Mieux vaut tôt que jamais
(mais tout vient à point à qui sait attendre)

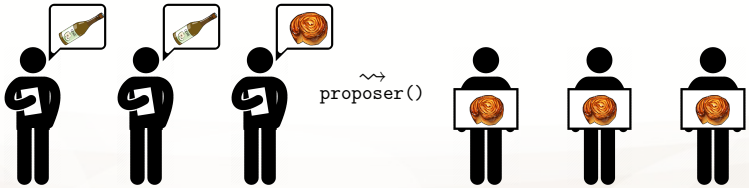
Anaïs Durand, Michel Raynal, Gadi Taubenfeld

31 Mai 2024

UNIVERSITÉ Clermont Auvergne

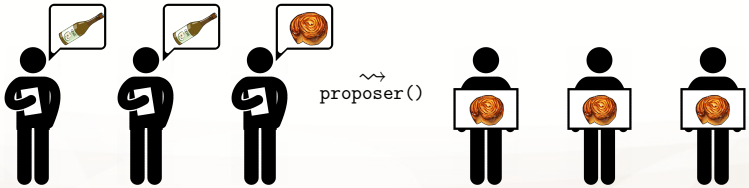
Consensus

Objet partagé, one-shot, avec opération `proposer`



Consensus

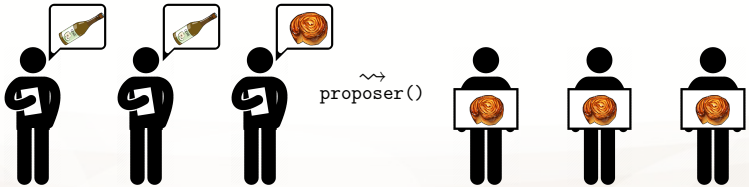
Objet partagé, one-shot, avec opération `proposer`



- Validité :

Consensus

Objet partagé, one-shot, avec opération `proposer`

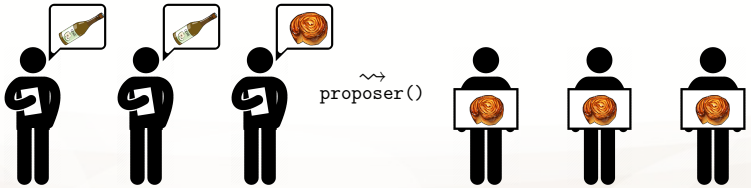


- Validité :

- Accord :

Consensus

Objet partagé, one-shot, avec opération `proposer`



- Validité :

- Accord :

- Terminaison :

UNIVERSITÉ Clermont Auvergne

Exclusion mutuelle (Mutex)

Objet partagé avec deux opérations `acquérir` et `relâcher`

Exclusion mutuelle (Mutex)

Objet partagé avec deux opérations `acquérir` et `relâcher`

- Exclusion mutuelle :

Exclusion mutuelle (Mutex)

Objet partagé avec deux opérations `acquérir` et `relâcher`

- Exclusion mutuelle :

- Absence d'interblocage :

UNIVERSITÉ Clermont Auvergne

Problèmes et résultats fondamentaux

- **Consensus :** Impossible dans les systèmes asynchrones
  - à passage de messages [Fischer, Lynch, Paterson, 1985] et
  - en lecture/écriture [Loui, Abu-Amara, 1987]

  si **un** processus peut **tomber en panne**

Problèmes et résultats fondamentaux

- **Consensus :** Impossible dans les systèmes asynchrones
  - à passage de messages [Fischer, Lynch, Paterson, 1985] et
  - en lecture/écriture [Loui, Abu-Amara, 1987]

  si **un** processus peut **tomber en panne**

- **Exclusion mutuelle :**
  Possible dans les systèmes asynchrones en lecture/écriture [Dijkstra, 1965]
  s'il n'y a **aucune panne** (même sans participation)

Problèmes et résultats fondamentaux

- **Consensus :** Impossible dans les systèmes asynchrones
  - ▷ à passage de messages [Fischer, Lynch, Paterson, 1985] et
  - ▷ en lecture/écriture [Loui, Abu-Amara, 1987]

  si **un** processus peut **tomber en panne**

- **Exclusion mutuelle :**
  Possible dans les systèmes asynchrones en lecture/écriture [Dijkstra, 1965]
  s'il n'y a **aucune panne** (même sans participation)

Consensus ⇔ Exclusion mutuelle

Peut-on **unifier** ces deux résultats fondamentaux ?

