Acheminement de Messages Instantanément Stabilisant : Solution Optimale sur les Topologies Linéaires *

Alain Cournier†, Swan Dubois‡, Anissa Lamani†, Franck Petit‡ et Vincent Villain†

† MIS, Université de Picardie Jules Verne, France
‡LIP6, UMR7606, UPMC Sorbonne Universités & INRIA. France

Résumé
Nous présentons un algorithme instantanément stabilisant pour l'acheminement de messages. Contrairement aux algorithmes précédemment proposés pour ce problème, notre algorithme nécessite un nombre de tampons qui est indépendant de tout paramètre global du réseau (taille, diamètre,...). En effet, nous montrons que le problème peut être résolu en utilisant quatre tampons par lien de communication. La topologie du réseau considérée est une chaîne de nœuds qui peut être un overlay d'un réseau dynamique à large échelle comme un système pair à pair, une grille, etc. Nous montrons que, sous certaines conditions, le protocole proposé tolère un réseau dynamique.

Mots-clés : Acheminement de messages, dynamicité, réseaux pair à pair, stabilisation instantanée

1. Introduction

Nous avons assisté ces dernières années à l'évolution de systèmes distribués à *large échelle*, en particulier, les *réseaux pair-à-pair*. Parmi les grands défis à considérer dans le développement de tels systèmes se trouvent (i) le *passage à l'échelle* et (ii) la prise en compte de la *dynamicité*, c'est-à-dire la propension du système à résister aux changements topologiques (ajout ou suppression de nœuds ou de liens).
Un protocole *auto-stabilisant* [9] garantit que si le système se trouve dans un état arbitraire, il retrouve de lui-même un comportement répondant aux spécifications en un temps fini. L'auto-stabilisation se révèle donc être une approche naturelle pour développer des protocoles résistants aux fautes transistoires ou aux changements topologiques. Ces protocoles supportent d'autant mieux le passage à l'échelle et la dynamicité s'ils ne requièrent pas la connaissance d'un paramètre global du système, par exemple le nombre de nœuds (noté n) ou le diamètre du réseau (noté D).
Nous nous intéressons au problème de la *communication point-à-point*. Certains nœuds du système (appelés nœuds émetteurs) souhaitent envoyer des messages à d'autres (appelés nœuds destinataires). Le problème consiste à délivrer tous ces messages en un temps fini. Ce problème englobe en réalité deux sous problèmes : (i) le problème du routage —calcul du chemin à suivre par les messages de l'émetteur vers le destinataire— et (ii) le problème d'acheminement de messages —gestion des ressources allouées au transport de messages. Il existe des solutions auto-stabilisantes permettant de résoudre le problème du routage, par exemple [10, 11, 12].
Dans cet article, nous nous concentrons sur le second problème, à savoir le problème d'acheminement de message. Dans la suite, nous supposons l'existence d'un algorithme de calcul de tables de routage auto-stabilisant. L'émetteur et le récepteur d'un message n'étant pas nécessairement voisins, chaque message doit être stocké temporairement sur chaque nœud par lequel il transite. Il s'agit donc de concevoir des protocoles qui contrôlent les mécanismes permettant aux messages d'être transmis d'un noeud à l'autre sur le chemin déterminé par l'algorithme de routage. Ce stockage temporaire de messages est réalisé à l'aide d'espaces mémoire appelés tampons. La mémoire de chaque nœud étant finie, ceux-ci ne peuvent allouer qu'un nombre fini de tampons pour le transport des messages. Ce nombre fini de tampons par nœud introduit un certain nombre de problèmes, les plus délicats étant la *famine* —un noeud ne peut jamais générer de message— et l'*interblocage* —un ensemble de messages s'empêchent mutuellement

* Ce travail est financé par l'ANR "SPADES".

d'avancer—, ce qui interdit de réduire arbitrairement le nombre de tampons. Notre objectif devient alors de trouver le nombre minimum de tampons nécessaires pour résoudre le problème.

Des solutions auto-stabilisantes ont été proposées pour résoudre le problème d'acheminement de message dans [3, 13]. L'objectif de notre travail est de proposer des solutions instantanément stabilisantes pour ce problème. Un protocole instantanément stabilisant [1] apporte une plus forte garantie qu'un protocole auto-stabilisant car il assure que, quel que soit l'état initial du système, ce dernier répond toujours à ses spécifications. Autrement dit, un protocole instantanément stabilisant est un protocole auto-stabilisant qui stabilise en un temps nul.

Pour le problème d'acheminement de messages, la stabilisation instantanée garantit que tout message émis sera délivré à sa destination en un temps fini alors que l'auto-stabilisation garantit que seulement un nombre fini de messages ne sera pas délivré. Les solutions instantanément stabilisantes connues à ce jour [7, 8] ne conviennent pas aux systèmes à large échelle car elles nécessitent un nombre de tampons par nœud dépendant de la taille [7] ou du diamètre [8] du réseau.

