

Quand le consensus est plus simple que la diffusion fiable

Carole Delporte-Gallet, Stéphane Devismes, Hugues Fauconnier, Franck Petit, Sam Toueg

▶ **To cite this version:**

Carole Delporte-Gallet, Stéphane Devismes, Hugues Fauconnier, Franck Petit, Sam Toueg. Quand le consensus est plus simple que la diffusion fiable. 11èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (AlgoTel 2009), Jun 2009, Carry-Le-Rouet, France. inria-00383349

HAL Id: inria-00383349
https://inria.hal.science/inria-00383349

Submitted on 12 May 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Quand le consensus est plus simple que la diffusion fiable

Carole Delporte-Gallet[1], Stéphane Devismes[2], Hugues Fauconnier[1], Franck Petit[3] et Sam Toueg[4]

[1] *LIAFA, Université Denis Diderot, Paris VII*
[2] *VERIMAG, Université Joseph Fourier, Grenoble I*
[3] *INRIA/LIP, Université de Lyon/ENS Lyon*
[4] *Department of Computer Science, University of Toronto*

Nous nous intéressons à l'impact de la limitation de la mémoire sur des problèmes algorithmiques distribués classiques dans le contexte de systèmes asynchrones sujets aux pannes franches. Les résultats présentés sont issus de [DGDF$^+$08].

Keywords: Tolérance aux pannes, consensus, consensus répété, diffusion fiable

1 Introduction

Contexte. Il n'est pas nécessaire d'insister sur l'importance des problèmes d'*accord* (le consensus, par exemple) et de *diffusion* (la diffusion fiable, par exemple). Ce sont des outils présents dans la plupart des applications distribuées. Ici, nous nous intéressons à ces problèmes dans le cadre de la *tolérance aux pannes*.

En 1985, Fischer, Lynch et Paterson [FLP85] ont démontré un résultat d'impossibilité fondamental pour l'algorithmique distribuée tolérante aux pannes : il n'existe pas d'algorithme déterministe de consensus dans un réseau asynchrone où au plus un processus tombe en panne. Depuis, plusieurs approches ont été proposées pour contourner cette impossibilité : l'approche probabiliste, considérer des systèmes partiellement synchrones, utiliser des *détecteurs de pannes*.

Dans cet article, nous utilisons des détecteurs de pannes [CT96]. Cette approche consiste à supposer l'existence d'une fonction locale à chaque processus qui donne des informations (erronées ou non) sur les défaillances — généralement, une liste de processus suspectés d'être en panne. D'un point de vue théorique, un détecteur de défaillances encapsule la connaissance nécessaire sur les défaillances des processus permettant de résoudre un problème. Il existe plusieurs familles de détecteurs de défaillances, *e.g.*, $\mathcal{P}$, $\mathcal{S}$, Ω, ... Ces familles peuvent être comparées par réduction : un détecteur $\mathcal{D}$ est dit *plus faible* qu'un détecteur $\mathcal{D}'$ s'il existe un algorithme utilisant $\mathcal{D}'$ qui simule $\mathcal{D}$. De plus, s'il n'existe pas d'algorithme utilisant $\mathcal{D}$ qui simule $\mathcal{D}'$, alors $\mathcal{D}$ sera dit *strictement plus faible* que $\mathcal{D}'$ (dans ce cas, nous noterons $\mathcal{D} < \mathcal{D}'$). Puisqu'il existe une hiérarchie parmi les familles de détecteurs de défaillances, une question naturelle est la suivante : « Quel est le détecteur de défaillances *le plus faible*[†] qui permet de résoudre un problème donné ? ». Cette question a été largement traitée dans le contexte des systèmes avec mémoire infinie, par exemple, pour le consensus, la diffusion fiable, la conception de registres atomiques, ... En revanche, cette question n'a pas été abordée, à notre connaissance, pour des systèmes avec processus à mémoire finie.

Contribution. Nous considérons des réseaux asynchrones, bi-directionnels, complets, ayant un nombre fini de processus. Un nombre quelconque de processus peut tomber définitivement en panne. Les liens de communication peuvent perdre des messages mais de manière équitable[‡]. Cette dernière hypothèse, nous contraint à supposer que les liens de communication sont FIFO (voir [LMF88]). Dans la suite, nous

[†] *I.e.*, quel est le détecteur *nécessaire* et *suffisant*.
[‡] *I.e.*, si un processus p envoie une infinité de messages au processus q, alors q en recevra une infinité.

considérons deux types de ces systèmes : les systèmes $\Phi^{\mathcal{F}}$ où chaque processus dispose d'une mémoire finie et les systèmes Φ^{I} où les processus n'ont pas de borne sur la taille de leur mémoire.