Modèle de calcul

- n processus $p_1, p_2, ..., p_n$ identifiés
- Registres atomiques en lecture/écriture
- Participation des processus (non participation $\equiv$ panne initiale)

- **Contention :** # processus ayant accédé à un registre partagé

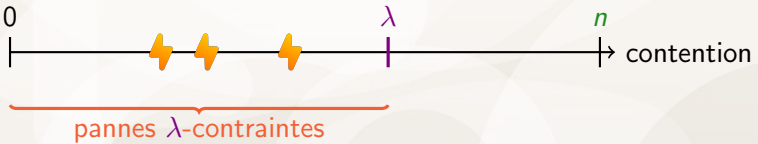
[Taubenfeld, 2018], [Durand, Raynal, Taubenfeld, 2022]

UNIVERSITÉ Clermont Auvergne

Modèle de calcul

- n processus $p_1, p_2, ..., p_n$ identifiés
- Registres atomiques en lecture/écriture
- Participation des processus (non participation $\equiv$ panne initiale)

- **Contention :** # processus ayant accédé à un registre partagé



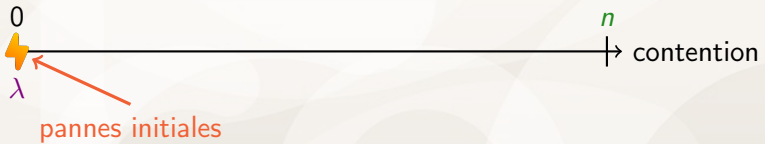
[Taubenfeld, 2018], [Durand, Raynal, Taubenfeld, 2022]

UNIVERSITÉ Clermont Auvergne

Modèle de calcul

- n processus $p_1, p_2, ..., p_n$ identifiés
- Registres atomiques en lecture/écriture
- Participation des processus (non participation $\equiv$ panne initiale)

- **Contention :** # processus ayant accédé à un registre partagé



[Taubenfeld, 2018], [Durand, Raynal, Taubenfeld, 2022]

UNIVERSITÉ Clermont Auvergne

Modèle de calcul

- n processus $p_1, p_2, ..., p_n$ identifiés
- Registres atomiques en lecture/écriture
- Participation des processus (non participation $\equiv$ panne initiale)

- **Contention :** # processus ayant accédé à un registre partagé

[Taubenfeld, 2018], [Durand, Raynal, Taubenfeld, 2022]

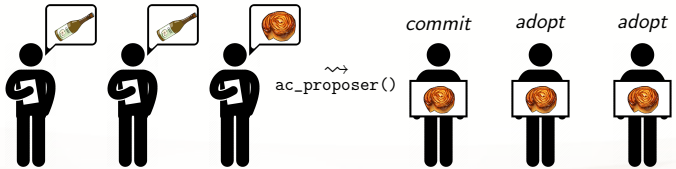
Possibilité du **consensus** asynchrone avec :

- $n \geq k$ processus
- $\leq k$ pannes $(n - k)$-contraintes

Adopt-commit

[Gafni, 1998]

Objet partagé, one-shot, avec opération `ac_proposer`

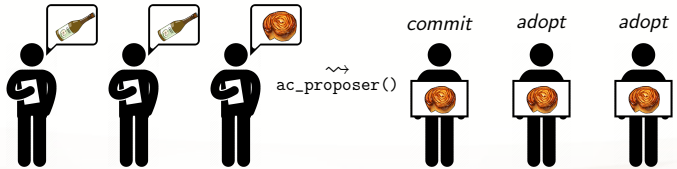


Terminaison + **validité** du consensus

Adopt-commit

[Gafni, 1998]

Objet partagé, one-shot, avec opération `ac_proposer`



Terminaison + **validité** du consensus

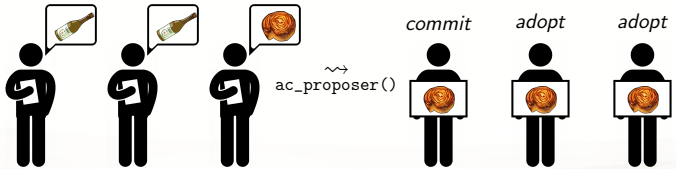
- **Obligation :**

UNIVERSITÉ Clermont Auvergne

Adopt-commit

[Gafni, 1998]

