Le consensus vectoriel : une nouvelle spécification du problème du consensus dans un modèle Byzantin

Assia Doudou André Schiper
E.P.F.L., Département d'Informatique
1015 Lausanne (Suisse)
e-mail: {doudou,schiper}@di.epfl.ch

Résumé

Le concept de détecteur de défaillances a été défini par Chandra et Toueg afin de résoudre le problème du consensus dans un environnement asynchrone avec pannes franches. Dans ce travail, nous présentons une extension de ce concept afin de traiter des défaillances plus générales, dites défaillances byzantines. Nous discutons également les limitations de la spécification du consensus en présence de défaillances byzantines et proposons pour lever ces limitations la spécification d'un nouveau problème, appelé Consensus vectoriel. L'algorithme proposé pour résoudre le consensus vectoriel est un algorithme avec signature de messages. Il se base sur le paradigme du coordinateur tournant et sur celui du détecteur de défaillances. L'algorithme nécessite $\lceil (2n+1)/3 \rceil$ processus corrects et des canaux FIFO fiables.

Mots-clés : consensus, défaillances byzantines, détecteur de défaillances, système asynchrone

---

1. Introduction

Le problème du consensus est un problème fondamental dans les systèmes distribués, car il représente une brique de base permettant de construire d'autres protocoles d'agrément tel que l'ordre total. L'objectif du consensus est de permettre à un ensemble de processus, chacun possédant sa propre valeur initiale, de décider de manière irrévocable sur une de ces valeurs initiales. De manière plus formelle, le consensus est spécifié par les propriétés suivantes : (1) **agrément :** deux processus ne peuvent décider différemment, (2) **terminaison :** tout processus doit finalement décider et (3) **validité :** la valeur décidée est une des valeurs proposées. Cependant, un résultat établi par Fischer, Lynch et Paterson prouve que le problème du consensus n'a pas de solution dans un système asynchrone, même en présence d'une seule défaillance [3]. Cette impossibilité a été contournée par Chandra et Toueg en définissant le paradigme du détecteur de défaillances [1].

Ce paradigme a été pendant longtemps considéré uniquement dans des modèles avec défaillances franches. Malkhi et Reiter ont été les premiers à étendre cette notion au modèle byzantin [6]. Toutefois, le travail présenté par Malkhi et Reiter se situe dans un modèle où les processus communiquent grâce à une primitive de diffusion fiable qui suppose de plus un ordre causal. La première contribution de cet article est d'étendre ce paradigme à un modèle byzantin plus général, où les processus ne sont pas obligés d'utiliser un diffusion fiable; en outre aucune hypothèse sur l'ordre causal n'est nécessaire.

Par ailleurs, Malkhi et Reiter considèrent le problème du *consensus strict* qui est une variante du consensus traditionnel. Dans le consensus strict, on substitue la propriété de validité par la propriété de *validité stricte*. Cette dernière assure que la valeur décidée est celle d'un processus correct si et seulement si tous les processus corrects ont la même valeur initiale. L'intérêt d'une telle spécification est de garantir que la décision est une valeur initiale d'un processus correct, moyennant une condition sur les valeurs initiales des processus corrects. Dans le modèle byzantin, on considère la validité stricte plutôt que la validité traditionnelle, car cette dernière a l'inconvénient de ne pas exclure que la décision soit la valeur initiale d'un processus byzantin. Toutefois, le consensus strict est trop restrictif pour servir à la résolution d'autres protocoles d'agrément tel que l'ordre total.

Ceci nous conduit à la deuxième contribution de cette article, à savoir la définition du *Consensus Vectoriel*. L'objectif de cette nouvelle spécification est de définir un problème mieux adapté à la résolution d'autres problèmes d'agrément.

2. Modélisation du système

Nous considérons un système distribué asynchrone composé d'un ensemble fini de n processus, complètement connectés grâce à un ensemble de canaux. Les défaillances considérées sont des défaillances arbitraires (défaillances byzantines). Ces défaillances correspondent à un comportement imprévisible de certains processus qui ne respectent pas la spécification de leur protocole (processus byzantins). Le nombre de processus byzantin f est inférieur à $n/3$. Chaque processus possède une clef privée secrète et une clef publique. La clef privée sert à signer les messages avant leurs envois et à les protéger de toute modification durant cet envoi. La clef publique sert uniquement à authentifier l'origine du message [8]. Nous supposons également que les canaux sont FIFO et fiables.

3. Le détecteur de défaillances $\diamond M$

Un détecteur de défaillances est un oracle local à chaque site du système, dont le rôle est de fournir des informations sur les défaillances des autres processus. Dans un système asynchrone, ces informations peuvent être erronées. En effet, la non connaissance a priori des délais de communications peut conduire à suspecter à tort un processus correct mais lent. Chandra et Toueg ont défini deux propriétés qui caractérisent le détecteur de défaillances noté $\diamond S$: (1) **complétude forte** : tous les processus incorrects finissent par être suspectés, et (2) **justesse faible** : il existe un temps à partir duquel un processus correct n'est plus jamais suspecté. Cette définition est insuffisante dans un modèle byzantin. À titre d'exemple, un processus Byzantin peut décider volontairement d'ignorer une partie ou la totalité des processus en n'envoyant plus de messages alors qu'il n'est pas en panne, ou encore induire les processus corrects en erreur en leur envoyant des messages erronés. Face à la diversité des défaillances possibles, l'approche que nous avons choisie consiste à capturer au niveau du détecteur uniquement les défaillances qui empêchent la progression du protocole de consensus. La détection d'erreurs plus complexes, liées à la sémantique des messages, est déléguée au consensus lui même. Ceci nous conduit à définir la notion de processus **muet**.

