MajecSTIC 2010
Bordeaux, France, du 13 au 15 octobre 2010

Gestion auto-stabilisante avec garantie de service de la connaissance des clusters voisins

Colette Johnen[1] et Fouzi Mekhaldi[2]

1 : Université de Bordeaux, LaBRI, CNRS, F-33405 Talence - France.
2 : Université Paris-Sud, LRI, CNRS, F-91405 Orsay - France.

Contact : mekhaldi@lri.fr

Résumé
Le partitionnement en clusters est proposé dans les réseaux mobiles sans infrastructure pour améliorer leurs performances. Comme les protocoles de partitionnement sont adaptatifs aux changements topologiques, la structure hiérarchique produite sera dynamique : des clusters peuvent apparaître et disparaître au cours du temps. Par conséquent, tous les protocoles hiérarchiques doivent être également adaptatifs à ces changements.
Dans cet article, nous proposons un protocole de connaissance de voisinage des clusters, baptisé CNK. CNK permet à chaque leader de cluster de connaître tous ses clusters voisins : l'identité de leurs leaders, les chemins menant à eux, ainsi que la liste de leurs membres. En plus, CNK est auto-stabilisant, et réagit efficacement aux changements de la structure hiérarchique. Un service minimum sera garanti, et maintenu même suite à l'occurrence des changements de la structure hiérarchique. Le service minimum est définit comme suit : *un leader connaît l'identité de tous les leaders des clusters voisins, ainsi que des chemins menant à eux*. Un tel service assure la continuité de fonctionnement des protocoles de couches supérieures, comme le routage hiérarchique.

Mots-clés : auto-stabilisation, garantie de service, partitionnement, voisinage d'un cluster.

---

1. Introduction

L'auto-stabilisation est l'une des approches de conception d'algorithmes distribués tolérants aux pannes. Elle assure que le système, indépendamment de sa configuration initiale, retrouve en un temps fini (appelé temps de stabilisation) une configuration légitime répondant à ses spécifications. Les algorithmes auto-stabilisants sont attractifs vu qu'ils ne nécessitent aucune initialisation globale, ni d'intervention extérieure suite à des perturbations. Cependant, durant toute la période de convergence vers une configuration légitime, les algorithmes auto-stabilisants ne garantissent aucune propriété. Si des perturbations empêchant d'atteindre une configuration légitime se produisent fréquemment, la disponibilité, la fiabilité, et l'intégrité du système seront fortement compromises. De ce fait, les algorithmes auto-stabilisants ne sont pas totalement appropriés aux réseaux à grand échelle, où intrinsèquement, les perturbations sont fréquentes.

Dans la dernière décennie, de nouvelles approches visant à étendre l'auto-stabilisation, ont été développées. Nous nous intéressons aux approches garantissant des propriétés durant la période de convergence, à savoir : la super-stabilisation [3], la convergence sûre [7, 8], et l'auto-stabilisation robuste [4–6]. La super-stabilisation assure que le système préserve une propriété de sûreté suite à une faute subie dans une configuration légitime. Donc, cette approche s'intéresse simplement aux fautes qui se produisent après la phase de convergence, et non pas aux fautes qui surviennent durant la convergence. La convergence sûre, et l'auto-stabilisation robuste garantissent d'atteindre *très rapidement* une configuration sûre où une propriété de sûreté est assurée. A partir d'une telle configuration, pendant la convergence vers une configuration légitime, aucune action de l'algorithme ne falsifie la propriété de sûreté. Ainsi, un service minimum est assuré. De plus, l'auto-stabilisation robuste maintient le service minimal suite à l'occurrence de certains événements.

Nous considérons une nouvelle approche appelée auto-stabilisation avec garantie de service. Un protocole est auto-stabilisant avec garantie de service s'il garantit de fournir un service minimum tel que (1) il le préserve suite à l'occurrence de certains événements, appelés admissibles, et (2) il le préserve pendant l'évolution du protocole pour offrir un service optimum, c-à-d pendant la convergence vers une configuration légitime. Le but principale de cette approche n'est pas la rapidité de convergence vers un service minimal, mais la préservation de ce service sous les actions du protocole ainsi que les événements admissibles. Dans cette approche, les événements admissibles doivent être connus, et pris en considération lors de la conception du protocole. Ainsi, l'occurrence des fautes inconnues est gérée par l'auto-stabilisation. Alors que, les événements admissibles sont gérés de tel sorte que le service minimum soit offert. Notez que le délai minimum entre deux occurrences d'événements admissibles peut être infime. Néanmoins, le service minimum reste toujours assuré.

Motivation. Le partitionnement en clusters est proposé, dans les réseaux mobiles sans infrastructure, pour améliorer les performances de certains protocoles. Chaque cluster est géré par un leader qui joue le rôle d'un coordinateur local. Ce partitionnement crée un premier niveau hiérarchique (niveau 1) sur une topologie plate (niveau 0). Une architecture à multi-niveaux peut être créée progressivement : les clusters du niveau i sont considérés comme les nœuds du niveau i+1, et le partitionnement est réitéré. Par conséquent, la création d'un niveau $i+1$ nécessite la connaissance du voisinage au niveau i. Plus précisément, chaque cluster doit connaître ses clusters voisins.