Objet partagé, one-shot, avec opération `ac_proposer`



Terminaison + **validité** du consensus

- **Obligation :**

- **Accord faible :**

UNIVERSITÉ Clermont Auvergne

Adopt-commit

[Gafni, 1998]

Objet partagé, one-shot, avec opération `ac_proposer`

Implémentable dans un système asynchrone
en lecture/écriture
avec pannes

Terminaison +

- Obligation :

- Accord faible :

UNIVERSITÉ Clermont Auvergne

Mutex restreint

Objet partagé avec une opération `acquérir` (et pas d'opération `relâcher` !)

- Exclusion mutuelle :

- Absence d'interblocage :

Mutex restreint

Objet partagé avec une opération `acquérir` (et pas d'opération `relâcher` !)

Implémentable dans un système asynchrone
en lecture/écriture
sans pannes

- Exc

Consensus tolérant k pannes $(n-k)$-contraintes

Objets partagés :

- `INPUT`$[1..n]$: tableau de registres atomiques, initialisés à $\perp$
- `DEC` : registre atomique, initialisé à $\perp$
- `AC` : objet adopt/commit
- `MUT` : objet mutex restreint

Consensus tolérant k pannes $(n-k)$-contraintes

```
    proposer(in_i) {
(1)     INPUT[i] := in_i;

    }
```

Consensus tolérant k pannes $(n-k)$-contraintes

```
    proposer(in_i) {
(1)     INPUT[i] := in_i;
(2)     repeat {
            input_i := lecture async. (non-atomique)
                       de INPUT[1..n];
        } until (input_i contient au plus k entrées à ⊥);




    }
```

Consensus tolérant k pannes $(n-k)$-contraintes

```
    proposer(in_i) {
(1)     INPUT[i] := in_i;
(2)     repeat {
            input_i := lecture async. (non-atomique)
                       de INPUT[1..n];
        } until (input_i contient au plus k entrées à ⊥);
(3)     val_i := max(valeurs dans input_i[1..n]);
    }
```

Consensus tolérant k pannes $(n-k)$-contraintes

```
    proposer(in_i) {
(1)     INPUT[i] := in_i;
(2)     repeat {
            input_i := lecture async. (non-atomique)
                       de INPUT[1..n];
        } until (input_i contient au plus k entrées à ⊥);
(3)     val_i := max(valeurs dans input_i[1..n]);
(4)     ⟨tag_i, res_i⟩ := AC.ac_proposer(val_i);
    }
```

Consensus tolérant k pannes $(n-k)$-contraintes

```
    proposer(in_i) {
(1)     INPUT[i] := in_i;
(2)     repeat {
            input_i := lecture async. (non-atomique)
                       de INPUT[1..n];
        } until (input_i contient au plus k entrées à ⊥);
(3)     val_i := max(valeurs dans input_i[1..n]);
(4)     ⟨tag_i, res_i⟩ := AC.ac_proposer(val_i);
(5)     if (tag_i = commit) { DEC = res_i; return DEC; }

    }
```

Consensus tolérant k pannes $(n-k)$-contraintes

```
    proposer(in_i) {
(1)     INPUT[i] := in_i;
(2)     repeat {
            input_i := lecture async. (non-atomique)
                       de INPUT[1..n];
        } until (input_i contient au plus k entrées à ⊥);
(3)     val_i := max(valeurs dans input_i[1..n]);
(4)     ⟨tag_i, res_i⟩ := AC.ac_proposer(val_i);
(5)     if (tag_i = commit) { DEC = res_i; return DEC; }
(6)     Lancer en parallèle le thread T;
(7)     Attendre jusqu'à ce que DEC ≠ ⊥;
(8)     Tuer T; return DEC;
    }
```

Consensus tolérant k pannes $(n-k)$-contraintes

```
    proposer(in_i) {
(1)     INPUT[i] := in_i;
(2)     repeat {
            input_i := lecture async. (non-atomique)
                       de INPUT[1..n];
        } until (input_i contient au plus k entrées à ⊥);
(3)     val_i := max(valeurs dans input_i[1..n]);
(4)     ⟨tag_i, res_i⟩ := AC.ac_proposer(val_i);
(5)     if (tag_i = commit) { DEC = res_i; return DEC; }
(6)     Lancer en parallèle le thread T;
(7)     Attendre jusqu'à ce que DEC ≠ ⊥;
(8)     Tuer T; return DEC;
    }
```

```
    Thread T {
(9)     MUT.acquérir();
(10)    if (DEC = ⊥) { DEC = res_i; }
    }
```