Nous présentons dans cet article un protocole instantanément stabilisant pour l'acheminement de messages sur une chaîne qui utilise un nombre de tampons indépendant de tout paramètre global du réseau (quatre tampons par lien de communication). Nous montrons aussi que, sous certaines conditions, notre protocole peut tolérer la dynamicité. Nous espérons que ce protocole sera une première étape pour obtenir des résultats similaires sur des topologies plus générales.

Le présent article est structuré de la manière suivante : dans la section 2, nous définissons le modèle et quelques termes qui vont être utilisés par la suite. Dans la section 3, nous présentons notre protocole instantanément stabilisant pour l'acheminement de messages. La propriété de tolérance à la dynamicité dynamicité sera discutée dans la section 4. Nous conclurons l'article dans la section 5.

2. Modèle et définitions

Modèle. Le réseau est représenté par un graphe connecté non orienté $G = (V, E)$. V est l'ensemble des nœuds (ou processeurs) et E l'ensemble des liens de communication bidirectionnels. Un lien (p, q) existe si et seulement si les deux processeurs p et q sont voisins. Tout nœud est capable de distinguer chacun de ses liens. Pour simplifier la présentation, nous faisons référence au lien (p, q) par l'étiquette q dans le code de p. Nous notons N_p l'ensemble des voisins d'un processeur p. Dans la suite, nous considérons que le réseau est une chaîne de n nœuds et que les nœuds portent des identités. Par souci de simplification, nous notons par $p_0, p_1, \ldots, p_{n-1}$ l'ensemble des identités d'une extrémité à l'autre de la chaîne.

Nous considérons dans cet article le modèle classique à mémoire partagée introduit par Dijkstra [9] connu sous le nom de modèle à états. Dans ce modèle, les communications entre voisins sont modelisées par la lecture directe des variables des voisins plutôt que par l'échange de messages. Le programme de chaque processeur est constitué d'un ensemble de variables partagées (ou simplement "variables" lorsqu'il n'y a pas d'ambigüité) et un nombre fini d'actions. Chaque nœud peut uniquement écrire dans ses propres variables et lire ses propres variables et celles de ses voisins. Une action est constituée ainsi : $< \mathtt{Label} >::< \mathtt{Garde} >\rightarrow< \mathtt{Action} >$. La `garde` d'une action est un prédicat sur les variables de p et celles de ses voisins. Une `action` est une séquence d'instructions qui met à jour une ou plusieurs variables de p. L'état d'un nœud est défini par les valeurs de chacune de ses variables. Une configuration est le produit des états de tous les nœuds du système.

A chaque étape, chaque nœud évalue ses gardes. Il est dit *activable* si l'une d'elles est vraie. Il est alors autorisé à exécuter l'*action* correspondante. Nous supposons que les *exécutions* du système (séquences d'étapes) sont gérées par un *ordonnanceur* : à chaque étape, il sélectionne au moins un processus activable pour que celui-ci exécute sa règle. Cet ordonnanceur permet de modéliser l'asynchronisme du système. La seule hypothèse que nous faisons sur l'ordonnancement est qu'il est *faiblement équitable*, c'est-à-dire qu'aucun processus ne peut être infiniment longtemps activable sans être choisi par l'ordonnanceur.

Stabilisation instantanée. Soit $\mathcal{T}$ une tâche, et $S_{\mathcal{T}}$ une spécification de $\mathcal{T}$. Un protocole P est dit instantanément stabilisant pour $S_{\mathcal{T}}$ si et seulement si toute exécution de P issue d'une configuration arbitraire satisfait $S_{\mathcal{T}}$.

Le problème d'acheminement de message. Étant donné qu'il s'agit du modèle de commutation le plus répandu, nous nous plaçons dans un système à commutation de messages. Dans un tel modèle, tout tampon peut contenir un message en entier et tout message peut être contenu dans un seul tampon. Le principe de commutation est le suivant : tout message est intégralement stocké sur chaque processeur avant d'être relayé au suivant sur le chemin déterminé par l'algorithme de routage. Le message peut être supprimé du tampon du processeur lorsqu'il a été transmis au suivant. Nous donnons ci-dessous la spécification SP du problème d'acheminement de message :
Specification [SP]. Un protocole P satisfait SP si et seulement si :
(i) Tout message peut être émis en un temps fini et (ii) tout message valide (c'est-à-dire effectivement émis par un processeur) est délivré à sa destination une et une seule fois en un temps fini.