Les protocoles de partitionnement doivent être auto-adaptatifs aux changements topologiques du niveau physique (concernant les nœuds et les liens de communication). De ce fait, la structure produite sera dynamique, puisque des clusters peuvent apparaître et disparaître, et la composition des clusters peut changer au cours du temps. Tous les protocoles hiérarchiques doivent être également adaptatifs aux changements de l'organisation hiérarchique. De plus, la connaissance du voisinage d'un cluster est supposée existante par de nombreux protocoles de routage hiérarchique et partitionnement en multi-niveaux, comme [1]. Toutefois, il n'existe pas des protocoles calculant la connaissance du voisinage des clusters d'une manière distribuée auto-stabilisante, et préservant cette connaissance avec une garantie de service.

Contribution. Nous proposons le premier protocole auto-stabilisant avec garantie de service, baptisé CNK, calculant et maintenant efficacement la connaissance des clusters voisins dans un réseau partitionné. Le protocole CNK offre une interface (ensemble de données et de fonctions) aux couches supérieures en présentant une vue succincte du réseau. Chaque cluster connaît tous ses clusters voisins, et des chemins permettant de les atteindre. Plus précisément, chaque leader connaît les chemins menant aux leaders des clusters voisins, ainsi que la liste de leurs membres.
Le protocole CNK réagit de façon efficace aux changements de l'organisation hiérarchique tels que : (i) la création de nouveaux clusters, (ii) la destruction de clusters ou (iii) le changement de composition des clusters. Ces changements ne sont pas des anomalies ou des fautes mais des événements normaux dans un réseau dynamique. De plus, le délai minimal entre deux perturbations consécutives peut être infime (il dépend du protocole de partitionnement et du dynamisme du réseau). Les événements admissibles sont les changements de l'organisation hiérarchique cités précédemment. Le protocole CNK gère ces changements de façon transparente, de tel sorte qu'un service utile (minimal) reste assuré après l'occurrence de ces changements. Le service minimal est : *chaque leader connaît l'identité de tous les leaders des clusters voisins, et les chemins menant à eux*.
La garantie de service assure que suite à l'occurrence des changements cités précédemment, le service minimum reste assuré, ce qui permet la continuité du fonctionnement des protocoles hiérarchiques (tel que le routage, diffusion, ...).
Dans une organisation hiérarchique, les messages sont acheminés d'un cluster à un cluster voisin jusqu'au cluster destinataire (contenant le nœud destinataire). Cette tâche pourrait être réalisé efficacement en se basant sur le protocole CNK. Vu qu'à tout instant, chaque cluster connaît tous ses clusters voisins, et les chemins menant vers leurs leaders.

2. Modèle

Dans cet article, nous considérons un réseau dynamique. Le réseau est modélisé par un graphe connexe $G = (V, E)$. V est l'ensemble des sommets (ou nœuds), et E est l'ensemble des arêtes (ou

liens de communication). Un lien existe entre deux nœuds u et v si $(u, v) \in E$. Dans ce cas, u et v sont voisins. On note N_v l'ensemble des voisins d'un nœud v.
Nous prenons comme modèle de calcul le modèle à « *états* » : les variables d'un nœud définissent son *état*. L'ensemble des états des nœuds à un instant donné forme la *configuration* du système. Les variables des nœuds sont partagées : chaque nœud peut lire les variables de ses voisins, mais il ne peut modifier que les siennes. L'algorithme de chaque nœud v est un ensemble de *règles*, de la forme $R_i(v) : Garde_i \rightarrow Action_i$. La garde d'une règle $R_i(v)$ est exprimée à l'aide des variables locales de v, ainsi que celles de ses voisins. L'action d'une règle $R_i(v)$, modifie une ou plusieurs variables du nœud v. Une règle est exécutable si elle est activable, c-à-d si sa garde est satisfaite. On dit qu'un nœud est activable, s'il a au moins une règle activable. Un pas de calcul consiste en un ou plusieurs nœuds qui exécutent une règle. Une exécution est une succession de pas de calcul. On dit qu'une exécution est *équitable*, si pour tout nœud qui est infiniment souvent activable le long de cette exécution, alors il exécutera son action en un temps fini.
Nous utilisons la notion de *cycle asynchrone* pour mesurer la complexité en temps. Un cycle asynchrone d'une exécution e commençant à partir d'une configuration c, est le suffixe minimal de e pendant lequel tout nœud activable en c exécute une règle.