Nous nous intéressons à la conception d'algorithmes pour la diffusion fiable, le consensus et le consensus répété dans les systèmes de type $\Phi^{\mathcal{F}}$. Notre contribution est triple : tout d'abord, nous démontrons que le détecteur de défaillances $\mathcal{P}^{-}$ est le plus faible détecteur qui permette de résoudre la diffusion fiable dans les systèmes de type $\Phi^{\mathcal{F}}$. Ensuite, nous prouvons que le consensus peut être résolu dans $\Phi^{\mathcal{F}}$ avec un détecteur de défaillances plus faible : $\mathcal{S}$. En dernier lieu, nous montrons que $\mathcal{P}^{-}$ est le plus faible détecteur de défaillances permettant de résoudre le consensus répété dans $\Phi^{\mathcal{F}}$.

En conséquence, cela signifie que dans les systèmes à mémoire finie, (1) le consensus est plus simple à résoudre que la diffusion fiable et (2) la diffusion fiable est aussi difficile à résoudre que le consensus répété. Ce résultat s'oppose aux résultats dans les systèmes à mémoire infinie où la diffusion fiable est plus simple à résoudre que le consensus et où le consensus est aussi difficile à résoudre que le consensus répété.

Plan. Dans la section suivante, nous définissons les détecteurs de défaillances utilisés dans cet article. Puis, dans les sections 3 à 5, nous considérons la diffusion fiable, le consensus puis le consensus répété dans un système $\Phi^{\mathcal{F}}$. Les perspectives de ce travail sont présentées dans la section 6.

2 Détecteurs de défaillances

Nous utilisons deux types de détecteurs de défaillances connus et nous introduisons une variante du détecteur de défaillances *parfait*. Les détecteurs de défaillances se définissent à l'aide de deux propriétés : la *complétude* et l'*exactitude*. Tous les détecteurs considérés ici vérifient la complétude suivante : chaque processus qui tombe en panne finira par être suspecté pour toujours par les processus corrects (*i.e.*, les processus qui ne tombent jamais en panne). De ce fait, les détecteurs que nous utiliserons diffèrent par leur exactitude. Un détecteur *parfait* (noté $\mathcal{P}$) vérifie l'exactitude *forte* : aucun processus n'est suspecté avant qu'il ne tombe en panne. Un détecteur *presque parfait* (noté $\mathcal{P}^{-}$) vérifie l'exactitude *« presque » forte* : aucun processus correct ne sera jamais suspecté. Enfin, un détecteur *fort* (noté $\mathcal{S}$) vérifie l'exactitude *faible* : il existe un processus correct qui ne sera jamais suspecté. Notons que nous avons : $\mathcal{S} < \mathcal{P}^{-} < \mathcal{P}$. Sans perte de généralité, nous supposons que lorsqu'un processus est suspecté, il l'est pour toujours.

3 Diffusion fiable

Tout d'abord, nous considérons le problème de la *diffusion fiable* dans $\Phi^{\mathcal{F}}$. Tout algorithme de diffusion fiable vérifie les trois propriétés suivantes : **(validité)** si un processus correct diffuse un message m alors tous les processus corrects délivrent m ultimement, **(accord)** si un processus délivre un message de diffusion m alors tous les processus corrects délivrent m ultimement et **(intégrité)** pour chaque message de diffusion m, tout processus délivre m au plus une fois, et seulement si m a été antérieurement diffusé par l'émetteur de m. Nous montrons maintenant que $\mathcal{P}^{-}$ est nécessaire et suffisant pour résoudre la *diffusion fiable* dans $\Phi^{\mathcal{F}}$.

$\mathcal{P}^{-}$ est nécessaire. Pour démontrer que $\mathcal{P}^{-}$ est *nécessaire* pour résoudre la *diffusion fiable* dans $\Phi^{\mathcal{F}}$, nous procédons par réduction : nous supposons l'existence d'un algorithme de diffusion fiable $\mathcal{A}$ pour un système $\Phi^{\mathcal{F}}$ utilisant un détecteur de défaillances $\mathcal{D}$ et nous montrons que nous pouvons simuler $\mathcal{P}^{-}$ en utilisant $\mathcal{A}$.

Notre preuve utilise le résultat intermédiaire suivant : *il existe un entier k tel que pour tout processus p, tout processus correct q et toute exécution E de $\mathcal{A}$ où p diffuse et délivre k messages, il y a au moins un message de q qui est reçu par un processus.*

L'idée derrière ce résultat est la suivante : plusieurs diffusions peuvent être initiées par p en parallèle. Cependant les liens de communication du système étant non fiables, un ou plusieurs processus doivent garder en mémoire tout message diffusé par p jusqu'à avoir la certitude que les processus corrects, q en particulier, l'aient délivré. L'unique manière d'apprendre que q a délivré un message est qu'au moins un processus reçoive un accusé de réception de q. Or, comme les mémoires des processus sont finies, l'ensemble des messages pouvant être stockés dans le système est fini lui aussi.

