans le pire des cas, la boucle « tant que » en un temps borné t_p après t.

De la même manière, après t_p, il existe $K_p \in \mathbb{N}$ tel que p exécute un tour de boucle « tant que » au plus tous les K_p unités de temps car la boucle « tant que » contient un nombre borné d'instructions. □

Correction de l'algorithme : Complétude forte (3/6)

Corollaire 1

Tout processus qui tombe en panne finit par être suspecté en permanence par tous les processus corrects.

Preuve. Soient q un processus qui finit par tomber en panne et p un processus correct.

Correction de l'algorithme : Complétude forte (3/6)

Corollaire 1

Tout processus qui tombe en panne finit par être suspecté en permanence par tous les processus corrects.

Preuve. Soient q un processus qui finit par tomber en panne et p un processus correct.

D'après le lemme 2, il existe un temps t_p à partir duquel p exécute chaque tour de boucle « tant que » en temps borné K_p.

Correction de l'algorithme : Complétude forte (3/6)

Corollaire 1

Tout processus qui tombe en panne finit par être suspecté en permanence par tous les processus corrects.

Preuve. Soient q un processus qui finit par tomber en panne et p un processus correct.

D'après le lemme 2, il existe un temps t_p à partir duquel p exécute chaque tour de boucle « tant que » en temps borné K_p.

Donc, à partir de t_p, p finit par vérifier $\mathit{ElapseTime}[q] = 0$ pour toujours, d'après le lemme 1.

Correction de l'algorithme : Complétude forte (3/6)

Corollaire 1

Tout processus qui tombe en panne finit par être suspecté en permanence par tous les processus corrects.

Preuve. Soient q un processus qui finit par tomber en panne et p un processus correct.

D'après le lemme 2, il existe un temps t_p à partir duquel p exécute chaque tour de boucle « tant que » en temps borné K_p.

Donc, à partir de t_p, p finit par vérifier $ElapseTime[q] = 0$ pour toujours, d'après le lemme 1.

Par suite, q est supprimé de la liste *Alive* de p pour toujours et donc finit part être en permanence dans la liste *Suspected* de p. □

Correction de l'algorithme (4/6)

Lemme 3

Soient p et q deux processus corrects tels que $p \neq q$. Il existe $t_p, \Delta_p \in \mathbb{N}$ tel que après t_p, p reçoit $\langle ALIVE, q \rangle$ au plus tous les Δ_p unités de temps.

Preuve.

Soit $\diamond b$ une finalement bi-source.

Correction de l'algorithme (4/6)

Lemme 3

Soient p et q deux processus corrects tels que $p \neq q$. Il existe $t_p, \Delta_p \in \mathbb{N}$ tel que après t_p, p reçoit $\langle ALIVE, q \rangle$ au plus tous les Δ_p unités de temps.

Preuve.

Soit $\diamond b$ une finalement bi-source.

D'après le lemme 2 et par définition d'une finalement bi-source, il existe $t \in \mathbb{N}$ tels que après le temps t, p, q et $\diamond b$ exécutent un tour de boucle $\ll$ tant que $\gg$ au plus tous les K_p, K_q et $K_{\diamond b}$ unités de temps, respectivement ; et tout message envoyé par ou vers $\diamond b$ est livré en au plus $\delta_{\diamond b}$ unités de temps.

Correction de l'algorithme (4/6)

Lemme 3

Soient p et q deux processus corrects tels que $p \neq q$. Il existe $t_p, \Delta_p \in \mathbb{N}$ tel que après t_p, p reçoit $\langle ALIVE, q \rangle$ au plus tous les Δ_p unités de temps.

Preuve.

Soit $\diamond b$ une finalement bi-source.

D'après le lemme 2 et par définition d'une finalement bi-source, il existe $t \in \mathbb{N}$ tels que après le temps t, p, q et $\diamond b$ exécutent un tour de boucle $\ll$ tant que $\gg$ au plus tous les K_p, K_q et $K_{\diamond b}$ unités de temps, respectivement ; et tout message envoyé par ou vers $\diamond b$ est livré en au plus $\delta_{\diamond b}$ unités de temps.

