Mieux vaut tôt que jamais

Anaïs Durand[1], Michel Raynal[2], et Gadi Taubenfeld[3]

[1]*LIMOS, Université Clermont Auvergne CNRS UMR 6158, Aubière, France*
[2]*IRISA, Université de Rennes, 35042 Rennes, France*
[3]*Reichman University, Herzliya 46150, Israël*

Cet article unifie et généralise des résultats fondamentaux de la tolérance aux pannes dans des systèmes asynchrones. Plus précisément, il montre que pour tout $0 \leq k \leq n$, si les pannes de processus se produisent avant que le seuil de $n - k$ processus ayant commencé leur exécution ne soit dépassé, il existe un algorithme qui résoud le consensus pour n processus en présence de f pannes si et seulement si $f \leq k$. Cela permet d'unifier deux résultats très importants : le cas $k = 0$ est équivalent au très célébre résultat d'impossibilité de Fischer, Lynch et Paterson [FLP85] ; inversement, le cas $k = n$ revient au résultat de Dijkstra [Dij65] sur la possibilité de résoudre l'exclusion mutuelle peu importe le nombre de pannes, tant que ces pannes se produisent avant le début de l'exécution.

Mots-clefs : Consensus, exclusion mutuelle, panne de processus, contention, registres atomiques en lecture/écriture.

1 Introduction

Deux problèmes fondamentaux de l'algorithmique distribuée sont le consensus et l'exclusion mutuelle.

Consensus. Le *consensus* est un problème dans lequel les processus doivent s'accorder sur une valeur en commun. Plus précisément, un objet consensus fournit une opération `proposer()` qui prend en paramètre une valeur, la valeur proposée par le processus. Si le processus est *correct*, autrement dit s'il ne tombe pas en panne, il doit décider une valeur (*Terminaison*). Tous les processus doivent décider une même valeur (*Accord*) et la valeur décidée doit être une des valeurs proposées (*Validité*).

Fischer, Lynch et Paterson ont montré que ce problème est impossible à résoudre dans un système distribué asynchrone sujet à des pannes de processus, même si un seul processus tombe en panne [FLP85].

Exclusion mutuelle. L'*exclusion mutuelle* est un problème de synchronisation formalisé par Dijkstra [Dij65] qui consiste à fournir un objet appelé *verrou* ou *mutex* fournissant deux opérations `acquerir()` et `relacher()`. Il est supposé qu'un processus exécute la séquence de code suivante : `acquerir()`; <section critique>; `relacher()`, où la *section critique* est une séquence de code que l'on souhaite protéger. Deux processus ne doivent pas être en section critique simultanément (*Exclusion mutuelle*). Si aucun processus n'est en section critique, et que des processus souhaitent entrer en section critique (ils sont en train d'invoquer `acquerir()`), alors l'un d'entre eux finit par entrer en section critique (*Absence d'interblocage*).

L'exclusion mutuelle peut être résolue dans un système distribué asynchrone sans pannes en utilisant uniquement des registres atomiques en lecture/écriture et sans que la participation des processus ne soit requise [Dij65]. Sous ces hypothèses, il est possible de résoudre l'exclusion mutuelle si et seulement s'il est possible de résoudre le consensus : Le consensus peut être atteint en décidant la valeur du premier processus à entrer en section critique ; à l'inverse, l'exclusion mutuelle peut être réalisée en utilisant le consensus pour déterminer le prochain processus à entrer en section critique.

Pannes λ-contraintes. En plus du célèbre résultat d'impossibilité, [FLP85] propose un algorithme de consensus qui fonctionne dans un système asynchrone si une majorité de processus sont corrects et si toutes les pannes sont des pannes *initiales* (c'est-à-dire que le processus ne démarre jamais l'exécution de son algorithme). Le "timing" des pannes semblent donc fondamental. Pour capturer cette notion dans un système asynchrone, Taubenfeld propose de remplacer cette notion par le degré de contention, c'est-à-dire

le nombre de processus ayant commencé l'exécution de leur algorithme, et définit les *pannes faibles* [Tau18] renommées par la suite *pannes liées à la contention* [DRT22].