3. Problème de connaissance des clusters voisins

Dans cet article, nous nous plaçons dans une architecture hiérarchique basée sur des clusters à 1-saut. Dans cette architecture, le leader d'un cluster est à distance 1 des autres membres du cluster. On dit que deux clusters C_1, C_2 sont voisins s'ils possèdent deux nœuds x et y qui soient voisins (c-à-d, $\exists x \in C_1 \wedge \exists y \in C_2 \wedge x \in N_y$). Notons alors que deux leaders u et v de deux clusters voisins sont au plus à distance 3 entre eux. De plus, les chemins entre v et u ne comportent aucune passerelle qui soit leader. Dans la Figure 1, les clusters de CH1 et CH4 ne sont pas voisins bien que CH4 soit à distance 3 de CH1 : CH4 et CH1 ne sont pas voisins dans le graphe induit (Figure 2) par le partitionnement présenté dans la Figure 1.

Notre objectif est que chaque cluster connaisse tous ses clusters voisins. Donc, chaque leader v doit connaître les clusters ayant des leaders u à distance 3 au plus de v. Tel que, les chemins entre v et u ne comportent aucune passerelle qui soit leader.

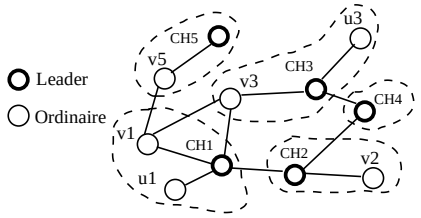

FIG. 1 – Exemple d'un réseau partitionné.

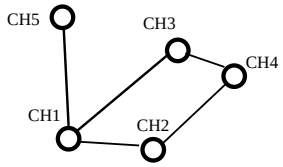

FIG. 2 – Abstraction d'un réseau partitionné.

Notons aussi que l'apparition ou la résignation d'un leader peut provoquer l'apparition de nouveaux clusters voisins. Dans la Figure 1, si $v5$ devient leader, et CH3 devient non leader, alors CH1 gagnera 2 clusters voisins : ceux de $v5$, et de CH4.

Définition 1 ***Le 3R-voisinage*** *d'un nœud v est l'ensemble des nœuds à distance 3 au plus de v, et accessible depuis v par un chemin ne comportant aucun leader.*

Spécifications du problème : la connaissance du voisinage doit vérifier les propriétés suivantes :
• **Complétude :** Chaque leader connaît tous les leaders de son 3R-voisinage, ainsi que les chemins menant à eux.
• **Exactitude :** les renseignements stockés par chaque leader v sont tous exactes : (1) tous les chemins connus par v sont valides. (2) v connaît la composition exacte de tous les clusters voisins.

Définition 2 ***(configuration légitime) :*** *Dans une configuration légitime, les propriétés de complétude et d'exactitude sont satisfaites.*

L'intérêt de l'auto-stabilisation avec garantie de service dépend principalement de deux facteurs.

1. La définition du service minimum : ce service est la propriété de complétude.
2. La liste des événements admissibles : toutes les actions du protocole de partitionnement.

4. Interaction entre le protocole CNK et le protocole de partitionnement

Bien que le protocole CNK suppose l'existence d'un protocole auto-stabilisant de partitionnement en clusters à 1-saut, CNK ne dépend pas des spécifications d'un tel protocole. Néanmoins, il nécessite une coopération de ce dernier, pour assurer la guarantie de service.
Le protocole de partitionnement indique à chaque nœud v son statut hiérarchique via la variable $Status_v$ (mise à jour seulement par ce protocole). Les statuts hiérarchiques habituels d'un nœud v sont : Cluster-head ($Status_v = CH$) qui joue le rôle d'un leader, et ordinaire ($Status_v = O$). Cependant, pour maintenir la connaissance des clusters voisins, une interaction entre le protocole de partitionnement et CNK est mise en place. Cette interaction est illustrée dans la Figure 4.
En effet, deux statuts hiérarchique intermédiaires sont introduits, à savoir : ***Presque-ordinaire*** *(*$Status_v = NO$*)*, et ***Presque-cluster-head*** *(*$Status_v = NCH$*)*.
Un Cluster-head v voulant devenir ordinaire, prend le statut presque-ordinaire. Alors qu'un nœud ordinaire u voulant devenir Cluster-head, prend le statut presque-Cluster-head.

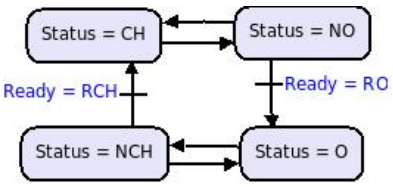

FIG. 3 – Interaction avec le protocole de partitionnement.

En passant vers l'un des statuts intermédiaires, le protocole de partitionnement envoie une requête au protocole CNK, et se met en attente d'une autorisation pour changer de statut. L'autorisation sera communiqué via la variable Ready (une variable du protocole CNK). La valeur RO de la variable $Ready_v$ (d'un nœud v), indique que v est prêt à être ordinaire. Alors que, la valeur RCH, indique que v est prêt à être cluster-head.
La variable Ready est mise à jour uniquement par la protocole CNK. Pour tout nœud ordinaire, la valeur par défaut de Ready est RO, alors que pour les cluster-heads est RCH.
Un nœud v presque-cluster-head peut devenir cluster-head seulement si $Ready_v = RCH$, mais il peut revenir au statut ordinaire à tout moment (même si $Ready_v = RCH$). De même, un nœud u presque-ordinaire peut devenir ordinaire seulement si $Ready_v = RO$, mais il peut revenir au statut cluster-head à tout moment (même si $Ready_v = RO$).