Graphe de Tampons. Afin de concevoir notre algorithme instantanément stabilisant, nous nous basons sur une structure appelée *graphe de tampons* [14]. Cette structure est un graphe orienté dont les noeuds sont un sous-ensemble des tampons du réseau et les arcs connectent des paires de tampons. La présence d'un arc entre deux tampons indique que des messages peuvent être transmis d'un tampon à un autre. Les arcs ne sont autorisés qu'entre tampons d'un même noeud ou entre tampons de noeuds voisins.

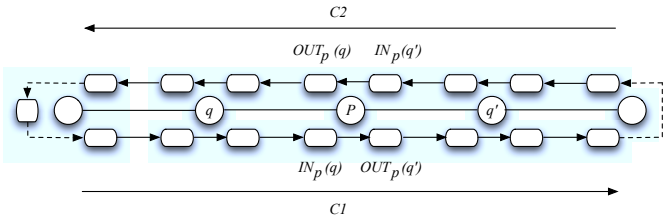


FIG. 1 – Graphe des tampons.

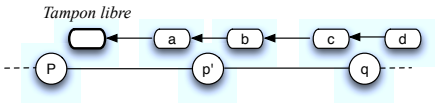
Nous définissons dans ce qui suit le graphe de tampons utilisé par notre protocole (Figure 1). Tout processeur p est doté de deux tampons par lien (p, q). Les nœuds formant les extrémités de la chaîne n'ont donc que deux tampons tandis que les autres en ont quatre. Chaque processeur possède deux tampons d'entrée notés $\mathrm{IN}_p(q)$ et $\mathrm{IN}_p(q')$ ainsi que deux tampons de sortie notés $\mathrm{OUT}_p(q)$ et $\mathrm{OUT}_p(q')$, $q, q' \in \mathrm{N}_p$ et $q \neq q'$ (un pour chaque voisin). La génération d'un message m s'effectue toujours dans un tampon de sortie du lien (p, q) (q étant le processeur indiqué par la table de routage pour m). Soit $nb(m, b)$ le prochain tampon du message m actuellement stocké dans le tampon b, $b \in \{\mathrm{IN}_p(q), \mathrm{OUT}_p(q)\}$, $q \in \mathrm{N}_p$. Nous avons les propriétés suivantes :

1. $nb(m, \mathrm{IN}_p(q)) = \mathrm{OUT}_q(p)$
2. $nb(m, \mathrm{OUT}_p(q)) = \mathrm{IN}_q(p)$

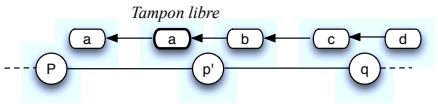
3. Acheminement de Messages

Dans cette section, nous présentons notre solution instantanément stabilisante pour l'acheminement de messages pouvant tolérer la corruption des tables de routage dans la configuration initiale.
Afin de simplifier l'explication de l'algorithme, nous supposons dans un premier temps que le réseau ne contient pas de messages n'ayant pas de destinataire. Cette restriction est supprimée plus tard —cf. section 4. Nous supposons la présence d'un algorithme de calcul des tables de routage auto-stabilisant, $Rtables$. $Rtables$ s'exécute en parallèle de notre algorithme. Nous supposons également que tout processus p a accès aux tables de routage grâce à la fonction $Suivant_p(d)$ qui retourne l'identité du voisin par lequel le message doit transiter afin d'atteindre sa destination d. Pour atteindre notre objectif, nous définissons notre graphe de tampons sur la chaîne comme étant deux chaînes, une pour chaque direction C1 et C2 —cf. Figure 1.