Une *panne λ-contrainte* est une panne qui se produit alors que le degré de contention n'a pas dépassé un seuil λ. Lorsque $\lambda = n$, où n est le nombre de processus, cela revient à une panne "classique" pouvant se produire à tout moment. À l'autre extrême, une panne 0-contrainte est équivalente à une panne initiale ou à un processus (correct) qui ne participe pas à l'algorithme. Il est donc possible de résoudre le consensus et l'exclusion mutuelle dans un système asynchrone sujet à n'importe quel nombre de pannes 0-contraintes.

Ces pannes ont été étudiées dans [DRT22, DRT23].

Contributions et état de l'art. Cet article unifie et généralise les résultats précédemment cités sur le consensus et l'exclusion mutuelle en prouvant le théorème suivant :

Théorème 1 *Dans un système asynchrone à registres partagés, pour tout $0 \leq k \leq n$, il existe un algorithme de consensus pour n processus en présence de f pannes $(n-k)$-contraintes si et seulement si $f \leq k$.*

Deux cas sont particulièrement intéressants :

- Le cas $k = 0$ indique qu'en présence de pannes n-contraintes (autrement dit de pannes "classiques"), aucune panne ne peut être tolérée. Cela implique le résultat d'impossibilité du consensus de [FLP85] (initialement montré dans un système asynchrone à passage de messages mais qui a été étendu aux systèmes asynchrones à registres partagés).
- Le cas $k = n$ indique que le consensus peut être résolu en présence de n pannes 0-contraintes. Cela revient au résultat sur l'exclusion mutuelle lorsque la participation des processus n'est pas requise [Dij65].

Deux résultats ont été montrés précédemment [Tau18, DRT22]. Il est possible de résoudre le consensus :

a). malgré une panne de processus, si elle se produit avant que la contention ne dépasse $\lambda = n - 1$, ou

b). malgré $k - 1$ pannes de processus, avec $k > 1$, si ces pannes se produisent avant que la contention ne dépasse $\lambda = n - k$.

Ces deux résultats étaient prouvés par la proposition de deux algorithmes complexes et basés sur des mécanismes très différents [Tau18, DRT22]. Il restait donc à déterminer (1) si ces résultats étaient optimaux mais également (2) s'il était possible d'unifier ces résultats en un seul algorithme de consensus. La contribution de cet article, comme indiquée dans le Théorème 1, répond négativement à la première question avec un nouveau résultat plus fort et optimal. Il répond également à la seconde question de façon positive avec la proposition d'un algorithme simple présenté dans la section 3 (le "si" du Théorème 1).

Par manque de place, l'optimalité de l'algorithme (le "seulement si" du Théorème 1), autrement dit, pour tout $0 \leq k \leq n$, l'impossibilité de résoudre le consensus pour n processus dans un système asynchrone à registres atomiques en lecture/écriture en présence de $k + 1$ pannes $(n - k)$-contraintes, n'est pas présentée ici. La preuve est disponible dans la version longue de cet article [DRT].

2 Modèle

Le système est composé de n processus asynchrones $p_1, \ldots, p_n$ identifiés. Un processus peut tomber en panne pendant l'exécution (c'est-à-dire s'arrêter prématurément). Il est dit *fautif*. Sinon, c'est un processus *correct*. Nous supposons que tous les processus corrects participent, autrement dit, qu'ils finissent par exécuter leur algorithme. Les processus communiquent via une mémoire partagée composée des objets suivants : des registres atomiques en lecture/écriture (L/E), un objet adopt/commit et un objet mutex.

Adopt-commit. L'objet *adopt-commit* [Gaf98] fournit une opération `ac_proposer()` prenant une valeur en entrée. Il satisfait les propriétés suivantes :

- *Terminaison :* Tout processus correct qui propose une valeur avec l'opération `ac_proposer()` voit son appel se terminer et retourner une paire $\langle tag, v\rangle$, où $tag \in \{\texttt{adopt}, \texttt{commit}\}$.
- *Validité :* La valeur v a été proposée par un processus.
- *Obligation :* Si tous les processus qui appellent `ac_proposer()` proposent la même valeur v, seule la paire $\langle \texttt{commit}, v\rangle$ peut être retournée.
- *Accord faible :* Si un processus décide $\langle \texttt{commit}, v\rangle$ alors tout processus qui décide retourne soit $\langle \texttt{commit}, v\rangle$, soit $\langle \texttt{adopt}, v\rangle$.

