Communication Optimalement Stabilisante sur Canaux non Fiables et non FIFO

Shlomi Dolev[1], Swan Dubois[2], Maria Potop-Butucaru[2] et Sébastien Tixeuil[3]

[1] *Université de Ben-Gurion (Israël), dolev@cs.bgu.ac.il*
[2] *UPMC Sorbonne Universités & INRIA (France), {swan.dubois,maria.potop-butucaru}@lip6.fr*
[3] *UPMC Sorbonne Universités & IUF (France), sebastien.tixeuil@lip6.fr*

Un protocole auto-stabilisant a la capacité de converger vers un comportement correct quel que soit son état initial. La grande majorité des travaux en auto-stabilisation supposent une communication par mémoire partagée ou bien à travers des canaux de communication fiables et FIFO. Dans cet article, nous nous intéressons aux systèmes auto-stabilisants à passage de messages à travers des canaux de capacité bornée mais non fiables et non FIFO. Nous proposons un protocole de communication (entre voisins) stabilisant et offrant une tolérance optimale. Plus précisément, ce protocole simule un canal de communication fiable et FIFO garantissant un nombre minimal de pertes, de duplications, de créations et de ré-ordonnancements de messages.

Keywords: Canaux non fiables, canaux non FIFO, auto-stabilisation

1 Motivations et Définitions

L'*auto-stabilisation* [Dij74] est une propriété des systèmes distribués permettant de tolérer des fautes transitoires (*i.e.* de durée finie) de type arbitraire. Plus précisément, un système est dit auto-stabilisant s'il garantit que toute exécution issue d'une configuration arbitraire retrouve en un temps fini un comportement conforme à la spécification du système et ceci sans aide extérieure (humaine ou autre).

Motivations Étant donné que l'auto-stabilisation est une propriété non triviale à satisfaire, une large part des travaux traitant de ce domaine se placent dans un modèle de communication très simple dans lequel tous les processeurs peuvent déterminer de manière atomique l'état de tous leurs voisins (ce modèle de calcul est connu sous le nom de modèle à états). Il est cependant évident que ce modèle n'est pas réaliste et qu'un modèle plus classique comme le modèle asynchrone à passage de messages est plus proche d'un système réel. Dans un tel modèle, les processeurs voisins communiquent par envoi et réception de messages à travers le canal de communication qui les sépare. Il existe des transformateurs permettant de passer de manière automatique du premier modèle au second [Dol00, DIM93] ainsi que des algorithmes écrits directement pour le modèle à passage de messages [DT06, BKM97] mais ceux-ci supposent l'existence d'un protocole de communication entre processeurs voisins. Le protocole de communication (entre voisins) le plus connu est le protocole du bit alterné (PBA). Il a été prouvé que ce protocole fournit des propriétés de stabilisation [AB93, DIM93]. En effet, pour toute exécution du PBA, il existe un suffixe qui satisfait la spécification (*i.e.* le PBA est pseudo-stabilisant [BGM93]). Après les résultats de [GM91, DIM93] qui montrent qu'il est impossible de fournir un protocole de communication avec une mémoire bornée si les canaux sont de capacité non bornée, les travaux récents se sont concentrés sur des systèmes avec des canaux de capacité bornée. Il existe différents protocoles de communication stabilisants à travers des canaux de capacité bornée qui diffèrent par les hypothèses faites sur le système (mémoire bornée, canaux, etc.) mais toutes les solutions connues [BGM93, HNM99, Var00] considèrent des canaux FIFO.

Un défaut commun à tous les protocoles de communication précédents est qu'ils ne fournissent aucune mesure de l'impact quantitatif des fautes transitoires sur les messages transmis. Partant d'une configuration initiale arbitraire, le contenu initial des canaux est lui aussi arbitraire, ce qui peut conduire le protocole à perdre, dupliquer des messages ou bien délivrer de faux messages (qui n'ont pas été envoyés mais résultent des fautes initiales). Du point de vue de l'application qui utilise le protocole de communication, il est

primordial de connaître des bornes sur le nombre de messages pouvant subir de tels aléas. À notre connaissance, seuls [DDNT10, DT06] traitent de ce problème dans une certaine mesure. En effet, ils peuvent être adaptés pour obtenir des protocoles de communication instantanénement stabilisants. La stabilisation instantanée assure que tout message envoyé sera délivré en un temps fini, mais le nombre de messages dupliqués ou de faux messages crées n'est pas étudié.