Nous étudions deux cas : $q = \diamond b$ et $q \neq \diamond b$.

Preuve du lemme 3, cas $q = \diamond b$

À partir de t, q envoie un message $\langle ALIVE, q \rangle$ au plus tous les K_q unités de temps.

Preuve du lemme 3, cas $q = \diamond b$

À partir de t, q envoie un message $\langle ALIVE, q \rangle$ au plus tous les K_q unités de temps.

Ces messages sont livrés à p au plus $\delta_{\diamond b}$ unités de temps après leur envoi.

Preuve du lemme 3, cas $q = \diamond b$

À partir de t, q envoie un message $\langle ALIVE, q \rangle$ au plus tous les K_q unités de temps.

Ces messages sont livrés à p au plus $\delta_{\diamond b}$ unités de temps après leur envoi.

Comme p effectue un tour de boucle « tant que » complet au moins tous $2.K_p$ unités de temps, en posant $\Delta_p = K_q + \delta_{\diamond b} + 2.K_p$ le lemme est vérifié dans ce cas.

Preuve du lemme 3, cas $q \neq \diamond b$

Similairement au cas précédent, à partir de t, q envoie un message $\langle ALIVE, q \rangle$ au plus tous les K_q unités de temps.

Preuve du lemme 3, cas $q \neq \diamond b$

Similairement au cas précédent, à partir de t, q envoie un message $\langle ALIVE, q \rangle$ au plus tous les K_q unités de temps.

De plus, $\diamond b$ relaie chacun des messages $\langle ALIVE, q \rangle$ tous les $K_q + \delta_{\diamond b} + 2.K_{\diamond b}$ unités de temps.

Preuve du lemme 3, cas $q \neq \diamond b$

Similairement au cas précédent, à partir de t, q envoie un message $\langle ALIVE, q \rangle$ au plus tous les K_q unités de temps.

De plus, $\diamond b$ relaie chacun des messages $\langle ALIVE, q \rangle$ tous les $K_q + \delta_{\diamond b} + 2.K_{\diamond b}$ unités de temps.

Donc, des messages $\langle ALIVE, q \rangle$ sont livrés à p depuis $\diamond b$ au moins tous les $K_q + 2.\delta_{\diamond b} + 2.K_{\diamond b}$ unités de temps.

Preuve du lemme 3, cas $q \neq \diamond b$

Similairement au cas précédent, à partir de t, q envoie un message $\langle ALIVE, q \rangle$ au plus tous les K_q unités de temps.

De plus, $\diamond b$ relaie chacun des messages $\langle ALIVE, q \rangle$ tous les $K_q + \delta_{\diamond b} + 2.K_{\diamond b}$ unités de temps.

Donc, des messages $\langle ALIVE, q \rangle$ sont livrés à p depuis $\diamond b$ au moins tous les $K_q + 2.\delta_{\diamond b} + 2.K_{\diamond b}$ unités de temps.

Comme p effectue un tour de boucle « tant que » complet au moins tous $2.K_p$ unités de temps, en posant $\Delta_p = K_q + 2.\delta_{\diamond b} + 2.K_{\diamond b} + 2.K_p$ le lemme est vérifié dans ce cas. $\square$

Correction de l'algorithme : Exactitude finalement forte (5/6)

Corollaire 2

Il existe un temps à partir duquel plus aucun processus correct n'est suspecté par un processus correct.

Preuve. Soient p et q deux processus corrects.

Correction de l'algorithme : Exactitude finalement forte (5/6)

Corollaire 2

Il existe un temps à partir duquel plus aucun processus correct n'est suspecté par un processus correct.

Preuve. Soient p et q deux processus corrects.

- Si $p = q$, alors q n'est jamais supprimé de *Alive* de p et donc jamais inséré dans la liste *Suspected* de p. Ainsi, le lemme est trivialement vérifié dans ce cas.