Important. Il est à noter que la notion d'un nœud leader utilisée pour définir les spécifications du problème (section 3), désigne un nœud qui est soit cluster-head, soit presque-ordinaire.

5. Le protocole CNK : Calcul auto-stabilisant des tables de voisinage

De manière décentralisée, chaque nœud v maintient une Table de Voisinage KT_v, dont la structure est illustré dans la Figure 4. Algorithme 1 décrit les variables et les macros utilisées par chaque nœud v, alors que les règles de calcul de la table sont décrites dans Algorithme 2.

Dans le protocole CNK, chaque nœud v maintient une variable HS_v indiquant son statut hiérarchique. HS_v est mise à jour à $Status_v$ via la règle R0 (décrite ultérieurement dans Algorithme 2). Notez que l'utilisation de la variable HS_v est nécessaire dû à l'impossibilité de partager la variable $Status_v$ vu qu'elle est une variable du protocole de partitionnement.
La table de voisinage KT_v est mise à jour par quatre type de règles, $Ri(v), i \in [0-3]$. Chaque règle $Ri(v)(i > 0)$ permet au nœud v de récupérer des informations concernant des leaders à distance i, connaissant uniquement les tables de ses voisins. De manière générale, chaque règle Ri est composée de trois sous-règles : insertion d'un nouveau enregistrement (Ri_1), mise à jour d'un enregistrement (Ri_2), et suppression d'enregistrements (Ri_3).

Nom	id	G1	G2	Liste	Hs	pif
Type	ID	ID or $\perp$	ID or $\perp$	$\{IDs\}$	{CH, NO, NCH, O}	$\{B, C, F\}$

FIG. 4 – Structure de la table de voisinage.

Chaque enregistrement de cette table correspond à un cluster voisin, et un chemin menant vers son leader. Les champs *id*, *G1*, *G2*, *Hs* et *Liste* correspondent respectivement à l'identité d'un leader u, la $1^{ère}$ et $2^{ème}$ passerelle sur le chemin de v vers u, le statut de u et la liste des membres de son cluster. L'utilité du champ pif, sera discuté par la suite (voir Section 6).

En l'absence d'une première (resp. deuxième) passerelle dans un chemin, le champs G1 (resp. G2) aura la valeur $\perp$. Par exemple, à partir du réseau illustré par Figure 1, quand les tables seront calculées, l'enregistrement $(CH3, v3, \perp)$ appartient à KT_{CH1}, et $(CH1, v3, \perp)$ appartient à KT_{CH3}.

Algorithme 1 : : Variables et macros du nœud v.

Variables d'entrée (depuis l'algorithme de partitionnement)
  $Status_v \in \{CH, O, NO, NCH\}$; Le statut de v.
  $Head_v \in \{IDs\}$; l'identité du leader de v.
Variables de sortie (vers l'algorithme de partitionnement)
  $Ready_v \in \{RO, RCH\}$; (définie dans la section 4)
Variables partagées
  $HS_v \in \{CH, O, NO, NCH\}$; Le statut de v, mis à jour selon $Status_v$.
  KT_v : La table de voisinage, dont la structure est décrite par Figure 4.
Macros
$Cluster_v$ :: **Si** $HS_v \in \{CH, NO\}$ **alors** $\{z \in N_v : Head_z = v\} \cup \{v\}$;
  Si $HS_v = NCH$ **alors** $\{v\}$;
$Insert(dest, g1, g2, List, status)$, ajoute un enregistrement à KT_v où le champ pif est mis à C.
$Delete(dest, g1, g2)$, supprime l'enregistrement de KT_v vérifiant $id = dest$, $G1 = g1$, $G2 = g2$.
$UpdatePIF(x, y, z, t)$, met à jour le champs pif à t, dans l'enregistrement de KT_v vérifiant $id = x$, $g1 = y$, $g2 = z$
$UpdateReady$ :: **Si** $HS_v = CH$ **alors** $Ready_v := RCH$;
  Si $HS_v = O$ **alors** $Ready_v := RO$;

6. Le protocole CNK : Mécanisme de garantie de service

La garantie de service assure que l'autorisation attribuée à une requête de changement de statut, à savoir la mise à jour de la variable $Ready$ (voir Section 4), ne falsifie pas la propriété de complétude. La mise à jour de $Ready$ constitue alors un point cruciale dans le protocole CNK.

