Stabilisation instantanée dans les systèmes à passage de messages[†]

Sylvie Delaët[1], Stéphane Devismes[2], Mikhail Nesterenko[3], Sébastien Tixeuil[4]

[1] *LRI UMR 8623, Université de Paris-Sud, Orsay, France*
[2] *VERIMAG UMR 5104, Université Joseph Fourier, Grenoble 1, France (supporté par le projet ANR* SHAMAN*)*
[3] *Computer Science Department, Kent State University, USA*
[4] *LIP6 UMR 7606, Université P. et M. Curie, Paris 6, France (supporté par les projets ANR* SHAMAN *et* R-Discover*)*

Nous abordons le problème de la *stabilisation instantanée* dans les systèmes répartis à passage de messages. Notre contribution est double. Tout d'abord, nous montrons que la stabilisation instantanée est impossible pour la plupart des problèmes dans de tels systèmes si nous supposons que la capacité des canaux de communication est finie mais non bornée. Nous montrons ensuite que la stabilisation instantanée devient réalisable si nous connaissons une borne sur la capacité des canaux de communication. Cette dernière contribution est constructive : nous proposons les deux premiers protocoles répartis instantanément stabilisants dans le modèle à passage de messages.

Keywords: Tolérance aux pannes, auto-stabilisation, stabilisation instantanée

1 Introduction

A partir de n'importe quel état initial du réseau[‡], un protocole *instantanément stabilisant* vérifie toujours ses spécifications [BDPV07]. Par définition, un protocole instantanément stabilisant tolère les fautes transitoires[§]. En effet, l'état initial quelconque du système peut être vu comme l'état immédiatement postérieur à la fin des pannes. De ce fait, dans un système réparti sujet aux pannes transitoires, un protocole instantanément stabilisant retrouve un comportement correct *immédiatement* après la fin de celles-ci. Toutefois, un protocole instantanément stabilisant n'offre aucune garantie pour les calculs ayant été tout ou partie exécutés durant les pannes. Bien entendu, toutes les propriétés de sûreté ne peuvent être garanties par la stabilisation instantanée. En fait, la stabilisation instantanée offre des garanties du point de vue *utilisateur* : suite à une demande de calcul de l'utilisateur, le protocole fournit le résultat correct correspondant à cette demande.

La stabilisation instantanée est souvent comparée à l'auto-stabilisation [Dij74] qui assure que le système *finit par* retrouver un comportement correct après la fin des fautes. Par définition, la stabilisation instantanée offre des garanties de sûreté plus fortes que l'auto-stabilisation : un protocole auto-stabilisant peut retourner des résultats incorrects à un nombre non-borné de requêtes « utilisateur » initiées après la fin des fautes.

Les solutions *instantanément stabilisantes* proposées jusqu'à maintenant étaient conçues dans un modèle haut-niveau — le modèle *à états* — où les processus étaient capables de lire la mémoire de leurs voisins et de modifier la leur en une seule étape atomique. Nous proposons ici de nous placer dans un modèle plus réaliste : le modèle *à passage de messages*. Notre contribution est double. Tout d'abord, nous montrons (Section 2) que la stabilisation instantanée est impossible pour une classe importante de problèmes dans des systèmes répartis à passage de messages si nous supposons que la capacité des canaux de communication est finie mais non bornée. Nous montrons ensuite (Section 3) que la stabilisation instantanée devient réalisable

[†] Les résultats présentés dans cet article sont issus de [DDNT08], ces travaux ont été soutenu par le projet ANR SOGEA.
[‡] *I.e.*, mémoire des processus non-initialisée et contenu quelconque des canaux de communication.
[§] *I.e.* des défaillances temporaires du système qui altèrent le contenu d'un ou plusieurs de ses composants.

si nous connaissons une borne sur la capacité des canaux de communication. Cette dernière contribution est constructive : nous proposons les deux premiers protocoles répartis instantanément stabilisants dans le modèle à passage de messages, l'un pour la *propagation d'informations avec retour* et l'autre pour l'*exclusion mutuelle*.

2 Résultat d'impossibilité

Pour démontrer notre résultat d'impossibilité, nous avons introduit la notion de spécification à *sûreté distribuée*. Nous avons ensuite démontré qu'aucun problème ayant une telle spécification n'a de solution instantanément stabilisante dans un système à passage de messages qui n'admet pas de borne sur la capacité de ses canaux de communication.

