Pannes de processus liées à la contention

Anaïs Durand, Michel Raynal, Gadi Taubenfeld

Pannes de processus liées à la contention †

Anaïs Durand[1], Michel Raynal[2,3] et Gadi Taubenfeld[4]

[1] *LIP6, Sorbonne Université, Paris, France*
[2] *IRISA, Université de Rennes, 35042 Rennes, France*
[3] *Department of Computing, Polytechnic University, Hong Kong*
[4] *The Interdisciplinary Center, Herzliya 46150, Israel*

Cet article est un résumé étendu de [DRT18] dans lequel nous nous intéressons à un nouveau type de pannes de processus défini récemment par l'un des auteurs [Tau18] : les pannes λ-*contraintes*. Cette nouvelle notion est explicitement liée à la contention. En effet, elle ne considère que les exécutions dans lesquelles les pannes de processus se produisent lorsque la contention est plus petite ou égale à un seuil λ donné. Si des pannes se produisent lorsque la contention a dépassé ce seuil λ, aucune propriété de correction (comme par exemple la terminaison) n'est garantie. [Tau18] montre que, lorsque $\lambda = n - 1$, il est possible de résoudre le problème du consensus dans un système asynchrone à registres atomiques de n processus même si un processus tombe en panne, outrepassant ainsi le résultat d'impossibilité FLP. Nous proposons ici des algorithmes pour les problèmes de k-accord et de renommage qui tolèrent à la fois des pannes de processus "classiques" et des pannes λ-contraintes.

Mots-clefs : k-accord, renommage, contention, systèmes asynchrones, tolérance aux pannes

1 Contexte

Le système contient n processus asynchrones, notés $p_1, \ldots, p_n$, qui communiquent en lisant et en écrivant dans des registres atomiques. Le paramètre t représente le nombre maximum de processus qui peuvent tomber en panne pendant l'exécution. Un processus *tombe en panne* lorsqu'il cesse prématurément et de façon définitive son exécution. On dit alors qu'il est *fautif*. Sinon, il est *correct*. La participation, c'est-à-dire l'exécution de l'algorithme local, de tous les processus corrects est requise. (Notez que cette hypothèse est classique, bien que souvent implicite, dans les algorithmes distribués à passage de messages.)

La *contention* est le nombre de processus qui ont commencé leur exécution. Le paramètre λ est un seuil de contention prédéfini. Une exécution peut alors être divisée en deux parties : un préfixe dans lequel la contention est plus petite ou égale à λ et un suffixe dans lequel la contention est supérieure à λ. Nous considérons deux types de pannes de processus : les pannes "classiques" qui peuvent se produire à n'importe quel moment, appelées ici *non-contraintes*, et celles pouvant se produire uniquement quand la contention est inférieure ou égale à λ, dites λ-*contraintes*. Ces dernières ont été présentées pour la première fois dans [Tau18] sous le nom de "weak failures".

Objets concurrents. Afin d'en faciliter la présentation, les algorithmes proposés utilisent deux types d'objets concurrents, la ℓ-exclusion mutuelle sans interblocage et le snapshot. Ces deux objets peuvent être construits dans un système asynchrone à registres sujet à des pannes.

- *ℓ-exclusion mutuelle sans interblocage (ou ℓ-mutex) :* Cet objet fournit deux opérations obtenir() et relâcher(). Il permet à au plus ℓ processus d'exécuter simultanément leur section critique (*exclusion mutuelle*). De plus, il garantit qu'au moins un processus appelant la fonction obtenir() termine son appel si moins de ℓ processus tombent en panne (*terminaison*).
- *Snapshot :* Cet objet fournit deux opérations atomiques écrire() et snapshot(). Il peut être modélisé par un vecteur de registres atomiques $SN[1..n]$ tel que, (a) lorsqu'un processus p_i appelle écrire(v), il écrit dans $SN[i]$ et (b) lorsque p_i appelle snapshot(), il obtient la valeur du vecteur $SN[1..n]$ comme si toutes les entrées avaient été lues simultanément et instantanément.

† Financé en partie par le projet franco-allemand DFG-ANR 40300781 DISCMAT et le projet ANR-16-CE40-0023-03 DESCARTES.

	$m = 0$ $f = k$	$m = \lceil k/2 \rceil$ $f = \lfloor k/2 \rfloor$	$m = k-1$ $f = 1$
pannes λ-contraintes	$\ell - k$	$2\lceil k/2 \rceil + \ell - k$	$\ell + k - 2$
pannes non-contraintes	$k-1$	$\lfloor k/2 \rfloor - 1$	0

TABLE 2: k-accord : compromis entre pannes λ-contraintes et non-contraintes quand $\lambda = n - \ell$.