Correction de l'algorithme : Exactitude finalement forte (5/6)

Corollaire 2

Il existe un temps à partir duquel plus aucun processus correct n'est suspecté par un processus correct.

Preuve. Soient p et q deux processus corrects.

- Si $p = q$, alors q n'est jamais supprimé de *Alive* de p et donc jamais inséré dans la liste *Suspected* de p. Ainsi, le lemme est trivialement vérifié dans ce cas.
- Sinon, d'après le lemme 3, q est régulièrement ajouté dans la liste *Alive* de p.

Correction de l'algorithme : Exactitude finalement forte (5/6)

Corollaire 2

Il existe un temps à partir duquel plus aucun processus correct n'est suspecté par un processus correct.

Preuve. Soient p et q deux processus corrects.

- Si $p = q$, alors q n'est jamais supprimé de *Alive* de p et donc jamais inséré dans la liste *Suspected* de p. Ainsi, le lemme est trivialement vérifié dans ce cas.
- Sinon, d'après le lemme 3, q est régulièrement ajouté dans la liste *Alive* de p.

  Supposons, par contradiction, que q est suspecté par p infiniment souvent.

Correction de l'algorithme : Exactitude finalement forte (5/6)

Corollaire 2

Il existe un temps à partir duquel plus aucun processus correct n'est suspecté par un processus correct.

Preuve. Soient p et q deux processus corrects.

- Si $p = q$, alors q n'est jamais supprimé de *Alive* de p et donc jamais inséré dans la liste *Suspected* de p. Ainsi, le lemme est trivialement vérifié dans ce cas.
- Sinon, d'après le lemme 3, q est régulièrement ajouté dans la liste *Alive* de p.

  Supposons, par contradiction, que q est suspecté par p infiniment souvent.

  Donc, q est supprimé infiniment souvent de la liste *Alive* de p.

Correction de l'algorithme : Exactitude finalement forte (5/6)

Corollaire 2

Il existe un temps à partir duquel plus aucun processus correct n'est suspecté par un processus correct.

Preuve. Soient p et q deux processus corrects.

- Si $p = q$, alors q n'est jamais supprimé de *Alive* de p et donc jamais inséré dans la liste *Suspected* de p. Ainsi, le lemme est trivialement vérifié dans ce cas.
- Sinon, d'après le lemme 3, q est régulièrement ajouté dans la liste *Alive* de p.

  Supposons, par contradiction, que q est suspecté par p infiniment souvent.

  Donc, q est supprimé infiniment souvent de la liste *Alive* de p.

  Entre chaque suppression et insertion de q, $Timer[q]$ est incrémenté.

Correction de l'algorithme : Exactitude finalement forte (5/6)

Corollaire 2

Il existe un temps à partir duquel plus aucun processus correct n'est suspecté par un processus correct.

Preuve. Soient p et q deux processus corrects.

- Si $p = q$, alors q n'est jamais supprimé de *Alive* de p et donc jamais inséré dans la liste *Suspected* de p. Ainsi, le lemme est trivialement vérifié dans ce cas.
- Sinon, d'après le lemme 3, q est régulièrement ajouté dans la liste *Alive* de p.

  Supposons, par contradiction, que q est suspecté par p infiniment souvent.

  Donc, q est supprimé infiniment souvent de la liste *Alive* de p.

  Entre chaque suppression et insertion de q, $Timer[q]$ est incrémenté.

  Donc, le temps entre deux réceptions de messages $\langle ALIVE, q \rangle$ consécutives augmente régulièrement d'après le lemme 2, ce qui contredit le lemme 3.

Correction de l'algorithme : Conclusion (6/6)

D'après les corollaires 1 et 2, nous avons le théorème suivant :

Théorème 1

L'algorithme 1 est un détecteur de défaillances de type $\diamond\mathcal{P}$ dans tout système $\mathcal{S}_{\diamond b}$.