6.1. Comment mettre à jour la variable $Ready$?

Suite à une configuration initiale incorrecte, un nœud v peut être amené à corriger la valeur de sa variable $Ready_v$. Si le nœud v est ordinaire et $Ready_v$ est à RCH, v corrige la valeur de $Ready_v$ à RO via la règle $RC_O(v)$. De même, si v est $cluster - head$ et $Ready_v$ est à RO, alors v corrige la valeur de $Ready_v$ à RCH via la règle $RC_{CH}(v)$. Ces deux règles sont décrites dans Algorithme 3.

La mise à jour de $Ready$ par un nœud $presque-cluster-head$ ou $presque-ordinaire$ nécessite un traitement particulier. Etudions ces deux cas un par un.

Cas 1. Selon la spécification de la propriété de complétude, un nœud presque-cluster-head v (n'étant pas encore leader) n'a pas besoin de connaître les leaders de son 3R-voisinage, ni d'être connu par eux. Cependant, dés que v met à jour $Ready_v$ à RCH, il peut devenir leader par une action du protocole de partitionnement. En tant que leader, v doit connaître les leaders de son 3R-voisinage, et doit être connu par eux. Autrement, la propriété de complétude sera compromise. D'où, avant que v ne mette à jour $Ready_v$, il doit connaître les chemins vers les leaders de son 3R-voisinage. En plus, ces leaders doivent connaître les chemins inverses (vers v). Cette connaissance supplémentaire peut être établie par une propagation d'information suivie d'un renvoi d'acquittement (PIF : Propagation of Information with Feedback). A la fin de la phase de propagation d'information, les leaders connaissent un chemin vers v. Les acquittements permettent à v de connaître les chemins inverses, et ensuite mettre à jour la variable $Ready$.

Cas 2. Il est simple aussi de montrer qu'un nœud presque-ordinaire ne doit mettre à jour sa variable $Ready$ à RO qu'après avoir procédé à une propagation d'information avec acquittement (PIF). Prenons l'exemple illustré par Figure 1. Tant que CH4 est leader, il doit connaître des chemins vers CH2 et CH3. Néanmoins, CH2 n'as pas besoin de connaître un chemin vers CH3, et vice versa. A l'instant où CH4 ne devient plus leader, CH2 et CH3 doivent se connaître mutuellement, sinon la propriété de complétude sera compromise. CH4 doit assurer que CH2 et CH3 se connaissent par un chemin passant par lui, avant qu'il mette $Ready$ à RO. En effet, CH4 doit diffuser ses connaissances (les chemins vers CH3 et CH2). A la réception de cette information, CH3 (resp. CH2) inscrit dans sa table KT un chemin vers CH2 (resp. CH3) en considérant CH4

Algorithme 2 : : Les règles du nœud v.

$\mathbf{R0_1}(v) :: (Status_v \neq O) \wedge (v, \perp, \perp) \notin KT_v \longrightarrow HS_v := Status_v; UpdateReady; Insert(v, \perp, \perp, Cluster_v, HS_v);$
$\mathbf{R0_2}(v) :: (Status_v \neq O) \wedge (v, \perp, \perp) \in KT_v \wedge (HS_v \neq Status_v)$
$\longrightarrow HS_v := Status_v; UpdateReady; UpdatePIF(v, \perp, \perp, C); Update(v, \perp, \perp, Cluster_v, HS_v);$
$\mathbf{R0_3}(v) :: (Status_v = O) \wedge \Big((HS_v \neq Status_v) \vee (v, \perp, \perp) \in KT_v\Big)$
$\longrightarrow HS_v := Status_v; UpdateReady; Delete(v, \perp, \perp);$
$\mathbf{R0_4}(v) :: (Status_v \neq O) \wedge \exists (v, \perp, \perp, List, hs) \in KT_v \wedge (HS_v = Status_v) \wedge [(List \neq Cluster_v) \vee (hs \neq HS_v)]$
$\longrightarrow Update(v, \perp, \perp, Cluster_v, HS_v);$

$\mathbf{R1_1}(v) : \exists u \in N_v \wedge \exists (u, \perp, \perp, List, Hs) \in KT_u \wedge (u, \perp, \perp) \notin KT_v \longrightarrow Insert(u, \perp, \perp, List, Hs);$
$\mathbf{R1_2}(v) : \exists (u, \perp, \perp, List', Hs') \in KT_v \wedge (u \in N_v) \wedge \exists (u, \perp, \perp, List, Hs) \in KT_u \wedge [(List' \neq List) \vee (Hs' \neq Hs)]$
$\longrightarrow Update(u, \perp, \perp, List, Hs);$
$\mathbf{R1_3}(v) : \exists (u, \perp, \perp) \in KT_v \wedge [(u \notin N_v) \vee (u, \perp, \perp) \notin KT_u] \longrightarrow Delete(u, \perp, \perp);$