Intérêt des pannes λ-contraintes. Ce nouveau type de pannes permet de concevoir des algorithmes qui peuvent tolérer plusieurs pannes λ-contraintes en plus de pannes non-contraintes. En particulier, considérons un problème qui peut être résolu en présence de t pannes non-contraintes mais pas en présence de $t+1$ telles pannes. Existe-t-il des solutions à ce problème qui tolèrent $t_1 \leq t$ pannes non-contraintes plus t_2 pannes λ-contraintes, avec $t_1 + t_2 > t$? Plusieurs exemples de tels algorithmes sont proposés ci-après.

Permettre à des algorithmes déjà conçus pour contourner les résultats d'impossibilités de résister en plus à des pannes λ-contraintes pourrait les rendre encore plus robustes. Par exemple, un algorithme *indulgent* ne viole jamais sa propriété de sûreté. Si les hypothèses de synchronisme sur lesquelles il repose sont assurées, il satisfait à terme sa propriété de vivacité. Un algorithme indulgent qui tolère en plus des pannes λ-contraintes pourrait, dans beaucoup de cas, satisfaire sa propriété de vivacité avant même que les hypothèses de synchronisme ne soient remplies.

2 Contributions

Dans ce papier, nous nous intéressons à deux problèmes, le k-accord et le renommage, dans des systèmes asynchrones à registres sujets à des pannes non-contraintes et λ-contraintes.

2.1 k-accord

Le *k-accord* est un objet concurrent "one-shot" avec une seule opération proposer(v). Cette opération permet au processus de proposer une valeur v et d'obtenir comme résultat une valeur *décidée*. Si tous les processus corrects proposent une valeur, chaque processus doit décider une valeur telle que les propriétés suivantes sont satisfaites :

— *Validité :* Toute valeur décidée a été proposée par un processus.
— *Accord :* Au plus k valeurs différentes sont décidées.
— *Terminaison :* Tout processus correct décide une valeur.

Lorsque $k = 1$, le k-accord est équivalent au consensus, problème impossible à résoudre dans les systèmes asynchrones à registres dès lors qu'un processus peut tomber en panne [LAA87]. Il est également impossible de résoudre le k-accord dans les systèmes asynchrones à registres avec $t \geq k$ processus qui tombent en panne [BG93]. Néanmoins, [Tau18] montre qu'il est possible de résoudre le k-accord malgré $t = 2(k-1)$ pannes $(n-k)$-contraintes, contournant ainsi le précédent résultat d'impossibilité.

Résultats. Nous proposons un algorithme générique de k-accord tolérant des pannes λ-contraintes avec $\lambda = n - \ell$ et $\ell \geq k$. Le fonctionnement de cet algorithme est présenté en détail dans la section 3. Il prend deux paramètres (en plus de n, t et ℓ) : les entiers $m \geq 0$ et $f \geq 1$ tels que $m + f = k$. Cet algorithme tolère $(2m + \ell - k)$ pannes $(n-\ell)$-contraintes et $(f-1)$ pannes non-contraintes (voir Table 1). Les paramètres m et f permettent à l'utilisateur d'ajuster le type de pannes qui est dominant dans le contexte d'application de cet algorithme. À un extrême, les valeurs $\langle m, f \rangle = \langle 0, k \rangle$ maximisent le nombre de pannes non-contraintes $(k-1)$. À l'autre extrême, les valeurs $\langle m, f \rangle = \langle k-1, 1 \rangle$ maximisent le nombre de pannes λ-contraintes $(k + \ell - 2)$. La Table 2 résume ce compromis.

Nbr. total de pannes tolérées	$t = 2m + \ell - k + f - 1$
Pannes λ-contraintes	$2m + \ell - k$
Pannes non-contraintes	$f - 1$

TABLE 1: k-accord : pannes tolérées avec $\lambda = n - \ell$, $k = m + f$ et $\ell \geq k$.

