Comment se mettre d'accord quand les autres dorment ?

Anaïs Durand, Michel Raynal, Gadi Taubenfeld

Comment se mettre d'accord quand les autres dorment ?[†]

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*

Arriver à un consensus est fondamental pour de nombreux problèmes de coordination en algorithmique distribuée. Seulement, résoudre ce problème est impossible dans un système asynchrone dès lors qu'un des processus peut être sujet à une panne. Supposons désormais que les pannes aient lieu avant que le degré de contention atteigne un certain seuil λ. Ces pannes sont dites λ-contraintes. Nous montrons qu'en enrichissant le système avec des objets de nombre-consensus $k \geq 1$, le consensus peut être résolu pour $n \geq k$ processus malgré au plus k pannes λ-contraintes. Cet article est un résumé étendu de [DRT22b].

Mots-clefs : Consensus, contention, pannes λ-contraintes, systèmes asynchrones, registres, nombre-consensus.

1 Introduction

Dans le problème du consensus, chaque processus propose une valeur. 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*). Ce problème est impossible à résoudre dans un système distribué asynchrone sujet à des pannes de processus, et ce même si un seul processus tombe en panne [FLP85]. En plus de ce résultat d'impossibilité très connu, Fischer, Lynch et Paterson proposent dans [FLP85] 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). Ce résultat amène à la question suivante : *Est-ce qu'avoir des connaissances préalables sur le timing des pannes peut permettre de résoudre le problème du consensus ?* La notion de "timing" a peu de sens 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* [DRT22a].

Étant donné un seuil de contention λ, une *panne λ-contrainte* est une panne se produisant alors que le degré de contention est plus petit ou égal au seuil λ. Dans [Tau18, DRT22a] est proposé un algorithme de consensus tolérant au plus une panne λ-contrainte où λ vaut $(n-1)$ dans des systèmes asynchrones à registres atomiques en lecture/écriture. Cet algorithme est optimal puisqu'il n'est pas possible de concevoir un algorithme de consensus tolérant plus d'une panne $(n-1)$-contrainte [Tau18, DRT22a]. Ces deux articles proposent également des bornes inférieures et supérieures pour le problème de k-accord en présence de pannes liées à la contention.

Contributions. En utilisant le problème du consensus, nous nous intéressons au lien entre le nombre de pannes liées à la contention qu'il est possible de tolérer, au seuil de contention et à la puissance de calcul des objets de base considérés, cette puissance étant exprimée en fonction de leur *nombre-consensus*. Le nombre-consensus d'un objet O est le nombre maximal de processus pour lequel il est possible de

[†]Ce travail a été supporté en partie par les projets ANR ByBloS (ANR-20-CE25-00002-01), PriCLeSS (ANR-10-LABX-07-81) et SKYDATA (ANR-22-CE25-0008).

résoudre le problème du consensus malgré un nombre arbitraire de pannes (pouvant avoir lieu à n'importe quel moment) en utilisant autant d'objets O que nécessaire et des registres lecture/écriture. Si un tel entier n'existe pas, le nombre-consensus de O est $+\infty$.

Nous proposons trois algorithmes de consensus. Le premier algorithme est construit sur des registres lecture/écriture et un objet adopt/commit, de nombre-consensus 1. Il tolère jusqu'à une panne se produisant avant que le degré de contention ne dépasse $n-1$. Cet algorithme est présenté en Section 3. Le second algorithme généralise le premier avec des objets de nombre-consensus $k \geq 1$. Il tolère jusqu'à k pannes $(n-k)$-contraintes. Cet algorithme est présenté en Section 4. Le troisième algorithme utilise également des objets de nombre-consensus $k \geq 1$. Il suppose que n est divisible par k et tolère jusqu'à $2k-1$ pannes $(n-2k+1)$-contraintes. Par manque de place, cet algorithme n'est pas présenté ici. Il est disponible dans [DRT22b]. Des résultats d'impossibilité sont également proposés dans [DRT22b].