$\mathbf{R2_1}(v) : \exists z \in N_v \wedge (HS_z \neq CH) \wedge \exists (u, \perp, \perp, List, Hs) \in KT_z \wedge (u \neq z) \wedge (u \neq v) \wedge (u, z, \perp) \notin KT_v$
$\longrightarrow Insert(u, z, \perp, List, Hs);$
$\mathbf{R2_2}(v) : \exists (u, z, \perp, List', Hs') \in KT_v \wedge (z \in N_v) \wedge \exists (u, \perp, \perp, List, Hs) \in KT_z \wedge [(List' \neq List) \vee (Hs' \neq Hs)]$
$\longrightarrow Update(u, z, \perp, List, Hs);$
$\mathbf{R2_3}(v) : \exists (u, z, \perp) \in KT_v \wedge (z \neq \perp) \wedge [(z \notin N_v) \vee (HS_z = CH) \vee (u, \perp, \perp) \notin KT_z] \longrightarrow Delete(u, z, \perp);$

$\mathbf{R3_1}(v) : (HS_v \neq O) \wedge \exists w \in N_v \wedge (HS_w \neq CH) \wedge \exists (u, z, \perp, List, Hs) \in KT_w \wedge (u \neq v) \wedge (z \neq v) \wedge$
$(z \neq \perp) \wedge ((u, w, z) \notin KT_v) \longrightarrow Insert(u, w, z, List, Hs);$
$\mathbf{R3_2}(v) : (HS_v \neq O) \wedge \exists (u, w, z, List', Hs') \in KT_v \wedge (w \in N_v) \wedge (HS_w \neq CH) \wedge \exists (u, z, \perp, List, Hs) \in KT_w \wedge$
$[(List' \neq List) \vee (Hs' \neq Hs)] \longrightarrow Update(u, w, z, List, Hs);$
$\mathbf{R3_3}(v) : (HS_v \neq O) \wedge \exists (u, w, z) \in KT_v \wedge [(w \notin N_v) \vee (HS_w = CH) \vee (u, z, \perp) \notin KT_w] \longrightarrow Delete(u, w, z);$
$\mathbf{R3_4}(v) : (HS_v = O) \wedge \exists (u, w, z) \in KT_v \wedge (w \neq \perp) \wedge (z \neq \perp) \longrightarrow Delete(u, w, z);$

comme passerelle, puis il retourne un acquittement. Quand CH4 reçoit les acquittements, il met à jour Ready.

Nous avons adapté un algorithme de PIF, nommé PFC présenté dans [2], s'exécutant sur un arbre orienté pour mettre en œuvre le mécanisme de garantie de service. Dans cet algorithme, chaque nœud v possède une variable S_v indiquant si v est en phase de : diffusion ($S_v = B$), acquittement ($S_v = F$), ou bien en phase de nettoyage ($S_v = C$). Sous l'algorithme PFC, une diffusion avec acquittement nécessite $2h + 1$ cycles de calculs, où h est la hauteur de l'arbre.

6.2. Adaptation de l'algorthime PFC

L'idée de notre solution consiste à considérer l'ensemble des tables de voisinage comme une forêt d'arbres. Ainsi, chaque enregistrement sera vue comme un nœud d'un seul arbre. La racine d'un arbre sera l'enregistrement associé au nœud initiateur du PIF. Par exemple, si v lance un PIF, alors $(v, \perp, \perp)$ de KT_v sera la racine de l'arbre. Le champs pif d'un enregistrement indiquera alors l'état de cet enregistrement vis-à-vis du PIF courant. Comme tout PIF est effectué dans le 3R-voisinage du nœud initiateur, la longueur d'un arbre est au maximum 3 ($h \leq 3$).

Définition 3 (Enregistrements d'un arbre de PIF)

- *Le parent dans l'arbre de l'enregistrement* $(u, \perp, \perp)$ *de* KT_z *est la racine* $(u, \perp, \perp)$ *de* KT_u.
- *Le parent de* $(u, z, \perp)$ *de* KT_w *est l'enregistrement* $(u, \perp, \perp)$ *de* KT_z.
- *Le parent de* (u, w, z) *de* KT_v *est l'enregistrement* $(u, z, \perp)$ *de* KT_w.

Notez que, la racine d'un enregistrement (u, w, z) de KT_v est l'enregistrement $(u, \perp, \perp)$ de KT_u.

Définition 4 (Feuilles d'un arbre de PIF)

L'enregistrement $(u, \perp, \perp)$ *de* KT_z *est feuille si* z *est cluster-head, ou il n'as pas de descendants.*
L'enregistrement $(u, z, \perp)$ *de* KT_w *est feuille si* w *est cluster-head, ou tous ses descendants sont ordinaires.*
Un enregistrement (u, z, w) *de* KT_v *est toujours feuille.*

L'Algorithme 3 présente les règles permettant à un nœud v d'effectuer (ou participer à) un PIF. La structure générale de l'arbre où le PIF sera effectué, est illustré dans la Figure 6.2.
Chaque nœud ayant le statut presque-ordinaire ou presque-cluster-head lance une diffusion via la règle $RB(v)$. Les nœuds qui ne sont pas des feuilles et à distance 1 (resp. 2) de la racine, participent