2.2 M-renommage

Considérons n processus ayant initialement des noms distincts provenant d'un ensemble de noms quelconque. Le *M-renommage* est un objet concurrent permettant aux processus d'obtenir de nouveaux noms

	$m = 0$ $f = t$	$m = \lceil t/2 \rceil$ $f = \lfloor t/2 \rfloor$	$m = t$ $f = 0$
pannes λ-contraintes	0	$\lceil t/2 \rceil$	t
pannes non-contraintes	t	$\lfloor t/2 \rfloor$	0
taille de l'espace de noms	$n+t$	$n+\lfloor t/2 \rfloor$	n

TABLE 4: $(n+f)$-renommage : compromis entre pannes λ-contraintes et non-contraintes quand $\lambda = n-t-1$.

distincts dans un ensemble de noms $\{1,\ldots,M\}$, où M est une valeur prédéfinie connue par les processus. L'opération renommer(id_i) permet à un processus p_i dont le nom initial est id_i d'obtenir un nouveau nom. Notez que le processus p_i ne connaît ni son index i, ni les noms initiaux des autres processus. Les propriétés suivantes doivent être respectées :

— *Validité :* Un nouveau nom appartient à l'ensemble $\{1,\ldots,M\}$.
— *Accord :* Deux processus ne peuvent obtenir le même nouveau nom.
— *Terminaison :* Si un processus appelle la fonction renommer(id) et ne tombe pas en panne, alors il termine son appel.

Dans un système asynchrone de n processus où au plus t processus peuvent tomber en panne, la taille minimale du nouvel espace de noms est bornée par $(n+t)$.

Résultats. Étant donné un nouvel espace de noms de taille $M = n + f$, nous proposons un algorithme général de M-renommage qui tolère des pannes λ-contraintes avec $\lambda = n-t-1$ et $t \leq n-1$. Cet algorithme prend deux paramètres (en plus de n et t) : les entiers $m \geq 0$ et $f \geq 0$ tels que $t = m + f$. Cet algorithme tolère m pannes $(n-t-1)$-contraintes et f pannes non-contraintes (voir Table 3). Par manque de place, nous ne détaillerons pas ici le fonctionnement de cet algorithme (qui est disponible dans [DRT18]). À nouveau, les paramètres m et f permettent à l'utilisateur d'ajuster le type de pannes qui est dominant dans le contexte d'application de cet algorithme mais aussi la taille du nouvel espace de noms. À un extrême, les valeurs $\langle m, f\rangle = \langle 0, t\rangle$ maximisent le nombre de pannes non-contraintes $(k-1)$ mais maximisent également la taille du nouvel espace de nom. Au contraire, la paire $\langle m, f\rangle = \langle t, 0\rangle$ maximise le nombre de pannes λ-contraintes et minimise la taille du nouvel espace de nom $(M = n)$, contournant ainsi le précédent résultat d'impossibilité. Ceci est résumé dans la Table 4.

Nbr. total de pannes tolérées	$t = m + f$
Pannes λ-contraintes	m
Pannes non-contraintes	f

TABLE 3: $(n + f)$-renommage : pannes tolérées avec $\lambda = n - t - 1$.

3 Algorithme de k-accord

Nous détaillons à présent le fonctionnement de l'algorithme de k-accord tolérant à la fois des pannes non-contraintes et des pannes $(n-\ell)$-contraintes. Comme annoncé dans la Section 1, il est supposé que tous les processus corrects participent. Cet algorithme est caractérisé par le théorème suivant.

Théorème 1. *Pour tout $n \geq 1$, $\ell \geq k$, $0 \leq t \leq n-1$, $m \geq 0$ et $f \geq 1$ tel que $t = 2m + \ell - k + f - 1$ et $k = m + f$, il est possible de résoudre le k-accord pour n processus en présence d'au plus t pannes dont $2m + \ell - k$ pannes λ-contraintes (où $\lambda = n - \ell$) et $f - 1$ pannes non-contraintes.*

La preuve de ce théorème n'est pas détaillée ici, voir [DRT18].

Objets partagés et variables locales. Les processus communiquent via les objets concurrents suivants :

— $PART[1..n]$: objet snapshot, initialisé à $[\nabla,\ldots,\nabla]$, utilisé pour indiquer la participation.
— DEC : registre atomique, initialisé à $\perp$ (une valeur qui ne peut pas être proposée), utilisé pour stocker les valeurs (une à la fois) qui peuvent être décidées.
— $MUTEX[1]$: objet f-mutex "one-shot" sans interblocage.
— $MUTEX[2]$: objet m-mutex "one-shot" sans interblocage.