Correction de l'algorithme

1. **Pas de blocage dans la boucle d'attente :**

   Participation des corrects + $\leq k$ pannes

```
      proposer(in_i) {
(1)       INPUT[i] := in_i;
(2)       repeat {
              input_i := lecture async. (non-atomique) de INPUT[1..n];
          } until (input_i contient au plus k entrées à ⊥);
```

Correction de l'algorithme

(3) $\texttt{val}_i$:= **max**(valeurs dans $\texttt{input}_i[1..n]$);

(4) $\langle \texttt{tag}_i, \texttt{res}_i \rangle$:= AC.ac_proposer($\texttt{val}_i$);

(5) **if** ($\texttt{tag}_i = \textit{commit}$) { DEC = $\texttt{res}_i$; **return** DEC; }

2 Si $\texttt{tag}_i = \textit{commit}$:

Accord faible de AC

$\Rightarrow$ tous les processus ont le **même** $\texttt{res}_i$

$\Rightarrow$ peut décider $\texttt{res}_i$

Correction de l'algorithme

```
(6)     Lancer en parallèle le thread T;
(7)     Attendre jusqu'à ce que DEC ≠ ⊥;
(8)     Tuer T; return DEC;
    }
```

```
    Thread T {
(9)     MUT.acquérir();
(10)    if (DEC = ⊥) { DEC = res_i; }
    }
```

3 Si $\texttt{tag}_i = \textit{adopt}$:

Exclusion mutuelle de `MUT` (+ absence d'opération `relâcher()`)

$\Rightarrow$ un seul processus obtient le mutex

$\Rightarrow$ impose $\texttt{res}_i$ aux autres

Uniquement si les participants à `MUT` sont **corrects**

Correction de l'algorithme

Pourquoi chaque participant p_i à MUT est correct ?

```
    proposer(in_i) {
(1)     INPUT[i] := in_i;
(2)     repeat {
            input_i := lecture async. (non-atomique) de INPUT[1..n];
        } until (input_i contient au plus k entrées à ⊥);
```

Quand p_i sort de la boucle d'attente :

1. soit p_i a lu $\geq n - k + 1$ entrées $\neq \perp$
2. soit p_i a lu exactement $n - k$ entrées $\neq \perp$

Correction de l'algorithme

Pourquoi chaque participant p_i à `MUT` est correct ?

1. p_i a lu $\geq n - k + 1$ entrées $\neq \perp$

   $\Rightarrow \geq n - k + 1$ processus ont écrit dans `INPUT`

Correction de l'algorithme

Pourquoi chaque participant p_i à `MUT` est correct ?

1. p_i a lu $\geq n - k + 1$ entrées $\neq \perp$

   $\Rightarrow \geq n - k + 1$ processus ont écrit dans `INPUT`

   $\Rightarrow$ contention $\geq n - k + 1$

Correction de l'algorithme

Pourquoi chaque participant p_i à `MUT` est correct ?

1. p_i a lu $\geq n - k + 1$ entrées $\neq \perp$

   $\Rightarrow \geq n - k + 1$ processus ont écrit dans `INPUT`

   $\Rightarrow$ contention $\geq n - k + 1$

   $\Rightarrow$ plus de pannes

   $\Rightarrow$ p_i est correct

Correction de l'algorithme

Pourquoi chaque participant p_i à MUT est correct ?

2 p_i a lu exactement $n - k$ entrées $\neq \perp$

p_i participe à MUT $\Rightarrow$ a obtenu *adopt*

(3) $\texttt{val}_i :=$ **max**(valeurs dans $\texttt{input}_i[1..n]$);

(4) $\langle \texttt{tag}_i, \texttt{res}_i \rangle := \texttt{AC.ac\_proposer}(\texttt{val}_i)$;

(5) **if** $(\texttt{tag}_i = \textit{commit})$ { $\texttt{DEC} = \texttt{res}_i$; **return** DEC; }

Correction de l'algorithme

Pourquoi chaque participant p_i à MUT est correct ?