Contributions Notre contribution dans cet article est double. Dans un premier temps, nous définissons un ensemble de métriques pour mesurer la performance d'un protocole de communication stabilisant et nous donnons des bornes inférieures pour plusieurs d'entre elles. En particulier, nous montrons que tout protocole de communication stabilisant à travers des canaux de capacité bornée non fiables et non FIFO peut être contraint à dupliquer un message, à délivrer un faux message ou à ré-ordonner un message. Dans un second temps, nous proposons un protocole optimal par rapport aux bornes inférieures évoquées précèdement.

Spécification Nous considérons ici un système distribué à passage de messages réduit à deux processeurs : p_i qui sera considéré comme émetteur de messages et p_j qui sera considéré comme récepteur de messages. Le canal de communication séparant p_i et p_j est constitué de deux canaux virtuels de directions opposées. Le premier, (i, j), permet à p_i d'envoyer des messages à p_j tandis que le second, (j, i), permet à p_j d'envoyer des acquittements à p_i. Chacun de ces canaux virtuels est asynchrone (le temps de livraison de tout message est fini mais non borné), a une capacité bornée de c messages (tout envoi de messages lorsque cette borne est atteinte conduit à la perte d'un message arbitraire), non fiable (tout message peut être perdu à un moment arbitraire) mais équitable (tout message envoyé infiniment souvent est reçu infiniment souvent) et non-FIFO (l'ordre d'arrivée des messages est indépendant de l'ordre d'envoi). Il faut noter que, en raison du contexte auto-stabilisant, chaque canal virtuel contient initialement jusqu'à c messages de contenu arbitraire.

La spécification que nous présentons à présent est inspirée de celle de [Lyn96] mais elle est adaptée au contexte auto-stabilisant. Supposons que nous avons une application distribuée qui souhaite envoyer des messages de p_i à p_j. Notre objectif est de fournir un protocole de communication à cette application qui remplit cette tâche de manière transparente malgré les caractéristiques du canal de communication. Cette application *envoit* un message lorsqu'elle demande au protocole de communication de faire parvenir un message depuis p_i vers p_j. Un message est *délivré* à p_j lorsque le protocole de communication fournit ce message à l'application s'exécutant sur p_j. Un message *fantôme* est un message délivré à p_j alors qu'il n'a pas été envoyé par p_i. Un message *dupliqué* est un message délivré plusieurs fois à p_j alors qu'il n'a été envoyé qu'une fois par p_i. Un message *perdu* est un message envoyé par p_i mais jamais délivré à p_j. Un message m est *ré-ordonné* lorsqu'il délivré à p_j avant un message m' alors que m a été envoyé après m' par p_i. Le but d'un protocole de communication stabilisant est alors de fournir des propriétés sur le nombre de messages perdus, dupliqués, fantômes et ré-ordonnés. Nous spécifions notre problème comme suit :

Définition 1 *Un protocole de communication est* $(\alpha, \beta, \gamma, \delta)$*-stabilisant sur des canaux c-bornés s'il remplit les conditions suivantes pour toute exécution issue d'une configuration arbitraire :*
- Dans le pire cas, seuls les α *premiers messages envoyés par* p_i *peuvent être perdus.*
- Dans le pire cas, seuls les β *premiers messages délivrés à* p_j *peuvent être des messages dupliqués.*
- Dans le pire cas, seuls les γ *premiers messages délivrés à* p_j *peuvent être des messages fantômes.*
- Dans le pire cas, seuls les δ *premiers messages délivrés à* p_j *peuvent être des messages ré-ordonnés.*

2 Bornes inférieures

Étant donné que tout protocole de communication doit avoir dans son code une instruction pour délivrer les messages à l'application et que, dans un contexte auto-stabilisant, le compteur ordinal peut être arbitrairement corrompu dans la configuration initiale, il est possible que la première instruction executée par le processeur récepteur soit la livraison d'un message qui n'a jamais été envoyé, *i.e.* un message fantôme. Si, de plus, ce message fantôme est identique à un autre message envoyé par p_i dans l'exécution considérée, ce message peut devenir un message dupliqué ou ré-ordonné. Nous obtenons les résultats suivants.

Théorème 1 *Il n'existe pas de protocole de communication* $(\alpha, \beta, \gamma, \delta)$*-stabilisant sur des canaux c-bornés avec* $\beta = 0$, $\gamma = 0$ *ou* $\delta = 0$.

3 Protocole de communication $(0,1,1,1)$-stabilisant

Nous sommes maintenant en mesure de présenter notre protocole de communication. Celui-ci est composé de deux fonctions : **Send(*m*)** qui est exécutée par p_i à chaque fois qu'il souhaite envoyer un message m (**Send** est bloquant, *i.e.* p_i doit attendre la fin de son exécution avant de commencer à envoyer le message suivant) et **Receive()** qui est exécutée par p_j en continu.