Intuitivement, une spécification à sûreté distribuée comporte (au moins) une propriété de sûreté qui dépend du comportement local de plusieurs processus, c'est-à-dire, certains comportements locaux aux processus sont valides s'ils sont exécutés séparément tandis qu'ils violent la spécification s'ils sont exécutés concurremment. Par exemple, l'exclusion mutuelle a une spécification à sûreté distribuée : un processus qui demande la section critique doit l'obtenir en un temps fini, mais plusieurs processus demandeurs ne peuvent exécuter la section critique concurremment. La plupart des problèmes distribués, tels les problèmes de synchronisation ou d'allocation de ressources, ont une spécification à sûreté distribuée.

La preuve de notre résultat d'impossibilité est basée sur le fait que dans un système à canaux non-bornés, les fautes transitoires peuvent mener à une configuration contenant un nombre non borné de messages corrompus quelconques. Puisqu'une spécification à sûreté distribuée implique le comportement local de plusieurs processus, elle nécessite que ces processus échangent des messages pour éviter de violer la spécification. Cependant, avec des canaux à capacité non-bornée, un processus ne peut pas déterminer si un message reçu à bien été envoyé par son voisin ou si ce message est le résultat d'une faute. Ainsi les communications sont biaisées et les processus ne peuvent distinguer un comportement correct d'un incorrect.

3 Protocoles

Nous montrons maintenant que la stabilisation instantanée devient réalisable si une borne sur la capacité des canaux est connue. Nous présentons deux solutions instantanément stabilisantes pour la *propagation d'informations avec retour* (*PIR*) et l'exclusion mutuelle. Ces deux protocoles fonctionnent dans un réseau complet de n processus. Les canaux de communication sont supposés non-fiables mais équitables[¶], à capacités bornées et FIFO. Le système est asynchrone. Cependant, tout message qui n'est pas perdu est reçu en un temps fini mais non borné. Si le canal de communication est plein (*i.e.*, sa borne est atteinte) lors de l'envoi d'un message, le message est perdu. Par souci de simplicité, nous considérons que les liens ont une capacité de 1. L'extension de nos travaux à une capacité bornée quelconque est directe en utilisant la méthode présentée dans [APSV91].

3.1 Propagation d'informations avec retour

Le problème du *PIR* est le suivant : suite à une requête, un processus — appelé *initiateur*[‖] — démarre une *phase de diffusion*, c'est-à-dire, il diffuse un message dans le réseau. Par la suite, chaque processus non-initiateur accuse la réception du message à l'initiateur (phase de *retour*). Le *PIR* courant termine une fois que l'initiateur a reçu un accusé de réception de chacun des autres processus. Un protocole de *PIR* nécessite les variables suivantes :

- La variable $\texttt{Req}_p$ est utilisée pour gérer les requêtes au niveau du processus p. $\texttt{Req}_p$ est affectée à `Wait` quand p reçoit une demande. $\texttt{Req}_p$ passe de `Wait` à `In` au démarrage du *PIR*. Enfin, $\texttt{Req}_p$ passe de `In` à `Done` à la fin du *PIR* courant. Notez que $\texttt{Req}_p$ peut repasser à `Wait` (*i.e.*, p peut à nouveau traiter une requête) uniquement lorsque le *PIR* courant est terminé (uniquement lorsque $\texttt{Req}_p = \texttt{Done}$).
- La variable $\texttt{BMes}_p$ contient le message à diffuser.

[¶] *I.e.*, si un processus p envoie une infinité de messages au processus q, alors q en recevra une infinité.

[‖] *N.b*, tout processus peut être initiateur d'un *PIR* et plusieurs *PIR* peuvent être exécutés concurremment.

- Le tableau $\texttt{FMes}_p[1\ldots n-1]$ contient les accusés de réception ($\texttt{FMes}_p[q]$ est utilisé pour stocker les accusés de réception des messages venant de q).

A partir de ces variables, le *PIR* fonctionne comme suit : $\texttt{BMes}_p$ et $\texttt{Req}_p$ sont respectivement affectées à m et `Wait` pour signifier au processus p qu'il y a une nouvelle requête de *PIR* pour le message m. Suite à cette requête, un *PIR* est démarré : $\texttt{Req}_p$ est affecté à `In`. Le *PIR* termine lorsque $\texttt{Req}_p$ est affecté à `Done`. Entre le début et la fin du *PIR*, le protocole génère deux types d'évènements au niveau applicatif : tout d'abord, un évènement « **B-receive**$\langle\ m\ \rangle$ **from** p » (B pour *Broadcast*) par processus $q \neq p$. Lorsque cet évènement arrive, l'application au niveau de q est supposée traiter le message m et ensuite mettre un accusé de réception Ack_m dans $\texttt{FMes}_q[p]$. Le protocole envoie ensuite $\texttt{FMes}_q[p]$ à p, ce qui génère un évènement « **F-receive**$\langle Ack_m\rangle$ **from** q » (F pour *Feedback*) au niveau applicatif de p.