à la diffusion via la règle IB-d1(v) (resp. IB-d2(v)). Quand les nœuds feuilles reçoivent la diffusion, ils lancent l'acquittement via la règle IF. Un nœud pourra propager son acquittement (vers la racine) lorsque tous ses descendants ont remis l'acquittement. Si le prédicat RF-guard est vérifié, tous les descendants de la racine ont remis l'acquittement. Dans ce cas, la racine peut terminer son PIF, par l'exécution d'une des deux règles $RR_{CH}(v)$ et $RR_O(v)$. Notez qu'en plus des règles de diffusion, et de remise d'acquittement, il existe des règles de correction : RC et IC. Ces règles mettent le champs pif d'un enregistrement à C. Une phase de nettoyage est ainsi lancée, soit pour marquer la fin d'un PIF, soit pour abandonner un PIF inutile ou incorrecte.

Les règles R0 (mettant à jour HS), et $Ri_1, i \in [1-3]$) sont plus prioritaires que les règles du PIF. Une règle de PIF n'est activable que si le prédicat Desactive est satisfait. Cette contrainte assure qu'un nœud participe à un PIF que s'il connaît tous les chemins connus par ses voisins. Ainsi, la connaissance de la racine est transmit pendant la diffusion. De plus, en fin de la phase d'acquittement, la racine récupérera les chemins vers les leaders de son 3R-voisinage en accédant seulement aux tables de ses voisins.

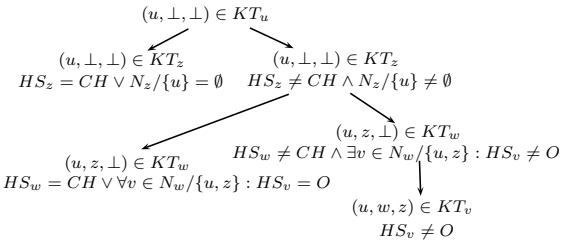

FIG. 5 – Structure de l'arbre d'un PIF.

Algorithme 3 : Algorithme du PIF au nœud v

Note : Gi_j est la garde de la règle Ri_j définie dans Algorithme 2.

$Desactive(v) :: \forall i \in [1\text{-}3], G0_i(v) = False \wedge Gi_1(v) = False \wedge Gi_2(v) = False$

/* Lancement de la diffusion par la racine */

RB$(v) :: \Big((HS_v = NCH \wedge Ready_v = RO) \vee (HS_v = NO \wedge Ready_v = RCH)\Big) \wedge (v,\perp,\perp,C) \in KT_v \wedge$
$\big(\forall x \in N_v, (v,\perp,\perp,C) \in KT_x\big) \wedge Desactive(v) = True \longrightarrow UpdatePIF(v,\perp,\perp,B);$

/* Terminaison du PIF, et mise à jour de Ready */

RF-Guard$(v) :: (v,\perp,\perp,B) \in KT_v \wedge (\forall x \in N_v, (v,\perp,\perp,F) \in KT_x) \wedge Desactive(v) = True$

RR$_O$ $(v) :: (HS_v = NO) \wedge RF\text{-}Guard(v) = True \longrightarrow Ready_v := RO;\ UpdatePIF(v,\perp,\perp,F);$

RR$_{CH}$ $(v) :: (HS_v = NCH) \wedge RF\text{-}Guard(v) = True \longrightarrow Ready_v := RCH;\ UpdatePIF(v,\perp,\perp,F);$

/* Correction de Ready par les ordinaires, et nœuds Cluster-head */

RC$_O$ $(v) : (HS_v = O) \wedge (HS_v = Status_v) \wedge (Ready_v = RCH) \longrightarrow Ready_v := RO;$

RC$_{CH}$ $(v) : (HS_v = CH) \wedge (HS_v = Status_v) \wedge (Ready_v = RO) \longrightarrow Ready_v := RCH;$

/* Participation à la diffusion par les nœuds à distance 1 et 2 de la racine */

IB-d1$(v) :: \exists (u,\perp,\perp,C) \in KT_v \wedge (u \in N_v) \wedge (u,\perp,\perp,pif) \in KT_u \wedge (pif \in \{B,F\}) \wedge$
$\Big((HS_v = CH) \vee (\forall x \in N_v/\{u\} : (u,v,\perp,C) \in KT_x)\Big) \wedge Desactive(v) = True \longrightarrow UpdatePIF(u,\perp,\perp,B);$

IB-d2$(v) :: \exists (u,z,\perp,C) \in KT_v \wedge (z \in N_v) \wedge (u,\perp,\perp,pif) \in KT_z \wedge (pif \in \{B,F\}) \wedge (HS_z \neq CH) \wedge$
$\Big((HS_v = CH) \vee (\forall x \in N_v/\{u,z\} : HS_x = O \vee (u,v,z,C) \in KT_x)\Big) \wedge Desactive(v) = True$
$\longrightarrow UpdatePIF(u,z,\perp,B);$