En utilisant le résultat intermédiaire, nous proposons maintenant un algorithme pour simuler $\mathcal{P}^{-}$. Cet algorithme utilise le module présenté dans la figure 1 : le processus p utilise $\mathcal{A}_{(p,q)}$ pour surveiller le pro-

cessus q. $\mathcal{A}_{(p,q)}$ fonctionne comme suit : p essaie de diffuser k messages mais attend d'avoir délivrer le message courant avant de passer à la diffusion suivante. Le processus q ne fait rien. Les autres processus exécutent $\mathcal{A}$ normalement. D'après le résultat intermédiaire, si q est correct alors p ne peut pas délivrer les k messages et donc ne sort jamais de la boucle « pour ». Si q fini par tomber en panne alors il n'est pas distinguable d'un processus initialement en panne. Or par définition, une diffusion fiable doit terminer en dépit des pannes. Donc, dans ce cas, p finira par sortir de la boucle « pour ». Comme $Output_p$ est initialisé à $\emptyset$ et qu'il est affecté à $\{q\}$ seulement si p sort de la boucle, nous avons la propriété suivante : si q est correct alors $Output_q = \emptyset$ pour toujours, sinon (si q tombe en panne) $Output_q$ vaut ultimement $\{q\}$. L'union des $Output_q$ pour chaque processus q donne à p un détecteur de défaillances de type $\mathcal{P}^-$. D'où, $\mathcal{P}^-$ est nécessaire.

```
1:      /* CODE DU PROCESSUS p */
2:  début
3:      Output_q ← ∅
4:      Pour i = 1 à k faire
5:          diffuser m en utilisant A
6:          attendre que p délivre m
7:      Fin Pour
8:      Output_q ← {q}
9:  fin
```

```
10:     /* CODE DU PROCESSUS q */
11: début
12: fin
13:     /* CODE DES PROCESSUS DANS Π − {p,q} */
14: début
15:     exécuter le code de A
16: fin
17:
18:
```

FIG. 1: $\mathcal{A}_{(p,q)}$

$\mathcal{P}^-$ est suffisant. Pour démontrer que $\mathcal{P}^-$ est *suffisant* pour résoudre la *diffusion fiable* dans $\Phi^{\mathcal{F}}$, nous proposons un algorithme de diffusion fiable utilisant $\mathcal{P}^-$ dans $\Phi^{\mathcal{F}}$. Cet algorithme de diffusion doit tolérer deux types de fautes : les pertes de messages et les pannes de processus. Pour tolérer les pertes de messages, nous utilisons tout simplement le protocole du bit alterné [Ste76]. De ce fait, nous disposons d'une primitive de communication point-à-point fiable en dépit du caractère non-fiable des liens. En utilisant cette primitive et $\mathcal{P}^-$, l'algorithme tolère les pannes de processus en appliquant les principes suivants. Soit p un processus souhaitant diffuser un message m. p envoie un message de diffusion $\langle p\text{-BRD}, m\rangle$ à tous les processus puis attend un accusé de réception p-ACK de tous les processus non-suspectés avant de délivrer m. Une fois que p a délivré m, la diffusion est terminée. Pour assurer la propriété d'accord nous devons assurer que si p tombe en panne durant la diffusion mais qu'un autre processus a délivré le message alors tous les processus corrects ont délivré ou délivreront m. Pour ce faire, chaque processus non-émetteur q délivre m seulement s'il est sûr que tous les processus corrects ont reçu m : lorsque q reçoit le message de diffusion $\langle p\text{-BRD}, m\rangle$, il re-transmet le message à tous les processus qu'il ne suspecte pas sauf p et attend un accusé de réception de tous les processus non-suspectés pour délivrer m. De ce fait, q peut recevoir un message de diffusion de p via un autre processus. Pour synchroniser le système, chaque processus q accuse réception à l'émetteur du message m seulement s'il n'est pas l'initiateur de la diffusion ou si q a déjà reçu un accusé de réception pour m de la part de tous les autres processus non-suspectés.

4 Consensus

Dans le problème du consensus, chaque processus *propose* une valeur et doit *décider* une des valeurs initiale de manière *unanime* et *irrévocable*. Plus formellement, tout algorithme de consensus doit vérifier les trois propriétés suivantes : **(accord)** au plus une valeur est décidée, **(intégrité)** seule une valeur initiale peut être décidée et **(terminaison)** tout processus correct décide ultimement.