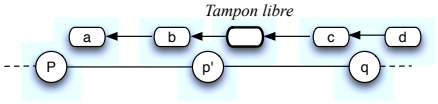
L'idée générale de l'algorithme est la suivante : lorsque un processeur souhaite générer un message, il consulte sa table de routage pour déterminer le voisin par lequel le message doit transiter afin d'atteindre sa destination. Rappelons que la génération de message s'effectue toujours dans le tampon de sortie. Une fois le message dans le tampon de sortie, il suit le graphe de tampons (selon la direction du graphe de tampons et non pas celle indiquée par la table de routage). Afin d'éviter la duplication de message, une couleur est attribuée à chaque message. Notons que si le message arrive à avancer (pas d'interblocage), soit (i) il finit par rencontrer sa destination et être consommé, soit (ii) il atteind le tampon d'entrée du processeur qui est l'extrémité de la chaine. Dans le second cas, si le processeur qui est à l'extrémité de la chaîne n'est pas le destinataire du message, ce qui signifie que le message a été généré dans la mauvaise direction. L'idée est de lui faire faire demi-tour en le copiant dans le tampon de sortie du même processeur —cf. Figure 1.
Soulignons que si les tables de routage sont correctes et stables et que tous les messages sont dans la bonne direction, alors ces derniers peuvent avancer dans C1 et C2 sans être interbloqués. Cependant, dans le cas contraire —les tables de routage ne sont pas correctes ou des messages vont dans la mauvaise direction—, des situations d'interblocage peuvent apparaître si aucun mécanisme de contrôle n'est instauré. Par exemple, supposons que dans l'état initial, tous les tampons soient pleins et contiennent des messages différents de manière à ce qu'aucun d'eux ne puisse être consommé. Il est clair que, dans cette configuration, nous sommes en présence d'un interblocage compte tenu du fait qu'aucun message ne peut avancer dans le système. Afin de débloquer la situation, nous devons supprimer au moins un message. Cependant, comme l'objectif est de proposer une solution instantanément stabilisante (aucun message valide ne peut être supprimé), nous devons introduire des mécanismes de contrôle qui nous permettent d'éviter d'atteindre une telle configuration dynamiquement. Dans notre cas, nous avons choisi d'utiliser l'algorithme du PIR (propagation d'information avec retour) proposé dans [1]. Une version adaptée à notre problème est proposée dans l'algorithme 1.
Avant d'expliquer comment nous allons exploiter le PIR, revenons sur la progression de message dans le graphe de tampons. Un tampon est dit libre si et seulement s'il ne contient aucun message ou s'il contient le même message que le tampon d'entrée auquel il est relié. Par exemple, si $\mathrm{IN}_p(q) = \mathrm{OUT}_q(p)$ alors $\mathrm{OUT}_q(p)$ est dit *libre*. Dans le cas contraire, le tampon est dit *occupé*. la progression des messages dans le graphe de tampons implique le remplissage et la libération de tampons, c'est-à-dire que chaque tampon est alternativement libre et occupé. Ce mécanisme induit clairement que des emplacements libres se déplacent dans le graphe de tampons, un emplacement libre correspondant à un tampon libre à un instant donné. Le mouvement des emplacements libres est illustré dans la Figure 2[2]. Remarquons que les emplacements vides avancent dans la direction opposée des messages. Ceci constitue la propriété principale du contrôle instauré par le PIR.



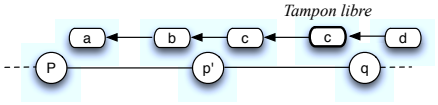
(a) Le tampon en entrée de p est libre. Le nœud p peut donc copier le message a.



(b) Le tampon de sortie de p est libre. Le nœud p′ peut donc copier le message b.



(c) Le tampon en entrée de p′ est libre. Le nœud p′ peut donc copier le message c.



(d) Le tampon de sortie de q est libre. Le nœud q peut donc copier le message d.

FIG. 2 – Le mouvement des emplacements libres.

[2] Dans l'algorithme, les actions b et c sont exécutées dans la même action.

Nous allons maintenant expliquer comment nous utilisons le PIR pour éviter un interblocage. Dans la suite de cette explication et pour des raisons de simplicité, seul p_0 dispose d'un tampon supplémentaire noté EXT, utilisé pour réaliser le changement de direction des messages.
Lorsqu'un message m est présent dans le tampon d'entrée du processeur p_0 et que celui-ci est dans la mauvaise direction, p_0 le copie dans son tampon supplémentaire EXT, libérant ainsi son tampon d'entrée, lequel devient libre. Le nœud p_0 initie alors un PIR. Le rôle du PIR est d'escorter l'emplacement vide du tampon d'entrée de p_0 au tampon de sortie de p_0, permettant ainsi de copier le message le message m qui est dans EXT dans le tampon de sortie de p_0. Une fois le message dans le tampon de sortie de p_0, nous sommes assurés qu'il est désormais dans la bonne direction (étant donné qu'il était précèdement dans la mauvaise direction). Une fois que le PIR est initié, aucun message ne peut être généré dans l'emplacement vide escorté par ce dernier. L'emplacement vide progresse en même temps que la phase de propagation du PIR. Nous sommes donc assurés que le tampon de sortie de l'autre extrémité de la chaine p_{n-1} se libére en un temps fini. De même, le message étant dans le tampon d'entrée de p_{n-1} est soit consommé —dans le cas où p_{n-1} en est le destinataire— soit copié dans le tampon de sortie de celui-ci. En effectuant cette opération, p_{n-1} initie la seconde phase du PIR (Retour). De la même manière, l'emplacement vide (étant désormais dans le tampon de sortie de p_{n-1}) est escorté par le retour du PIR. De proche en proche, le tampon de sortie de p_0 finit par contenir l'emplacement vide escorté par le PIR, permettant ainsi de mettre le message m qui est toujours contenu dans le tampon supplémentaire dans la bonne direction.
Notons que l'objectif du PIR étant de ramener un emplacement vide dans le tampon de sortie de p_0, la propagation du PIR est stoppée dès qu'il rencontre un tampon vide dans la chaîne C2, la phase retour est alors initiée. En d'autres termes, le PIR n'atteint pas l'autre extrémité de la chaîne dans ce cas de figure.
Remarquons que si un message m est présent dans le tampon supplémentaire de p_0 alors qu'aucun PIR n'a été initié, alors m était déjà dans le système dans l'état initial. Par conséquent, il s'agit d'un message non valide, c'est-à-dire n'ayant pas été généré par un processeur. Nous pouvons donc le supprimer sans risque de perdre un message valide. De la même manière, lorsqu'un PIR s'exécute et qu'à la fin du PIR aucun emplacement vide n'est récupéré dans le tampon de sortie de p_0, alors le message contenu dans le tampon supplémentaire de p_0 n'est pas un message valide et peut donc être supprimé.
Dans l'explication ci-dessus, nous avons supposé que seul p_0 disposait d'un tampon suppplémentaire EXT et que par conséquent, lui seul pouvait initier des PIR. Notons que dans la description informelle de notre algorithme, nous supposons la présence d'un processeur special p_0. Ce dernier peut initier des PIR et possède un tampon supplémentaire qui lui permet de faire faire demi-tour aux messages présents dans son tampon d'entrée dont il n'est pas destinataire (soulignons que les autres processeurs de la chaîne ne connaissent pas sa position). En fait, il est facile d'avoir un mécanisme symétrique, c'est-à-dire que les deux extrémités jouent le même rôle. Il suffit pour cela d'utiliser des variables différentes et indépendantes.