2 Modèle

Le système est composé de n processus asynchrones $p_1, \ldots, p_n$. Un processus peut tomber en panne pendant l'exécution (autrement s'arrêter prématurément). On dit alors qu'il est *fautif*. Sinon, c'est un processus *correct*. Les processus communiquent via une mémoire partagée composée des objets de base suivants : des registres atomiques en lecture/écriture (L/E), des objets atomiques de nombre-consensus $k \geq 1$ (ces objets sont uniquement utilisés en Section 4) et des objets adopt/commit.

Adopt-commit. L'objet *adopt/commit* [Gaf98] peut être construit dans un système asynchrone à registres L/E sujet à un nombre quelconque de pannes. Son nombre-consensus est donc 1. Cet objet fournit une opération ac_propose() prenant une valeur en entrée. L'objet adopt/commit satisfait les propriétés suivantes :

- *Terminaison :* Tout processus correct qui propose une valeur avec l'opération ac_propose() 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_propose() 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 $\langle \texttt{adopt}, v\rangle$.

Valeurs proposées et participation. Sans perte de généralité, nous supposerons que les valeurs proposées sont des entiers positifs et que $\perp$ est supérieur à toute valeur proposée. Nous supposerons également que tous les processus corrects participent à l'algorithme. Si un processus ne participe pas, nous considérerons qu'il a subit une panne initiale.

3 Consensus avec des registres en lecture/écriture

Dans cette section, nous présentons un algorithme qui résout le problème du consensus grâce à des registres L/E et un objet adopt/commit (de nombre-consensus 1) et qui tolère une panne ayant lieu avant que le degré de contention ne dépasse le seuil $\lambda = n-1$, voir Algorithme 1.

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 $\perp$, une valeur ne pouvant pas être proposée par un processus. $INPUT[i]$ servira à stocker la valeur proposée par p_i.
- DEC est un registre atomique multi-écrivains multi-lecteurs, initialisé à $\perp$. Il stockera la valeur décidée.
- $LAST$ contiendra l'index d'un processus.
- AC est un objet adopt/commit.

Chaque processus p_i dispose également de variables locales : val_i, res_i, tag_i et les tableaux $input1_i[1..n]$ et $input2_i[1..n]$. (Notez que par convention, les objets partagés ont un nom en majuscules alors que les variables locales ont un nom en minuscules.)

Description de l'algorithme. Quand un processus p_i exécute l'opération propose(in_i), il commence par déposer la valeur qu'il propose, in_i dans $INPUT[i]$ (Ligne 1) puis attend jusqu'à ce qu'au moins $(n-1)$

Algorithme 1 Consensus tolérant une faute $(n-1)$-contrainte (avec des registres L/E)

```
operation propose(in_i) is                                              code pour p_i
(1) INPUT[i] ← in_i ;
(2) repeat input1_i ← lecture asynchrone non-atomique de INPUT[1..n] ;
            input2_i ← lecture asynchrone non-atomique de INPUT[1..n] ;
    until (input1_i = input2_i ∧ input1_i contient au plus un ⊥) end repeat ;
(3) val_i ← min(valeurs déposées dans input1_i[1..n]) ;
(4) if (∃ j tel que input1_i[j] = ⊥) then LAST ← j end if ;
(5) ⟨tag_i, res_i⟩ ← AC.ac_propose(val_i) ;
(6) if (tag_i = commit ∨ LAST = i) then DEC ← res_i else wait(DEC ≠ ⊥) end if ;
(7) return(DEC) ;
```

entrées de *INPUT* soient différentes de leur valeur initiale $\perp$ (Ligne 2). Puisque nous supposons qu'au plus un processus peut tomber en panne et que tous les processus corrects participent à l'algorithme, p_i finit par sortir de cette boucle d'attente.