- **Processus muet :** soient p_i et p_j deux processus. Le processus p_i est muet pour un processus p_j s'il existe un temps à partir duquel : (1) p_i tombe en panne et s'arrête définitivement de participer au protocole (panne franche), ou (2) p_i n'envoie volontairement plus de messages à p_j, ou (3) p_i envoie uniquement des messages non signés ou incorrectement signés à p_j.

Cette notion de processus muet nous permet de définir la propriété de *complétude de mutisme.*

- **Complétude de mutisme :** il existe un temps à partir du quel tout processus p_i, qui est muet pour un processus p_j, est suspecté pour toujours par p_j.

Par ailleurs, comme la justesse faible ne concerne que les processus corrects, elle reste inchangée. Nous notons $\diamond M$ le nouveau détecteur de défaillances, défini par la *justesse faible* et la *complétude de mutisme.*

4. Consensus vectoriel

L'importance du consensus vient du fait qu'il constitue une composante de base permettant de résoudre d'autres protocoles d'agrément : plusieurs travaux, se situant dans un modèle avec défaillances franches, ont montré que des problèmes tels que l'ordre total, la validation atomique ou encore le synchronisme virtuel pouvaient être résolus par réduction au consensus [1, 4].

Cependant, on constate qu'aucune solution basée sur l'idée de réduction au consensus n'a été proposée, dans le modèle byzantin . Ceci vient du fait que le consensus traditionnel (agrément, validité, terminaison) est remplacé, dans ce modèle, par le consensus strict (agrément, validité stricte, terminaison). Or, comme nous l'avons souligné dans l'introduction, le consensus strict n'impose de décision que si tous les processus corrects ont la même valeur initiale. Cette condition le rend trop restrictif pour être utilisé pour la résolution d'autres protocoles d'agrément.

Afin de relâcher la restriction imposée par le consensus strict, nous définissons le *Consensus vectoriel.* Ce dernier permet à un ensemble de processus corrects, chacun ayant sa propre valeur initiale, de décider sur un vecteur qui contient au moins $f + 1$ valeurs initiales de processus corrects, et ceci en dépit de la présence de processus byzantins. De manière plus formelle, le consensus vectoriel est défini par les propriétés traditionnelles d'agrément et de terminaison, et une nouvelle propriété de **validité vectorielle**.

- **Validité vectorielle :** les processus décident sur un vecteur *vect* de taille n tel que : (1) pour tout i et tout processus p_i correct, $vect[i]$ est soit la valeur initiale de p_i, soit *null*, et (2) au moins $f+1$ éléments de *vect* sont des valeurs initiales de processus corrects.

On peut montrer que toute solution permettant de résoudre le consensus vectoriel permet également de résoudre le consensus strict. Par ailleurs, cette nouvelle spécification rend possible la réduction au consensus vectoriel d'un problème tel que l'ordre total. Toutefois, la discussion de cette réduction sort du cadre de cet article.

5. Principe de l'algorithme du consensus vectoriel

Le consensus vectoriel se déroule en deux étapes. La première est dédiée à la construction, pour chaque processus correct, d'un vecteur de valeurs initiales *signées*. Durant la seconde, chaque processus correct propose son vecteur de valeurs initiales signées à un module de consensus qui résout le consensus traditionnel (agrément, validité, terminaison). Une version complète de l'algorithme du consensus vectoriel ainsi que les preuves sont disponibles dans [2]. Nous nous contentons ici d'esquisser l'algorithme.

5.1. Première étape : construction du vecteur de valeurs initiales signées

Durant cette étape, chaque processus correct p_i envoie sa propre valeur initiale v_i dans un message **signé** (p_i, v_i). Après quoi, il collecte au moins $n-f$ messages similaires d'autres processus et associe le message *null* aux processus dont il ne reçoit rien. Ainsi, le processus p_i construit un vecteur de valeurs initiales signées noté $vect_i$ tel que : $vect_i[j]$ contient (p_j, v_j) si p_i reçoit la valeur initiale de p_j et *null* s'il ne la reçoit pas. Dès que $vect_i$ contient $n-f$ éléments, p_i le propose au module de consensus. On peut montrer que tout vecteur de valeurs initiales signées ainsi construit contient au moins $f+1$ valeurs initiales venant de processus corrects.