L'algorithme 2 présente notre mécanisme d'acheminement de message pour un processeur p. Le mécanisme du PIR est rappelé dans l'algorithme 1. Dans ce qui suit, nous présentons les différentes données et variables utilisées. Le caractère `'?'` dans les prédicats et les deux algorithmes signifie "*n'importe quelle valeur*". Le caractère ϵ désigne un tampon vide.

- **Message**
  - (m, d, c) : m correspond au message à transmettre, d représente l'identité de la source. En plus de m et d chaque message dispose d'un champ supplémentaire c qui est un entier dans $\{0, 1\}$, alternativement donné aux messages pour éviter leur duplication.
- **Variables**
  - *Pour l'algorithme 1 (*PIR*)*
    - $S_p = (P \vee R \vee N, q)$ fait référence à l'état du processeur p (P, R, et N correspondent respectivement à Propagation, Retour et Nettoyé), q est un pointeur sur un voisin de p ou NULL.
  - *Pour l'algorithme 2 (acheminement)*
    - $IN_p(q)$: Le tampon d'entrée du processeur p associé au lien (p, q).
    - $OUT_p(q)$: Le tampon de sortie du processeur p associé au lien (p, q).
    - EXT_p : le tampon supplémentaire si le processeur p est à l'extrémité de la chaîne.

- **Entrées/Sorties**
  - $Demande_p$: Booléen autorisant la communication avec la couche supérieure, mis à VRAI par l'application et à FAUX par l'algorithme d'acheminement.
  - PIR-$Demande_p$: C'est un booléen permettant la communication entre l'algorithme du PIR et de l'acheminement de message. Il est mis à VRAI par l'algorithme d'acheminement et à FAUX par l'algorithme du PIR.
  - La variable S_p de l'algorithme du PIR est une entrées de l'algorithme d'acheminement.
- **Procédures**
  - $Suivant_p(d)$: Retourne l'identité du processeur voisin de p indiqué par la table de routage pour la destination d.
  - $Livré_p(m)$: Délivre le message m à la couche supérieure de p.
  - $choisir(c)$: Choisit une couleur pour le message m qui est différente de celle des messages contenus dans les tampons reliés à celui qui va contenir m.
- **Liste des prédicats**
  - $Consommation_p(q, m) : IN_p(q) = (m, d, c) \wedge d = p \wedge OUT_q(p) \neq (m, d, c)$
  - $feuille_p(q) : S_q = (P, ?) \wedge (\forall\, q' \in N_p/\{q\}, S_{q'} \neq (P, p) \wedge (Consommation_p(q) \vee OUT_p(q') = \epsilon \vee OUT_p(q') = IN_{q'}(p)))$.
  - $NO\text{-}PIR_p : S_p = (N, NULL) \wedge \forall q \in N_p, S_q \neq (P, ?)$.
  - *init-PIR* : $S_p = (N, NULL) \wedge (\forall q \in N_p, S_q = (N, NULL)) \wedge$ $PIR\text{-}Demande_p = VRAI$.
  - $Inter\text{-}trans_p(q) : IN_p(q) = (m, d, c) \wedge d \neq p \wedge OUT_q(p) \neq IN_p(q) \wedge (\exists q' \in N_p/\{q\}, OUT_p(q') = \epsilon \vee OUT_p(q') = IN_{q'}(p))$.
  - $interne_p(q) : p \neq p_0 \wedge \neg\, feuille_p(q)$.
  - $Change\text{-}Route_p(m) : p = p_0 \wedge IN_p(q) = (m, d, c) \wedge d \neq p \wedge EXT_p = \epsilon \wedge OUT_q(p) \neq IN_p(q)$.
  - $\forall$ TAction $\in N, P$, nous définissons $TAction\text{-}initiateur_p$ comme étant le prédicat : $p = p_0 \wedge$ (la guarde de TAction dans p est vérifiée).
  - $\forall$ Tproc $\in \{interne, feuille\}$ et TAction $\in \{P, R\}$, T-Action-$Tproc_p(q)$ est définie par le prédicat : $Tproc_p(q)$ est VRAI $\wedge$ TAction de p est vérifiée.
  - PIR-$Synchro_p(q) : (P_q\text{-}interne_p \vee R_q\text{-}feuille_p \vee R_q\text{-}interne_p) \wedge S_q = (P, ?)$.