Remarquez que si initialement un seul processus exécute `ac_proposer`(v), il retourne la valeur v et toutes les futures invocation à `ac_proposer`() retourneront également v, de même s'il s'agit d'un groupe de processus proposant la même valeur v. L'objet adopt/commit peut être construit dans un système asynchrone à registres L/E sujet à un nombre quelconque de pannes.

Mutex sans interblocage restreint à l'acquisition. L'objet *mutex sans interblocage restreint à l'acquisition* est un objet mutex qui ne fournit qu'une seule opération `acquerir`() (sans opération `relacher`()). Autrement dit, un seul processus peut terminer son invocation de l'opération `acquerir`(). Les autres processus ayant invoqué `acquerir`() ne termineront jamais cette opération.

Ici, cet objet ne sera utilisé que par des processus corrects. Tout algorithme de mutex deadlock-free non tolérant aux pannes est donc utilisable comme par exemple [SP89].

3 Algorithme de consensus tolérant aux pannes λ-contraintes

Dans cette section, nous présentons un algorithme de consensus qui tolère jusqu'à k pannes ayant lieu avant que la contention ne dépasse le seuil $\lambda = n - k$, avec $0 \leq k \leq n$, voir Algorithme 1. Une intuition de la preuve de correction de cet algorithme est donnée ici. Pour une version détaillée, les lecteurs sont invités à consulter [DRT].

Les processus coopèrent en utilisant les objets partagés suivants :

- $INPUT[1..n]$ est un tableau de registres atomiques mono-écrivain multi-lecteurs. Chaque entrée du tableau est initialisée à $\perp$, valeur ne pouvant pas être proposée par un processus. Sans perte de généralité, la valeur $\perp$ est supposée inférieure à toute valeur proposée. $INPUT[i]$ stockera la valeur proposée par p_i.
- DEC est un registre atomique multi-écrivains multi-lecteurs, initialisé à $\perp$, qui stockera la valeur décidée.
- AC est un objet adopt/commit.
- MUT est un mutex sans interblocage restreint à l'acquisition.

Chaque processus p_i dispose en plus de quatres variables locales : val_i, res_i, tag_i et le tableau $input_i[1..n]$.

Description de l'algorithme. Lorsqu'un processus p_i exécute `proposer`(in_i), il commence par mettre in_i, la valeur qu'il propose, dans $INPUT[i]$ (ligne 1). Il attend ensuite jusqu'à ce qu'au moins $(n-k)$ entrées de $INPUT$ soient différentes de la valeur initiale $\perp$ (ligne 2). Par hypothèse, tous les processus corrects participent à l'algorithme et au plus k processus peuvent tomber en panne. Par conséquent, p_i finit par quitter cette boucle d'attente.

Algorithm 1: Consensus tolérant k pannes $(n-k)$-contraintes, $0 \leq k \leq n$

opération `proposer`(in_i) **:**
(1) $INPUT[i] \leftarrow in_i$;
(2) **répéter**
  $input_i \leftarrow$ lecture async. non-atomique de $INPUT[1..n]$;
  jusqu'à ce que $input_i$ contient au plus k $\perp$ **fin** ;
(3) $val_i \leftarrow \max$(valeurs déposées dans $input_i[1..n]$) ;
(4) $\langle tag_i, res_i \rangle \leftarrow AC$.`ac_proposer`($val_i$) ;
(5) **si** tag_i = `commit` **alors** $DEC \leftarrow res_i$; return DEC ; **fin ;**
(6) Lancer en parallèle le thread local T ;
(7) attendre($DEC \neq \perp$) ; tuer(T) ; return DEC ;

thread T **:**
(8) MUT.`acquerir`() ;
(9) **si** $DEC = \perp$ **alors** $DEC \leftarrow res_i$; **fin** ;

Une fois sorti, p_i détermine la plus grande valeur déposée dans $INPUT$ (ligne 3) et la place dans val_i. (Pour rappel, $\perp$ est inférieur à toutes les valeurs proposées par un processus.) La valeur val_i est ensuite proposée comme valeur à décider via l'invocation de `ac_proposer`(val_i) sur l'objet adopt/commit AC. (ligne 4). Cette invocation retourne une paire de valeurs $\langle tag_i, res_i \rangle$.

Si tag_i = `commit`, aucune autre valeur que res_i ne peut être retournée par AC (propriété d'accord faible). Le processus p_i peut donc écrire res_i dans le registre partagé DEC, décider cette valeur et terminer son exécution (ligne 5).