Ensuite, p_i calcule la plus petite valeur déposée dans le tableau *INPUT* (Ligne 3) et la stocke dans val_i. Pour rappel, $\perp$ est supérieur à toutes les valeurs proposées par un processus. Si *INPUT* contient encore une entrée égale à $\perp$, par exemple $INPUT[j]$, alors p_i comprends que p_j est en retard ou est l'unique processus fautif (et qu'il est tombé en panne avant d'écrire dans *INPUT*). Le processus p_i enregistre alors cette information en mettant l'index de p_j dans *LAST* (Ligne 4).

Par la suite, p_i propose que sa valeur val_i soit décidée en utilisant l'objet adopt/commit *AC*. L'invocation de AC.ac_propose(val_i) lui retourne une paire $\langle tag_i, res_i\rangle$ (Ligne 5). Trois cas sont alors possibles, dont vont dépendre la valeur que décidera p_i en Ligne 7 :

a). Si tag_i = `commit`, aucune autre valeur que res_i ne peut avoir été retournée par l'objet adopt/commit (propriété d'accord faible). Le processus p_i écrit donc res_i dans le registre partagé *DEC* (Ligne 6) et décide cette valeur (Ligne 7).

b). Si tag_i = `adopt` et p_i est le processus en retard ($LAST = i$), p_i a la garantie d'avoir vu toutes les entrées de *INPUT* avec des valeurs différentes de $\perp$. Il peut donc imposer res_i comme valeur à décider en la mettant dans *DEC* (Ligne 6).

c). Si tag_i = `adopt` $\wedge\ i \neq LAST$, p_i attend jusqu'à ce que $DEC \neq \perp$ pour pouvoir décider.

Correction de l'algorithme. Pourquoi est-on assuré que tous les processus ne soient pas dans le cas (c) et attendent indéfiniment que *DEC* soit différent de $\perp$?

Si aucun processus n'est détecté comme étant en retard, autrement dit l'index d'aucun processus n'est assigné à *LAST* à la ligne 4, lorsqu'un processus p_i exécute la ligne 3, il n'y a aucun $\perp$ dans *INPUT*. Par conséquent, tous les processus calculent la même valeur val_i. Grâce à la propriété d'obligation de l'objet adopt/commit, ils obtiennent forcément le tag `commit` à la ligne 5 et sont dans le cas (a).

À l'inverse, si un processus p_j est détecté comme étant en retard par p_i, il y a plusieurs possibilités. (1). Si p_j est correct, il finira par exécuter la ligne 5 (tous les processus corrects participent) et recevra le tag `commit` (cas (a)) ou le tag `adopt` (cas (b)). (2). Si p_j est fautif, il tombe en panne avant d'écrire in_j dans *INPUT*. En effet, il ne peut tomber en panne que lorsque le degré de contention ne dépasse pas $n-1$. Il ne peut donc pas être le dernier à écrire dans *INPUT*. Par conséquent, $INPUT[j]$ reste à $\perp$ pour toujours. Tous les autres processus p_x calculent donc la même valeur val_x. D'après la propriété d'obligation de l'objet adopt/commit et, comme précédemment, ils se retrouvent dans le cas (a).

La preuve de correction détaillée de l'algorithme 1 est disponible dans [DRT22b].

4 Consensus avec des objets de nombre-consensus $k \geq 1$

L'algorithme 2 est une généralisation de l'algorithme 1 avec des objets de nombre-consensus $k \geq 1$. Il tolère jusqu'à k pannes $(n-k)$-contraintes. L'algorithme 2 a un fonctionnement très proche de l'algorithme 1. Il utilise les mêmes objets partagés *DEC*, *LAST* et *AC*. Il utilise également un tableau *INPUT* de taille $\lceil n/k\rceil$ (au lieu de n), un tableau $kCONS[1..\lceil n/k\rceil]$ où chaque entrée $kCONS[x]$ est un objet de nombre-consensus

k (au lieu d'être un simple registre L/E) et un tableau de Booléens $PARTICIPANT[1..n]$ où chaque entrée est initialisée à `false`.