- **Initiateur (p_0)**
  - **P-Action : :** *init-PIR* $\rightarrow S_p := (P, -1)$, $PIR\text{-}Demande_p := FAUX$.
  - **N-Action : :** $S_p = (P, -1) \wedge \forall q \in N_p, S_q = (R, ?) \rightarrow S_p := (N, NULL)$.

- **Processeurs feuilles :** $feuille_p(q) = VRAI \vee |N_p| = 1$
  - **R-Action : :** $S_p = (N, NULL) \rightarrow S_p := (R, q)$.
  - **N-Action : :** $S_p = (R, ?) \wedge \forall q \in N_p, S_q = (R \vee N, ?) \rightarrow S_p := (N, NULL)$.

- **Processeurs internes**
  - **P-Action : :** $\exists! q \in N_p, S_q = (P, ?) \wedge S_p = (N, ?) \wedge \forall q' \in N_p/\{q\}, S_{q'} = (N, ?) \rightarrow S_p := (P, q)$.
  - **R-Action : :** $S_p = (P, q) \wedge S_q = (P, ?) \wedge \forall q' \in N_p/\{q\}, S_{q'} = (R, ?) \rightarrow S_p := (R, q)$.
  - **N-Action : :** $S_p = (R, ?) \wedge \forall q' \in N_p, S_{q'} = (R \vee N, ?) \rightarrow S_p := (N, NULL)$.

- **Correction (pour tout processeur)**
  - **Correct1 : :** $S_p = (P, q) \wedge S_q = (R \vee N, ?) \rightarrow S_p := (N, NULL)$.
  - **Correct2 : :** $feuille_p(q) \wedge S_p = (P, q) \rightarrow S_p := (R, q)$.

Algorithme 1: PIR

La preuve formelle de notre solution est omise du fait du manque d'espace [3]. Nous en présentons cependant l'idée générale. Notre protocole se démontre en 3 étapes. Nous montrons tout d'abord qu'à partir d'une configuration arbitraire, notre protocole est sans interblocage. Nous démontrons ensuite que tout processeur est capable de générer un message en un temps fini, autrement dit, il n'y a pas de situation de famine. Nous démontrons enfin que notre protocole est instantanément stabilisant, c'est-à-dire qu'en

[3] Les preuves formelles sont disponibles dans `http ://arxiv4.library.cornell.edu/abs/1006.3432`

- **Génération de messages (pour tout processeur)**
  R1 : : $Demande_p \wedge Suivant_p(d) = q \wedge [OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)] \wedge NO\text{-}PIR_p \rightarrow OUT_p(q) := (m, d, choisir(c)), Demande_p := FAUX.$

- **Consommation de messages (pour tout processeur)**
  R2 : : $\exists q \in N_p, \exists m \in M; Consommation_p(q, m) \rightarrow Livré_p(m), IN_p(q) := OUT_q(p).$

- **Transmission interne (pour les processeurs ayant deux voisins)**
  R3 : : $\exists q \in N_p, \exists m \in M, \exists d \in I; Inter\text{-}trans_p(q, m, d) \wedge (NO\text{-}PIR_p \vee PIR\text{-}Synchro_p(q)) \rightarrow OUT_p(q') := (m, d, choisir(c)), IN_p(q) := OUT_q(p).$

- **Transmission de message de q vers p (pour les processeurs ayant deux voisins)**
  R4 : : $IN_p(q) = \epsilon \wedge OUT_q(p) \neq \epsilon \wedge (NO\text{-}PIR_p \vee PIR\text{-}Synchro_p(q)) \rightarrow IN_p(q) := OUT_q(p).$