Idée générale L'idée de base de notre protocole est de modifier le PBA de manière à améliorer ses propriétés de tolérance. Si le processeur p_i souhaite envoyer un message m, il envoie celui-ci de manière périodique et p_j acquitte chaque copie de m qu'il reçoit. Le processeur p_j n'est autorisé à délivrer le message m que lorsqu'il en a reçu $c+1$ copies (pour assurer qu'au moins l'une d'entre elles a bien été envoyée par p_i). De plus, m n'est délivré que si la valeur du bit alterné qui lui est associée est différente de celle du dernier message délivré par p_j (de manière à assurer que le message ne soit pas dupliqué car p_i continue d'envoyer m tant qu'il n'a pas reçu suffisament d'acquittements). Afin d'assurer que p_j a reçu au moins $c+1$ copies du message, p_i attend d'avoir reçu $3c+2$ acquittements avant d'arrêter d'envoyer m (en effet, au plus $c+1$ acquittements sont dûs à la configuration initiale tandis que au plus c sont dûs à la présence initiale de messages erronnés dans le canal (i,j)). À ce stade, notre protocole ne garantit pas encore l'absence de pertes de messages à cause de l'utilisation du bit alterné (en effet, si le bit alterné du message et du récepteur ne sont pas initialement synchronisés, le premier message envoyé par p_i peut être perdu). Pour éviter cela, p_i alterne entre l'envoi de messages de synchronisation et de m. Plus précisèment, pour envoyer un message m, p_i commence par envoyer un message de synchronisation (noté $< SYNCHRO >$) jusqu'à recevoir $3c+2$ acquittements avant d'envoyer le message m lui-même jusqu'à en recevoir $3c+2$ acquittements. Il s'ensuit que seul le message de synchronisation est perdu dans le pire des cas.

Présentation détaillée Notre protocole est présenté en Figure 1. La procédure **Send** se contente d'envoyer un message de synchronisation puis le message reçu en paramètre (à l'aide de la fonction auxilliaire **SendMessage**) après avoir alterné la valeur du bit associé. Finalement, elle délivre un acquittement à l'application à l'aide de la fonction **DeliverAck**. La fonction auxilliaire **SendMessage** envoit périodiquement le message à l'aide de la fonction **SendPacket** (qui permet d'envoyer un paquet sur le canal (i,j)) et compte le nombre d'acquittement reçus en faisant appel à la fonction **ReceivePacket** (qui permet de récupérer un message dans le canal (j,i)). Celle-ci s'arrête lorsqu'elle a compté $3c+2$ acquittements.

Send

entrée : m : message à envoyer
variable : ab : booléen donnant la valeur du bit alterné actuelle
01 : $ab := \neg ab$
02 : **SendMessage** $(< SYNCHRO >, ab)$
03 : $ab := \neg ab$
04 : **SendMessage** (m, ab)
05 : **DeliverAck** (m)

SendMessage

entrée : m' : message à envoyer
ab : booléen donnant la valeur du bit alterné associé à m
variable : ack : entier donnant le nombre d'acquittements reçu pour la valeur actuelle de ab
01 : $ack := 0$
02 : *while* $ack < 3c+2$
03 : **SendPacket** (m', ab)
04 : *if* **ReceivePacket** $(ack, (m', ab))$
05 : $ack := ack + 1$;

Receive

variables :
$last\_delivered$: booléen donnant la valeur du bit alterné du dernier message délivré
Q : file de taille $c+1$ de 3-tuples $(m, ab, count)$, où m est un message, ab est une valeur du bit alterné, et $count$ est un entier donnant le nombre de paquets (m, ab) reçus pour m et ab depuis le dernier **DeliverMessage** ou **DropMessage**. L'opérateur [] renvoit un pointeur sur le $count$ associé à son paramètre et place ce 3-tuple en tête de liste
01 : *upon* **ReceivePacket** (m, ab)
02 : $Q[m, ab] := min(Q[m, ab] + 1, c+1)$
03 : *if* $Q[m, ab] \geq c+1$ *then*
04 : *if* $last\_delivered \neq ab$ *then*
05 : *if* $m \neq < SYNCHRO >$ *then*
06 : **DeliverMessage** (m)
07 : *else*
08 : **DropMessage** (m)
09 : $last\_delivered := ab$
10 : $Q := \perp$
11 : **SendPacket** $(ack, (m, ab))$

FIGURE 1: $\mathcal{SDL}$, un protocole de communication $(0,1,1,1)$-stabilisant.