Sinon, les processus lancent un thread local (ligne 6) en parallèle d'une boucle attendant que la valeur de DEC devienne différente de $\perp$ (ligne 7). Si cela se produit, p_i décide la valeur de DEC et termine son exécution. Dans le thread, p_i tente d'obtenir l'accès au mutex (ligne 8). Notez qu'à ce point de l'exécution, tous les processus participant au mutex sont corrects. Cette affirmation sera démontrée dans le prochain

paragraphe. Si p_i finit par entrer en section critique (ce sera le seul, par définition) et que $DEC = \perp$, il écrit res_i dans DEC et décide cette valeur (ligne 9) avant de terminer son exécution. Par hypothèse, l'objet mutex étant sans interblocage, un des processus participant finit par entrer en section critique et à imposer sa valeur aux autres.

Si un processus p_i décide à la ligne 5 et un autre processus p_j décide à la ligne 9, la propriété d'accord faible de l'objet adopt-commit garanti qu'ils décident la même valeur.

Les processus participant au mutex sont corrects. Cette affirmation est le cœur de la preuve de correction de l'algorithme, étant donné qu'il utilise un objet mutex non tolérant aux pannes. Pour montrer qu'un processus p_i participant au mutex est correct, il faut considérer deux cas : lorsque le processus p_i est sorti de la boucle d'attente de la ligne 2, (a) soit il y avait au moins $n - k + 1$ entrées différentes de $\perp$ dans $INPUT$, (b) soit il y en avait exactement $n - k$. Dans le cas (a), il y a au moins $n - k + 1$ processus qui ont commencé leur exécution. Par hypothèse, il ne peut donc plus y avoir de pannes et p_i est correct. Dans le cas (b), il faut d'abord noter que p_i n'a pas obtenu le tag `commit` à la ligne 4, sinon p_i aurait décidé à la ligne 5 sans participer au mutex. Par la contraposée de la propriété d'obligation de l'adopt-commit, au moins un autre processus p_j a proposé à l'adopt-commit une valeur différente de celle proposée par p_i. Les valeurs maximums calculées par p_i et p_j à la ligne 3 sont donc différentes. Par conséquent, p_j a observé un nombre de valeurs différentes de $\perp$ dans $INPUT$ plus grand que celui vu par p_i, autrement dit supérieur à $n-k$. De plus, p_j a calculé sa valeur maximum et l'a proposée à AC avant que p_i ne termine l'invocation de `ac_proposer()`. Il y a donc plus de $n - k$ processus participants à l'exécution lorsque p_i et p_j terminent leur exécution de la ligne 4. Les processus p_i et p_j sont donc tous les deux corrects.

4 Conclusion

Cet article renforce l'idée qu'il est préférable que les pannes se produisent tôt, puisqu'il est possible de tolérer plus de pannes de processus lorsque l'on sait a priori qu'elles vont se produire au début de l'exécution. Ce résultat fait également le lien entre deux résultats fondamentaux de l'algorithmique distribuée sur le consensus [FLP85] et l'exclusion mutuelle [Dij65]. C'est un indice supplémentaire sur la relation très forte entre ces deux problèmes.

Références

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9) :569, 1965.

[DRT] Anaïs Durand, Michel Raynal, and Gadi Taubenfeld. Better sooner rather than latter. In *SIROCCO'24*. to appear, `https://doi.org/10.48550/arXiv.2309.11350`.

[DRT22] Anaïs Durand, Michel Raynal, and Gadi Taubenfeld. Contention-related crash failures : Definitions, agreement algorithms, and impossibility results. *Theor. Comput. Sci.*, 909 :76–86, 2022.

[DRT23] Anaïs Durand, Michel Raynal, and Gadi Taubenfeld. Reaching agreement in the presence of contention-related crash failures. *Theor. Comput. Sci.*, 966-967 :113982, 2023.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2) :374–382, 1985.

[Gaf98] Eli Gafni. Round-by-round fault detectors : Unifying synchrony and asynchrony (extended abstract). In *PODC'98*, pages 143–152, 1998.

[SP89] Eugene Styer and Gary L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *PODC'89*, pages 177–191, 1989.

[Tau18] Gadi Taubenfeld. Weak failures : Definitions, algorithms and impossibility results. In *NETYS'18*, pages 51–66, 2018.