- **Suppression de message après sa transmission (pour les processeurs ayant deux voisins)**
  R5 : : $\exists q \in N_p, OUT_p(q) = IN_q(p) \wedge (\forall q' \in N_p \setminus \{q\}, IN_p(q') = \epsilon) \wedge (NO\text{-}PIR_p \vee PIR\text{-}Synchro_p(q)) \rightarrow OUT_p(q) := \epsilon, IN_p(q') := OUT_{q'}(p).$

- **Suppression de message après sa transmission (pour les processeurs qui sont à l'extrémité)**
  R5' : : $N_p = \{q\} \wedge OUT_p(q) = IN_q(p) \wedge IN_p(q) = \epsilon \wedge ((p = p_0) \Rightarrow (EXT_p = \epsilon)) \wedge (NO\text{-}PIR_p \vee PIR\text{-}Synchro_p(q)) \rightarrow OUT_p(q) := \epsilon, IN_p(q) := OUT_q(p).$

- **Redirection des messages (Pour les processeurs qui sont à l'extrémité)**
  - **R6** : : $Change\text{-}Route_p(m) \wedge [OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)] \rightarrow OUT_p(q) := (m, d, choisir(c)), IN_p(q) := OUT_q(p).$
  - **R7** : : $Change\text{-}Route_p(m) \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p) \wedge PIR\text{-}Demande_p = FAUX \rightarrow PIR\text{-}Demande_p := VRAI.$
  - **R8** : : $Change\text{-}Route_p(m) \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p) \wedge PIR\text{-}Demande_p \wedge P\text{-}initiateur \rightarrow EXT_p := IN_p(q), IN_p(q) := OUT_q(p).$
  - **R9** : : $p = p_0 \wedge EXT_p \neq \epsilon \wedge [OUT_p(q) = \epsilon \vee OUT_p(q) = IN_q(p)] \wedge N\text{-}Initiateur \rightarrow OUT_p(q) := EXT_p, EXT_p := \epsilon.$
  - **R10** : : $p = p_0 \wedge EXT_p \neq \epsilon \wedge OUT_p(q) \neq \epsilon \wedge OUT_p(q) \neq IN_q(p) \wedge N\text{-}Initiateur \rightarrow EXT_p := \epsilon.$
  - **R11** : : $|N_p| = 1 \wedge p \neq 0 \wedge IN_p(q) = (m, d, c) \wedge d \neq p \wedge OUT_p(q) = \epsilon \wedge OUT_q(p) \neq IN_p(q) \rightarrow OUT_p(q) := (m, d, choisir(c)), IN_p(q) := OUT_q(p).$

- **Correction (pour p_0)**
- **R12** : : $p = p_0 \wedge EXT_p \neq \epsilon \wedge S_p \neq (P, -1) \rightarrow EXT_p = \epsilon.$
- **R13** : : $p = p_0 \wedge S_p = (P, ?) \wedge PIR\text{-}Demande = VRAI \rightarrow PIR\text{-}Demande = FAUX.$
- **R14** : : $p = p_0 \wedge S_p = (N, ?) \wedge PIR\text{-}Demande = VRAI \wedge [(IN_p(q) = (m, d, c) \wedge d = p) \vee IN_p(q) = \epsilon] \rightarrow PIR\text{-}Demande = FAUX.$

Algorithme 2: Acheminement de messages

partant d'une configuration initiale arbitraire, même si les tables de routage ne sont pas stables, tout message généré par un processeur est délivré à sa destination en un temps fini.

4. Dynamicité

Dans les environnements dynamiques, les processeurs peuvent rejoindre ou quitter le réseau à tout moment. Pour maintenir la propriété de stabilisation instantanée de notre protocole, nous supposons qu'aucun crash n'est possible. Lorsqu'un processeur désire quitter le réseau, il doit auparavant libérer ses tampons, c'est-à-dire envoyer les messages qui sont dans ses tampons et attendre leur réception tout en refusant la réception de nouveaux messages.
Dans cette discussion, nous supposons que le réseau reconstruit conserve une topologie en chaîne. Notons la contradiction qu'il y a entre (i) le fait de garder des messages ayant des destinations fantômes dans l'espoir de les voir se reconnecter au réseau et (ii) le fait d'éviter des situation de congestions (charge supérieure aux capacités des ressources). Si aucune borne n'est connue sur le nombre de connexions et de déconnexions, ce problème n'admet pas de solution. Le seul moyen consiste donc à redéfinir le problème dans un contexte dynamique. Nous pouvons notamment modifier le second point de la spécification comme suit : *un message valide* m *généré par un processeur* p *pour la destination* q *est livré à* q *en un temps fini si* m, p *et* q *sont continuellement dans la même composante connexe durant l'acheminement du message* m.
Ceci peut sembler être une hypothèse forte mais dans la pratique ce genre d'hypothèses est souvent implicite. Remarquons que cette nouvelle spécification est similaire à la spécification SP si nous considérons un environnement statique. Notre algorithme est donc facilement adaptable pour être instantanément stabilisant pour cette nouvelle spécification pour des chaînes dynamiques. Nous pouvons maintenant supprimer des messages de la manière suivante : nous supposons que chaque message a un