À chaque réception de message (réalisée grâce à la fonction **ReceivePacket**), la procédure **Receive** incrémente le compteur associé au message qu'elle vient de recevoir. Dans le cas où le message a été reçu $c+1$ fois, la file servant à stocker les messages reçus est vidée. Si, de plus, la valeur du bit alterné est différente de celle du dernier message reçu au moins $c+1$ fois, alors le message est soit délivré à l'appli-

cation à l'aide de **DeliverMessage** (s'il s'agit d'un message normal) ou bien détruit à l'aide de la fonction **DropMessage** (s'il s'agit d'un message de synchronisation qui est donc sans intérêt pour l'application). Dans les deux cas, le bit du récepteur est alterné. Tout message reçu est acquitté (à l'aide de la fonction **SendPacket**) avant de traiter le suivant.

Propriétés Nous avons vu précédement que p_i attend de recevoir $3c+2$ acquittements de chaque message pour arrêter de l'envoyer, ce qui garantit que p_j a reçu au moins $2c+2$ copies de ce message (dont au moins $c+1$ réellement envoyées par p_i) et donc que ce message a bien été délivré à p_j si $ab \neq last\_delivered$. Si ce n'est pas le cas, l'usage du message de synchronisation nous garantit que notre protocole ne perd aucun message envoyé par p_i. L'usage du bit alterné nous garantit l'absence de duplication après la première réception (si le premier message reçu est un message fantôme, celui-ci peut être la copie d'un message valide ultérieur, ce qui cause au pire une duplication). Le fait d'attendre de recevoir $c+1$ copies de chaque message avant de le délivrer garantit que seul le premier message délivré peut être un message fantôme. Enfin, le fait qu'un message *m* soit délivré à p_j entre le début et la fin de l'exécution de **Send(m)** par p_i et que les appels à cette fonction soient bloquants pour p_i implique que seul le premier message peut être ré-ordonné (si le premier message reçu est un message fantôme, celui-ci peut être la copie d'un message valide ultérieur, ce qui cause au pire un ré-ordonnancement). En conclusion, nous avons le résultat suivant † :

Théorème 2 *$\mathcal{SDL}$ est un protocole de communication $(0,1,1,1)$-stabilisant à travers des canaux de communication de capacité bornée mais non fiables et non FIFO.*

4 Conclusion

Dans cet article, nous avons introduit des mesures de l'effet de fautes transitoires sur les performances des protocoles de communication entre voisins dans un système à passage de messages. Nous avons ensuite fourni un protocole optimal par rapport à ces mesures dans le cas où les canaux de communications ont une capacité bornée, sont non fiables et non FIFO. Toutefois, notre protocole induit un surcoût de communication ; la question de savoir s'il est possible de conserver cette tolérance optimale aux fautes transitoires en baissant ce surcoût de communication de manière significative est toujours ouverte.

Références

[AB93] Y. Afek and G. Brown. Self-stabilization over unreliable communication media. *Dist. Comp.*, 7(1) :27–34, 1993.

[BGM93] J. Burns, M. Gouda, and R. Miller. Stabilization and pseudo-stabilization. *Dist. Comp.*, 7(1) :35–42, 1993.

[BKM97] J. Beauquier and S. Kekkonen-Moneta. Fault-tolerance and self stabilization : impossibility results and solutions using self-stabilizing failure detectors. *Int. J. Systems Science*, 28(11) :1177–1187, 1997.

[DDNT10] S. Delaët, S. Devismes, M. Nesterenko, and S. Tixeuil. Snap-stabilization in message-passing systems. *Journal of Parallel and Distributed Computing*, 2010.

[DDPBT10] Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Stabilizing data-link over non-fifo channels with optimal fault-resilience. *CoRR*, abs/1011.3632, 2010.

[Dij74] E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Comm. ACM*, 17(11) :643–644, 1974.

[DIM93] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1) :3–16, 1993.

[Dol00] S. Dolev. *Self-stabilization*. MIT Press, 2000.

[DT06] S. Dolev and N. Tzachar. Empire of colonies : Self-stabilizing and self-organizing distributed algorithms. In *OPODIS*, pages 230–243, 2006.

[GM91] M. Gouda and N. Multari. Stabilizing communication protocols. *Trans. Comput.*, 40(4) :448–458, 1991.

[HNM99] R. Howell, M. Nesterenko, and M. Mizuno. Finite-state self-stabilizing protocols in message-passing systems. In *WSS*, pages 62–69, 1999.

[Lyn96] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[Var00] G. Varghese. Self-stabilization by counter flushing. *SIAM J. Comput.*, 30(2) :486–510, 2000.

†. Une preuve complète de ce résultat peut être trouvée dans [DDPBT10].