5.2. Deuxième étape : module de consensus

Le module de consensus utilisé au sein du consensus vectoriel est dérivé de l'algorithme du *Early Consensus* [9]. Cet algorithme est basé sur le paradigme du coordinateur tournant et se déroule en une ou plusieurs rondes asynchrones. Une ronde est divisée en deux phases. Durant la phase 1, le coordinateur propage son estimation $estimate_c$ à tous les autres processus. Tout processus correct recevant $estimate_c$ pour la première fois la rediffuse à son tour. Dès qu'un processus correct collecte une majorité d'$estimate_c$, il décide cette valeur. Toutefois, si le coordinateur est suspecté par une majorité de processus correct, la phase 2 intervient afin de garantir que, si une décision v est prise dans la ronde actuelle, alors tous les processus corrects commenceront la ronde suivante avec leurs valeurs courantes égales à v.

La prise en compte des défaillances byzantines nous a conduit à utiliser le détecteur de défaillances $\diamond M$ (défini plus haut) et à introduire des mécanismes de contrôle de validité des messages. Le contrôle consiste tout d'abord à détecter les messages corrompus, c-à-d non conformes au protocole. Pour cela, on associe à chaque message un certificat prouvant sa validité. Un message possédant un certificat correct est appelé message *correctement formé*. Le contrôle des messages est également nécessaire pour détecter les messages qui auraient été volontairement omis par des processus byzantins. Ces messages sont appelés messages *manquants*.

5.2.1. Messages *correctement formés*

Lorsque un processus p_j reçoit un message m il doit être en mesure de contrôler la validité de ce dernier. Pour cela, l'expéditeur p_j du message m doit joindre au message un ensemble d'informations redondantes appelé *certificat*. Un certificat est composé de $\lceil (2n+1)/3 \rceil$ messages signés par des processus différents. Ceci le rend infalsifiable, car sa construction implique la participation d'une majorité de processus corrects.

5.2.2. Messages *manquants*

Afin de détecter si certains messages attendus sont volontairement omis, nous exploitons la caractéristique FIFO des canaux de communication et les propriétés suivantes de notre algorithme de consensus :

- dans chaque ronde r, un processus correct envoie un nombre limité de messages aux autres processus. Soit n_{ronde} ce nombre.
- un message m envoyé par un processus p_i dans la ronde r inclut la valeur r.

Lorsqu'un processus p_i correct attend un message m d'un processus p_j dans la ronde r, il effectue le contrôle suivant :

- Si p_i reçoit de p_j un message dont le numéro de ronde r' est supérieur à r, alors la propriété FIFO des canaux permet à p_i de conclure que p_j est un processus byzantin qui a volontairement omis l'envoi du message attendu.
- Si p_i reçoit plus que $r \cdot n_{ronde}$ messages de p_j et qu'il n'a toujours pas reçu m, alors la connaissance du nombre maximal de messages à attendre permet à p_i de conclure que p_j est un processus byzantin qui a omis d'envoyer le message attendu.

6. Conclusion

Dans cet article, nous avons présenté une nouvelle spécification du problème de consensus adaptée au modèle byzantin. La nouvelle spécification se distingue par la propriété de *validité vectorielle*. La validité vectorielle, l'agrément et la terminaison définissent le problème du *consensus vectoriel*. L'objectif de cette nouvelle spécification est de définir un consensus résistant aux défaillances byzantines et pouvant être utilisé pour construire d'autres protocoles d'agrément. L'approche suivie pour résoudre le problème du consensus vectoriel a consisté tout d'abord à définir un détecteur de défaillances adapté au modèle byzantin. Nous avons capturé l'ensemble des comportements pouvant perturber la progression du protocole dans la notion de processus muet. Cette notion a été utilisée pour définir la propriété de complétude de mutisme qui, conjointement à la justesse faible, définissent notre détecteur de défaillances noté $\diamond M$. Finalement, nous avons esquissé un algorithme qui résout le problème du consensus vectoriel. Cet algorithme permet également de résoudre le consensus strict et ceci avec un degré de latence[1] de 4 alors que l'algorithme proposé par Malkhi et Reiter a un degré de latence de 9. L'algorithme complet ainsi que les preuves sont donnés dans [2].

Bibliographie

1. T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *proc. 10th annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, 1991.
2. A. Doudou and A. Schiper. Muteness Failure Detectors for Consensus with Byzantine Processes. TR 97/230, EPFL, Dept d'Informatique, October 1997.
3. M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32:374–382, April 1985.
4. R. Guerraoui and A. Schiper. Consensus service: a modular approach for building agreement protocols in distributed systems. In *IEEE 26th Int Symp on Fault-Tolerant Computing (FTCS-26)*, pages 168–177, June 1996.
5. D. Malkhi and M. Reiter. A High-throughput Secure Reliable Multicast Protocol. In *Proc. 9th Computer Security Foundations Workshop (CSFW96)*, pages 9–17, June 1996.
6. D. Malkhi and M. Reiter. Unreliable Intrusion Detection in Distributed Computations. In *Proc. 10th Computer Security Foundations Workshop (CSFW97)*, June 1997.
7. M.K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proc. 2nd ACM Conf. on Computer and Communications Security*, pages 68–80, November 1994.
8. R.L. Rivest, A. Shamir, and L.Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
9. A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.

[1]La notion de degré de latence a été définie par Schiper dans [9], afin de mesurer le coût d'un algorithme distribué.