champ supplémentaire qui est de type booléen initialement placé à FAUX à la génération du message. Lorsqu'un message arrive à une extrémité de la chaine qui n'est pas sa destination, deux cas sont possibles selon la valeur du champ additionnel (i) elle est égale à FAUX ou (ii) elle est égale à VRAI. Dans le premier cas, le processeur la modifie en mettant le booléen à VRAI et renvoie le message dans l'autre direction. Dans le second cas, le processeur peut supprimer le message car il n'est pas valide. En effet, dans le cas contraire, il aurait rencontré sa destination en parcourant toute la chaîne.
Finalement, afin d'éviter des situations de famine, la vitesse de connexion et de déconnexion des processeurs doit être suffisamment lente pour éviter des séquences d'exécution du PIR qui pouraient empêcher des processeurs de générer des messages.

5. Conclusion

Nous avons présenté le premier protocole instantanément stabilisant pour l'acheminement de messages qui utilise un nombre de tampons qui est indépendant d'un paramètre global du système. Notre protocole fonctionne sur une topologie en chaîne qui utilise quatre tampons pour chaque lien de communication. Il tolère la dynamicité du réseau à condition que la topologie reste une chaîne. Ce travail constitue un premier pas vers l'obtention de résultats similaires sur des topologies plus générales. Plus particulièrement, en combinant un algorithme d'acheminement de messages instantanément stabilisant avec n'importe quel algorithme d'overlay auto-stabilisant, par exemple [4] pour une DHT ou [2, 5, 6] pour une structure d'*arbre*, nous pourrions obtenir des solutions garantissant à l'utilisateur d'avoir les bonnes réponses en interrogeant l'overlay.

Bibliographie

1. F. Petit A. Bui, A-K. Datta and V. Villain. Snap-stabilization and PIF in tree networks. *Distributed Computing*, 20(1) :3–19, 2007.
2. J. Aspnes and G. Shah. Skip Graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, January 2003.
3. B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilizing end-to-end communication. *Journal of High Speed Networks*, 5(4) :365–381, 1996.
4. M. Bertier, F. Bonnet, A-M. Kermarrec, V. Leroy, S. Peri, and M. Raynal. D2HT : the best of both worlds, Integrating RPS and DHT. In *European Dependable Computing Conference*, pages 135–144, 2010.
5. E. Caron, A-K. Datta, F. Petit, and C. Tedeschi. Self-stabilization in tree-structured P2P service discovery systems. In *27th International Symposium on Reliable Distributed Systems (SRDS 2008)*, pages 207–216. IEEE, 2008.
6. E. Caron, F. Desprez, F. Petit, and C. Tedeschi. Snap-stabilizing Prefix Tree for Peer-to-Peer Systems. In *SSS 2007*, pages 82–96. Springer Verlag Berlin Heidelberg, 2007.
7. A. Cournier, S. Dubois, and V. Villain. A snap-stabilizing point-to-point communication protocol in message-switched networks. In *23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009)*, pages 1–11, 2009.
8. A. Cournier, S. Dubois, and V. Villain. How to improve snap-stabilizing point-to-point communication space complexity ? *Theoretical Computer Science*, In Press, 2010.
9. E-W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commum. ACM*, 17(11) :643–644, 1974.
10. S-T. Huang and N-S. Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Inf. Process. Lett.*, 41(2) :109–117, 1992.
11. C. Johnen and S. Tixeuil. Route preserving stabilization. In *Self-Stabilizing Systems, 6th International Symposium, SSS 2003, Proceedings*, volume 2704 of *Lecture Notes in Computer Science*, pages 184–198, 2003.
12. A. Kosowski and L. Kuszner. A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves. In *Parallel Processing and Applied Mathematics, 6th International Conference, PPAM, Proceedings*, volume 3911 of *Lecture Notes in Computer Science*, pages 75–82, 2005.
13. E. Kushilevitz, R. Ostrovsky, and A. Rosén. Log-space polynomial end-to-end communication. In *STOC '95 : Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 559–568. ACM, 1995.
14. P-M. Merlin and P-J. Schweitzer. Deadlock avoidance in store-and-forward networks. In *Jerusalem Conference on Information Technology*, pages 577–581, 1978.