Tout *PIR* démarré après la fin des fautes doit fonctionner correctement en dépit des messages quelconques présents initialement dans les canaux. Des messages peuvent aussi être perdus. Pour contrer les pertes de messages, le protocole envoie régulièrement des duplicatas des messages. Pour gérer ces duplicatas et les messages corrompus présents dans les canaux, nous marquons nos messages avec des valeurs variant de 0 à 4. Deux tableaux sont utilisés pour gérer ces valeurs : le processus p stocke dans $\texttt{State}_p[q]$ la valeur utilisée pour marquer les messages qu'il envoie à q et dans $\texttt{NState}_p[q]$ la marque du dernier message accepté provenant de q.

En utilisant ces deux tableaux, notre protocole fonctionne comme suit. Quand p démarre un *PIR*, il affecte $\texttt{State}_p[q]$ à 0, pour tout q. Le *PIR* courant termine lorsque $\texttt{State}_p[q] \geq 4$ pour tout indice q.

Durant le *PIR*, p envoie régulièrement des messages $\langle \texttt{PIR},\texttt{BMes}_p,\texttt{FMes}_p[q],\texttt{State}_p[q],\ \texttt{NState}_p[q]\rangle$ à tous ses voisins q vérifiant $\texttt{State}_p[q] < 4$. Lorsqu'un processus q reçoit $\langle \texttt{PIR},B,F,x,-\rangle$ de p, il affecte $\texttt{NState}_q[p]$ à x. Ensuite, q envoie $\langle \texttt{PIR},\texttt{BMes}_q,\texttt{FMes}_q[p],\texttt{State}_q[p],\ \texttt{NState}_q[p]\rangle$ à p si $x < 4$. Enfin, p incrémente $\texttt{State}_p[q]$ seulement quand il reçoit $\langle \texttt{PIR},B,F,-,x\rangle$ de q avec $x = \texttt{State}_p[q]$ et $x < 4$.

L'idée principale de l'algorithme est la suivante : supposons que p démarre un *PIR* du message m. Tant que $\texttt{State}_p[q] < 4$, $\texttt{State}_p[q]$ est incrémenté seulement lorsque p reçoit un message $\langle \texttt{PIR},B,F,-,x\rangle$ de q tel que $x = \texttt{State}_p[q]$. Donc, $\texttt{State}_p[q]$ sera égale à 4 seulement après que p aura reçu des messages de q avec les valeurs 0,1,2 et 3. Or, initialement, il y a au plus un message dans le canal de p vers q et au plus un message dans le canal de q vers p. Donc, ces messages peuvent au plus causer deux incrémentations de $\texttt{State}_p[q]$. Enfin, la valeur initiale de $\texttt{NState}_q[p]$ peut causer au plus une incrémentation de $\texttt{State}_p[q]$. A partir du moment où $State_p[q] = 3$, nous avons la certitude que p incrémentera $State_p[q]$ seulement après avoir reçu un message envoyé par q consécutivement à la réception du message diffusé par p : ce message est un accusé de réception de m.

Il reste à voir quand générer les évènements **B-receive** et **F-receive** :

- Chaque processus q reçoit au plus 4 copies du message diffusé par p. Mais, q génère un seul évènement **B-receive** par message de diffusion, lorsqu'il affecte $\texttt{NState}_q[p]$ à 3.
- Après le démarrage d'un *PIR*, p est sûr de recevoir un accusé de réception correct de q lorsqu'il reçoit $\langle \texttt{PIR},B,F,-,x\rangle$ de q avec $x = \texttt{State}_p[q] = 3$. p génère un évènement **F-receive** uniquement lorsque $\texttt{State}_p[q]$ passe de 3 à 4.

3.2 Exclusion mutuelle.

Un mécanisme d'*exclusion mutuelle* assure qu'une section spéciale du code appelée *section critique* est exécutée par au plus un processus à la fois. Un protocole instantanément stabilisant d'exclusion mutuelle assure (seulement) que les processus qui demandent la section critique après la fin des fautes l'exécuteront de manière exclusive. Cela signifie, en particulier, que nous ne garantissons pas la sûreté pour les processus ayant fait leur demande pendant la période de fautes.