Description de l'algorithme. Les lignes N1 et N2 sont nouvelles. Elles permettent d'éviter des situations de blocage dans la boucle 2-M même s'il y a k pannes. Le reste de l'algorithme est très similaire à l'algorithme 1, en particulier les lignes ayant le même numéro ont le même rôle. La principale différence réside dans le fait que chaque ensemble d'au plus k processus p_i, p_j, *etc.* tel que $\lceil i/k \rceil = \lceil j/k \rceil$ définit un *cluster* de processus partageant le même objet $kCONS[\lceil i/k \rceil]$ de nombre-consensus k. Tous les processus d'un même cluster simulent l'algorithme 1 en agissant comme s'ils étaient un seul et unique processus $P_{\lceil i/k \rceil}$. En particulier, ils écrivent la même valeur dans $INPUT[\lceil i/k \rceil]$, celle retournée par leur objet $kCONS[\lceil i/k \rceil]$.

Dans le cas où les k pannes ont lieu dans le même cluster, seul ce cluster est fautif ce qui est équivalent à l'unique panne tolérée par l'algorithme 1. Dans les autres cas, tous les clusters sont corrects, puisqu'au moins un processus est correct au sein du cluster et simule le processus équivalent dans l'algorithme 1.

Une preuve détaillée de cet algorithme est disponible dans [DRT22b], ainsi qu'une adaptation permettant de tolérer jusqu'à $(2k-1)$ pannes $(n-2k+1)$-contraintes quand n est divisible par k.

Algorithme 2 Consensus tolérant k fautes $(n-k)$-contraintes (avec des objets de nombre-consensus $k \geq 1$)

```
operation propose(in_i) is                                              code pour p_i
(N1)   PARTICIPANT[i] ← true;
(N2)   repeat participant_i ← lecture asynchrone de PARTICIPANT[1..n];
       until participant_i[1..n] contaient au plus k entrées false end repeat;
(1-M)  INPUT[⌈i/k⌉] ← kCONS[⌈i/k⌉].propose(in_i);
(2-M)  repeat input1_i ← lecture asynchrone non-atomique de INPUT[1..⌈n/k⌉];
              input2_i ← lecture asynchrone non-atomique de INPUT[1..⌈n/k⌉];
       until input1_i = input2_i ∧ input1_i contient au plus un ⊥ end repeat;
(3)    val_i ← min(valeurs déposées dans input1_i);
(4)    if (∃ j tel que input1_i[j] = ⊥) then LAST ← j end if;
(5)    ⟨tag_i, res_i⟩ ← AC.ac_propose(val_i);
(6-M)  if (tag_i = commit ∨ LAST = ⌈i/k⌉) then DEC ← res_i else wait(DEC ≠ ⊥) end if;
(7)    return(DEC);
```

5 Conclusion

Cet article explore la puissance de calcul d'un modèle hybride où la capacité de l'adversaire à engendrer des pannes dépend du degré de contention. Plus précisément, il étudie des systèmes asynchrones enrichis avec des objets de nombre-consensus $k \geq 1$. En particulier, il a été montré que pour $n \geq k$, le consensus peut être résolu dans ce contexte malgré la présence de k pannes se produisant avant que le degré de contention n'ait atteint le seuil $\lambda = n - k$. Ces résultats permettent de contourner le résultat d'impossibilité de Fischer, Lynch et Paterson et d'améliorer notre compréhension du lien entre tolérance aux pannes et synchronisation dans les systèmes distribués asynchrones.

Références

[DRT22a] Anaïs Durand, Michel Raynal, and Gadi Taubenfeld. Contention-related crash failures : Definitions, agreement algorithms, and impossibility results. *Theor. Comput. Sci.*, 909 :76–86, 2022.

[DRT22b] Anaïs Durand, Michel Raynal, and Gadi Taubenfeld. Reaching consensus in the presence of contention-related crash failures. In *SSS 2022*, pages 193–205, 2022.

[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.

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