Dans le cas particulier où $m = 0$ et $f = k$, aucun processus n'essaie d'accéder à $MUTEX[2]$ dans l'algorithme. Par conséquent, il est inutile de définir la notion de 0-mutex. Chaque processus dispose également des variables locales suivantes : $part_i$ est utilisée pour stocker une copie locale de $PART$; $compt_i$ est un compteur local ; et $grp_i \in \{1,2\}$.

```
opération proposer(v_i) :
(1)  PART.écrire(▲);
(2)  repeat part_i ← PART.snapshot();
(3)       compt_i ← |{x tel que part_i[x] = ▲}|;
(4)  until compt_i ≥ n − t end repeat;
(5)  if compt_i ≤ λ then
(6)       grp_i ← 2
(7)  else
(8)       grp_i ← 1
(9)  end if;
(10) lance en parallèle les threads T1 et T2.
% Les deux threads et l'opération terminent
% quand p_i appelle return() (ligne 12 ou 18).
```

```
thread T1 is
(11) loop forever
(12)     if DEC ≠ ⊥ then return(DEC) end if;
(13) end loop.


thread T2 is
(14) if grp_i = 1 ∨ m > 0 then
(15)     MUTEX[grp_i].obtenir();
(16)         if DEC = ⊥ then DEC ← v_i end if;
(17)     MUTEX[grp_i].relâcher();
(18)     return(DEC).
(19) end if;
```

ALGORITHME 1: k-accord en présence d'au plus $2m + \ell - k$ pannes $(n-\ell)$-contraintes et $f-1$ pannes non-contraintes.

Explication de l'algorithme. Le comportement du processus p_i est détaillé dans l'Algorithme 1. Quand p_i appelle proposer(v_i), où v_i est la valeur proposée, p_i commence par indiquer sa participation (ligne 1). Par la suite, l'objet snapshot *PART* est consulté jusqu'à ce qu'au moins $n-t$ processus participent (lignes 2-4). (Notez que cela se produit forcément car il y a au moins $n-t$ processus corrects et tous les processus corrects participent.) Lorsqu'il y a suffisamment de participants, p_i rejoint le groupe 1 ou le groupe 2 selon la valeur de $compt_i$ (lignes 5-9) et lance en parallèle deux threads $T1$ et $T2$ (ligne 10).

Dans le thread $T1$, p_i boucle jusqu'à ce que DEC contienne une valeur proposée. Le processus p_i décide alors cette valeur (ligne 12). L'exécution de return() à la ligne 12 ou 18 termine l'appel à proposer().

Le thread $T2$ est le cœur de l'algorithme. Le processus p_i essaie d'entrer dans la section critique contrôlée par le f-mutex ou m-mutex $MUTEX[grp_i]$ (ligne 15). S'il y parvient et que DEC a toujours sa valeur initiale $\perp$, p_i affecte à DEC la valeur v_i qu'il propose (ligne 16). Enfin, p_i libère la section critique (ligne 17) et décide (ligne 18). Au plus f processus du groupe 1 et m processus du groupe 2 peuvent simultanément exécuter la ligne 16. Il ne peut donc pas y avoir plus de $f + m = k$ valeurs différentes décidées.

4 Conclusion

Dans ce papier, nous nous intéressons à la conception d'algorithmes tolérants à la fois des pannes de processus "classiques" non-contraintes et un nouveau type de pannes liées à la contention, appelées λ-contraintes. Nous proposons un algorithme de k-accord et un algorithme de renommage tolérants ces deux types de pannes. Les algorithmes proposés permettent d'ajuster la proportion de pannes tolérées de chaque type selon le contexte d'exécution. Par exemple, pour le k-accord, il est possible d'échanger une panne "forte" non-contrainte contre deux pannes "faibles" $(n-k)$-contraintes.

Dans [Tau18], un autre type de pannes lié à la contention est défini : les pannes qui ne se produisent que lorsque la contention dépasse un seuil prédéfini. Cette définition peut sembler naturelle, notamment dans des cas où la forte contention peut ralentir des processus au point où certains d'entre eux peuvent abandonner et cesser leur exécution. Cependant, la question suivante reste ouverte : Etant donné un problème qui peut être résolu en présence de t pannes non-contraintes mais pas en présence de $t+1$ telles pannes, existe-t-il un algorithme résolvant ce problème malgré $t' > t$ pannes se produisant après un seuil de contention donné ?

Références

[BG93] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC'93*, pages 91–100, 1993.

[DRT18] Anaïs Durand, Michel Raynal, and Gadi Taubenfeld. Set agreement and renaming in the presence of contention-related crash failures. In *SSS'18*, pages 269–283, 2018.

[LAA87] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4(163-183) :31, 1987.

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