2 p_i a lu exactement $n - k$ entrées $\neq \perp$

p_i participe à MUT $\Rightarrow$ a obtenu *adopt*

(3) $\texttt{val}_i :=$ **max**(valeurs dans $\texttt{input}_i[1..n]$);

(4) $\langle \texttt{tag}_i, \texttt{res}_i \rangle := \texttt{AC.ac\_proposer}(\texttt{val}_i)$;

(5) **if** $(\texttt{tag}_i = \textit{commit})$ { $\texttt{DEC} = \texttt{res}_i$; **return** DEC; }

contraposée de l'**obligation** de AC $\Rightarrow \exists p_j$ tel que $val_i \neq val_j$

Correction de l'algorithme

Pourquoi chaque participant p_i à MUT est correct ?

2 p_i a lu exactement $n - k$ entrées $\neq \perp$

p_i participe à MUT $\Rightarrow$ a obtenu *adopt*

(3) $\texttt{val}_i :=$ **max**(valeurs dans $\texttt{input}_i[1..n]$);
(4) $\langle \texttt{tag}_i, \texttt{res}_i \rangle := \texttt{AC.ac\_proposer}(\texttt{val}_i)$;
(5) **if** $(\texttt{tag}_i = \textit{commit})$ { $\texttt{DEC} = \texttt{res}_i$; **return** DEC; }

contraposée de l'**obligation** de AC $\Rightarrow \exists p_j$ tel que $val_i \neq val_j$

$\Rightarrow p_j$ a lu plus d'entrées $\neq \perp$ dans INPUT

Correction de l'algorithme

Pourquoi chaque participant p_i à MUT est correct ?

2 p_i a lu exactement $n - k$ entrées $\neq \perp$

p_i participe à MUT $\Rightarrow$ a obtenu *adopt*

```
(3)  val_i := max(valeurs dans input_i[1..n]);
(4)  ⟨tag_i, res_i⟩ := AC.ac_proposer(val_i);
(5)  if (tag_i = commit) { DEC = res_i; return DEC; }
```

contraposée de l'**obligation** de AC $\Rightarrow \exists p_j$ tel que $val_i \neq val_j$

$\Rightarrow$ p_j a lu plus d'entrées $\neq \perp$ dans INPUT

$\Rightarrow$ contention $> n - k$ quand p_j invoque ac_proposer
ET avant que p_i termine son invocation à ac_proposer

Correction de l'algorithme

Pourquoi chaque participant p_i à MUT est correct ?

2 p_i a lu exactement $n - k$ entrées $\neq \perp$

p_i participe à MUT $\Rightarrow$ a obtenu *adopt*

```
(3)  val_i := max(valeurs dans input_i[1..n]);
(4)  ⟨tag_i, res_i⟩ := AC.ac_proposer(val_i);
(5)  if (tag_i = commit) { DEC = res_i; return DEC; }
```

contraposée de l'**obligation** de AC $\Rightarrow \exists p_j$ tel que $val_i \neq val_j$

$\Rightarrow$ p_j a lu plus d'entrées $\neq \perp$ dans INPUT

$\Rightarrow$ contention $> n - k$ quand p_j invoque ac_proposer
ET avant que p_i termine son invocation à ac_proposer

$\Rightarrow$ plus de pannes $\Rightarrow$ p_i et p_j sont corrects

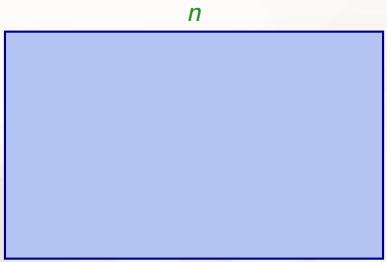
Impossibilité du **consensus** asynchrone avec :

- $n > k$ processus
- $> k$ pannes $(n - k)$-contraintes

Impossibilité du consensus tolérant $k+1$ pannes $(n-k)$-contraintes

Par contradiction : $\mathcal{A}$ algorithme de consensus :

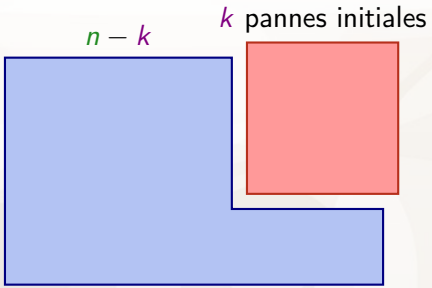
- pour $n > k$ processus
- tolérant $k+1$ pannes $(n-k)$-contraintes