Nous montrons ici que $\mathcal{S}$ est suffisant pour réaliser un unique consensus dans $\Phi^{\mathcal{F}}$. Pour ce faire, nous adaptons l'algorithme de [CT96] afin qu'il fonctionne dans $\Phi^{\mathcal{F}}$. Dans cet algorithme, chaque processus dispose d'un tableau V (indexé sur les identités) où il stocke les valeurs proposées par les processus (initialement, chaque case est égale à $\perp$ sauf $V[p]$ qui est égale à la valeur proposée par p). Cet algorithme se déroule en deux phases : une phase d'*union* où le processus apprend un maximum de valeurs proposées puis une phase d'*intersection* où le processus « oublie » certaines valeurs.

La phase d'union se déroule en $n-1$ rondes asynchrones (où n est le nombre de processus). Durant chaque ronde, chaque processus envoie régulièrement son tableau aux autres processus (*n.b.*, les messages sont répétés car les liens ne sont pas fiables). Lorsqu'un processus p reçoit un tableau V', il met à jour

V avec les nouvelles valeurs apprises. La ronde courante termine lorsque p a reçu le tableau de chaque processus non-suspecté. A la fin de la $(n-1)^{\text{ième}}$ ronde, le tableau de chaque processus correct contient au moins toutes les valeurs proposées connues par les processus jamais suspectés (il en existe au moins un, par définition de $\mathcal{S}$).

La phase d'intersection se déroule en une seule ronde asynchrone. Durant cette ronde, chaque processus envoie régulièrement son tableau aux autres processus. La ronde d'un processus termine lorsqu'il a reçu le tableau de chaque processus non-suspecté. A la fin de cette ronde, le processus supprime les valeurs qui ne sont pas présentes dans tous les tableaux qu'il a reçus. De ce fait, le tableau de chaque processus encore vivant contient exactement les mêmes valeurs, les processus décident alors la première valeur du tableau différente de $\perp$ (une telle valeur existe car au moins un processus n'a jamais été suspecté). D'où, ils décident tous la même valeur : l'algorithme réalise un consensus.

5 Consensus répété

Le problème du *consensus répété* consiste à réaliser autant de consensus qu'on le souhaite. En utilisant la méthode de preuve utilisée pour la diffusion fiable, nous avons prouvé que $\mathcal{P}^-$ était nécessaire pour résoudre le consensus répété dans $\Phi^{\mathcal{F}}$. Pour démontrer que $\mathcal{P}^-$ est nécessaire, nous avons construit un algorithme qui utilise une version simplifiée (car il utilise un détecteur de défaillances plus fort, $\mathcal{P}^-$) de l'algorithme de consensus simple vu précédemment. Notre algorithme construit une barrière de synchronisation entre chaque exécution d'un consensus simple. Cette barrière de synchronisation est réalisée en deux rondes asynchrones. Lors de la première ronde, les processus envoient régulièrement un message « décide ». Lors de la seconde ronde, les processus envoient régulièrement un message « nouveau ». Un processus passe à la seconde ronde après avoir reçu un message « décide » ou « nouveau » de chaque processus non-suspecté (*n.b.*, un processeur peut déjà être dans la seconde ronde alors que d'autres sont toujours dans la première). De la même manière, un processus termine la seconde ronde et démarre un nouveau consensus après avoir reçu un message « nouveau » ou un message de l'algorithme de consensus simple de chaque processus non-suspecté. Cette barrière assure qu'aucun processus ne démarre un nouveau consensus alors qu'un autre est encore en train d'exécuter le consensus précédent. D'où, l'algorithme réalise le *consensus répété*.

6 Perspectives

La perspective immédiate de ce travail est de trouver le détecteur de défaillances le plus faible pour résoudre le consensus (simple) dans $\Phi^{\mathcal{F}}$. Nous avons déjà prouvé ici que $\mathcal{S}$ était suffisant. Donc, $\mathcal{S}$ semble être un bon candidat. Dans de futurs travaux, nous souhaitons étendre nos résultats à d'autres types de problèmes comme par exemple la conception de registres distribués. Nous souhaiterions aussi étudier si le fait de considérer des spécifications non-uniformes ou un nombre maximal de pannes permet d'obtenir des solutions avec des détecteurs de défaillances plus faibles.

Références

[CT96] T. Deepak Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2) :225–267, 1996.

[DGDF+08] C. Delporte-Gallet, S. Devismes, H. Fauconnier, F. Petit, and S. Toueg. With finite memory consensus is easier than reliable broadcast. In *OPODIS*, pages 41–57, 2008.

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

[LMF88] N. A. Lynch, Y. Mansour, and A. Fekete. Data link layer : Two impossibility results. In *Symposium on Principles of Distributed Computing*, pages 149–170, 1988.

[Ste76] V. Stenning. A data transfer protocol. *Computer Networks*, 1 :99–110, 1976.