Notre protocole est appelé le protocole $\mathcal{ME}$. $\mathcal{ME}$ utilise une variable $\mathcal{ME}.\texttt{Req}$ qui a le même rôle que la variable `Req` du *PIR*. L'idée de notre protocole est la suivante : nous utilisons les identités des processus. Le processus ayant la plus petite identité est appelé le processus *leader*. Le leader décide — en utilisant la variable `Val` — quel processus peut exécuter la section critique. `Val` prend sa valeur dans $\{0\ldots n-1\}$ et nous supposons que chaque processus numérote localement ses canaux de communication de 1 à $n-1$. Un processus p est autorisé à exécuter la section critique si p est le leader et $\texttt{Val}_p = 0$ ou si p n'est pas

le leader et la valeur de la variable `Val` du leader désigne (suivant l'ordre local du leader) le canal liant le leader à p. Quand un processus apprend qu'il peut accéder à la section critique : (1) il s'assure qu'aucun autre processus ne peut exécuter la section critique ; (2) il exécute ensuite la section critique s'il le souhaite ; (3) enfin, il avertit le leader lorsqu'il termine l'étape 2 pour que le leader désigne un autre processus. $\mathcal{ME}$ s'exécute par phases, de la phase 0 à 4. Lorsqu'un processus est demandeur, il ne peut accéder à la section critique qu'après avoir exécuté la phase 0 : p ne peut accéder à la section critique que si $\mathcal{ME}.\texttt{Req}_p = \texttt{In}$ et $\mathcal{ME}.\texttt{Req}_p$ passe de `Wait` à `In` seulement lors de la phase 0. Ainsi, notre protocole assure qu'après avoir exécuté la phase 0, un processus exécute toujours la section critique seul. Les 5 phases sont décrites ci-dessous.

Phase 0. Quand un processus p est en phase 0, il démarre un *PIR* pour collecter les identités des autres processus et évaluer qui est le leader. De plus, il affecte $\mathcal{ME}.\texttt{Req}_p$ à `In` si $\mathcal{ME}.\texttt{Req}_p = \texttt{Wait}$. Il passe ensuite à la phase 1.

Phase 1. Lorsqu'un processus p est en phase 1, il attend la fin du *PIR* lancé précédemment. Puis, il démarre un *PIR* du message `ASK` pour apprendre s'il est autorisé à accéder à la section critique. Puis, il passe à la phase 2. A la réception d'un message `ASK` depuis le canal de p, un processus q répond par `YES` si $\texttt{Val}_q$ désigne le canal de p, `NO` sinon. Chaque processus ne tient compte que de la réponse du leader.

Phase 2. Lorsqu'un processus p est en phase 2, il attend la fin du *PIR* démarré en phase 1. Après ce *PIR*, il sait s'il est autorisé à exécuter la section critique. S'il l'est, il démarre un *PIR* du message `EXIT`. Ensuite il passe à la phase 3. Le but du message `EXIT` est de forcer les autres processus à redémarrer en phase 0. Ce dernier *PIR* assure qu'un autre processus que p ne pourra exécuter la section critique avant que p ait notifié au leader qu'il a quitté la section critique. En effet, dû à la configuration initiale quelconque, un processus $q \neq p$ peut croire qu'il est autorisé à entrer en section critique si q n'est jamais passé par la phase 0. Inversement, après être repassé par la phase 0, q ne reçoit aucune autorisation avant que p ait notifié au leader qu'il a quitté la section critique.

Phase 3. Lorsqu'un processus p est en phase 3, il attend la fin du *PIR* courant, puis s'il est autorisé à entrer en section critique, (1) il entre en section critique et passe $\mathcal{ME}.\texttt{Req}_p$ de `In` à `Done` si $\mathcal{ME}.\texttt{Req}_p = \texttt{In}$ puis soit (2.a) p est le leader et $\texttt{Val}_p$ passe de 0 à 1 ou (2.b) p n'est pas le leader et démarre un *PIR* du message `EXITCS` pour notifier au leader qu'il est sorti de la section critique et enfin (3) il passe à la phase 4. A la réception d'un message `EXITCS`, le leader incrémente sa variable `Val` modulo $n+1$ pour autoriser un autre processus à entrer en section critique.

Phase 4. En phase 4, un processus attend la fin du PIR précédent puis retourne à la phase 0.

4 Conclusion

Nous avons démontré que la *stabilisation instantanée* est impossible pour une large classe de spécifications dans les systèmes à passage de messages où la capacité des canaux est finie mais non bornée. Cependant, nous avons aussi démontré que la stabilisation instantanée devient réalisable si nous supposons une borne sur la capacité : nous avons proposé les deux premiers protocoles instantanément stabilisants dans le modèle à passage de messages. Les perspectives immédiates de ce travail sont d'étendre nos résultats à des réseaux plus complexes, *e.g.*, avec des topologies quelconques, des processus sujets à des types de pannes supplémentaires (*e.g.*, des pannes franches).

Références

[APSV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction (extended abstract). In *FOCS*, pages 268–277, 1991.

[BDPV07] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-stabilization and pif in tree networks. *Distributed Computing*, 20(1) :3–19, 2007.

[DDNT08] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-stabilization in message-passing systems. Research Report 6446, INRIA, February 2008.

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