Impossibilité du consensus tolérant $k+1$ pannes $(n-k)$-contraintes

Par contradiction : $\mathcal{A}$ algorithme de consensus :

- pour $n > k$ processus
- tolérant $k+1$ pannes $(n-k)$-contraintes

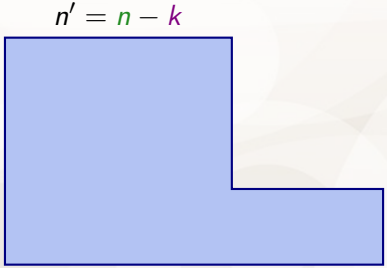


UNIVERSITÉ Clermont Auvergne

Impossibilité du consensus tolérant $k+1$ pannes $(n-k)$-contraintes

Par contradiction : $\mathcal{A}$ algorithme de consensus :

- pour $n > k$ processus
- tolérant $k+1$ pannes $(n-k)$-contraintes

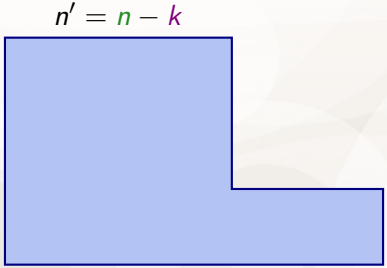


UNIVERSITÉ Clermont Auvergne

Impossibilité du consensus tolérant $k+1$ pannes $(n-k)$-contraintes

Par contradiction : $\mathcal{A}$ algorithme de consensus :

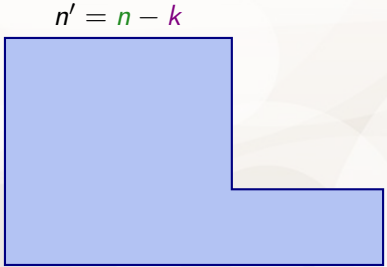
- pour $n' = n - k$ processus
- tolérant 1 panne $(n-k)$-contrainte



Impossibilité du consensus tolérant $k+1$ pannes $(n-k)$-contraintes

Par contradiction : $\mathcal{A}$ algorithme de consensus :

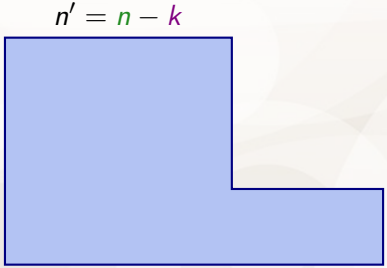
- pour $n' = n - k$ processus
- tolérant 1 panne "classique"



Impossibilité du consensus tolérant $k+1$ pannes $(n-k)$-contraintes

Par contradiction : $\mathcal{A}$ algorithme de consensus :

- pour $n' = n - k$ processus
- tolérant 1 panne "classique" $\Rightarrow$ **contradiction avec** [FLP85], [LA87]



Contribution

Pour tout $0 \leq k \leq n$:

$\exists$ algorithme de consensus tolérant f pannes $(n-k)$-contraintes $\Leftrightarrow f \leq k$

Contribution

Pour tout $0 \leq k \leq n$:

$\exists$ algorithme de consensus tolérant f pannes $(n-k)$-contraintes $\Leftrightarrow f \leq k$

Pour $k = 0$:

$\exists$ algorithme de consensus tolérant f pannes n-contraintes $\Leftrightarrow f \leq 0$

$\equiv$ [Fischer, Lynch, Paterson, 1985]

UNIVERSITÉ Clermont Auvergne

Contribution

Pour tout $0 \leq k \leq n$:

$\exists$ algorithme de consensus tolérant f pannes $(n-k)$-contraintes $\Leftrightarrow f \leq k$

Pour $k = n$:

$\exists$ algorithme de consensus tolérant f pannes 0-contraintes $\Leftrightarrow f \leq n$

$\equiv$ [Dijkstra, 1965]

UNIVERSITÉ Clermont Auvergne

Contribution

Pour tout $0 \leq k \leq n$:

$\exists$ algorithme de consensus tolérant f pannes $(n-k)$-contraintes $\Leftrightarrow f \leq k$

UNIVERSITÉ Clermont Auvergne