/* Renvoie d'acquittement par les nœuds à distance 3, 2 et 1 de la racine */

IF-d1$(v) :: \exists (u,\perp,\perp,B) \in KT_v \wedge u \in N_v \wedge (u,\perp,\perp,pif') \in KT_u \wedge pif' \in \{B,F\} \wedge \big((HS_v = CH) \vee$
$(\forall x \in N_v/\{u\} : (u,v,\perp,F) \in KT_x)\big) \wedge Desactive(v) = True \longrightarrow UpdatePIF(u,\perp,\perp,F);$

IF-d2$(v) :: \exists (u,z,\perp,B) \in KT_v \wedge z \in N_v \wedge (u,\perp,\perp,pif') \in KT_z \wedge pif' \in \{B,F\} \wedge (HS_z \neq CH) \wedge$
$\big((HS_v = CH) \vee (\forall x \in N_v/\{u,z\}, HS_x = O \vee (u,v,z,F) \in KT_x)\big) \wedge Desactive(v) = True$
$\longrightarrow UpdatePIF(u,z,\perp,F);$

IF-d3$(v) :: \exists (u,z,w,pif) \in KT_v \wedge (pif \in \{C,B\}) \wedge (z \in N_v) \wedge (u,w,\perp,pif') \in KT_z \wedge$
$(pif' \in \{B,F\}) \wedge (HS_z \neq CH) \wedge (HS_v \neq O) \wedge Desactive(v) = True \longrightarrow UpdatePIF(u,z,w,F);$

/* Règles de correction : corrigent les configurations initiales incorrectes, et lancent la phase de nettoyage */

RC$(v) :: (v,\perp,\perp,pif) \in KT_v \wedge \big((HS_v = NCH \wedge Ready_v = RO \wedge pif = F) \vee$
$(HS_v = NO \wedge Ready_v = RCH \wedge pif = F) \vee (HS_v = CH \wedge pif \in \{B,F\})\big) \longrightarrow UpdatePIF(v,\perp,\perp,C);$

IC-d1$(v) :: \exists (u,\perp,\perp,pif) \in KT_v \wedge (pif \in \{B,F\}) \wedge (u \in N_v) \wedge (u,\perp,\perp,C) \in KT_u \longrightarrow UpdatePIF(u,\perp,\perp,C);$

IC-d2$(v) :: \exists (u,z,\perp,pif) \in KT_v \wedge (pif \in \{B,F\}) \wedge (z \in N_v) \wedge (u,\perp,\perp,C) \in KT_z \longrightarrow UpdatePIF(u,z,\perp,C);$

IC-d3$(v) :: \exists (u,z,w,pif) \in KT_v \wedge (pif \in \{B,F\}) \wedge (z \in N_v) \wedge (u,w,\perp,C) \in KT_z \wedge$
$(HS_z \neq CH) \wedge (HS_v \neq O) \longrightarrow UpdatePIF(u,z,w,C);$

6.3. Définition du prédicat de sûreté

Par définition, la propriété de complétude dépend des protocoles de partitionnement et CNK (les variables $\mathtt{Status}_v$ et KT_v). Nous avons montré en Section 6.1, que la propriété de complétude n'est pas close pour les actions du protocole de partitionnement. Il a fallu établir des connaissances supplémentaires avant d'autoriser le changement de statut par le protocole de partitionnement.
L'une des contributions de cet article réside dans la définition (et les preuves associées) d'un prédicat de sûreté SP qui vérifie les conditions suivantes :

1. **Indépendance :** SP est définit indépendamment du protocole de partitionnement, et donc basé uniquement sur les variables du protocole CNK.
2. **Inclusion :** il existe un ensemble de configurations OC, à partir du quel si SP est vérifié, alors la propriété de complétude est vérifiée.
3. **Convergence et préservation :** Le protocole CNK garantit d'atteindre une configuration satisfaisant SP, à partir de laquelle SP reste vérifier après les actions du protocole CNK et les actions du protocole de partitionnement.

Définir un prédicat SP vérifiant ces conditions assure que toute exécution équitable du protocole CNK, atteint une configuration qui satisfait SP, et donc la propriété de complétude (condition d'inclusion). La propriété de complétude restera vérifiée après tout pas de calcul du protocole de partitionnement et CNK (condition d'indépendance, et de convergence et préservation). Les preuves que SP vérifie les conditions précédentes sont retirées dû au manque d'espace.

Définition 5 (Quasi-leader / Quasi-ordinaire)

• *Un nœud v est quasi-leader, soit parce qu'il est déjà leader ou parce qu'il peut devenir leader à tout instant. On note,* $\mathsf{QL}(v) \equiv (\mathsf{HS}_v = \mathsf{CH}) \vee (\mathsf{HS}_v = \mathsf{NO}) \vee (\mathsf{HS}_v = \mathsf{NCH} \wedge \mathtt{Ready}_v = \mathsf{RCH})$
• *Un nœud v est quasi-ordinaire, soit parce qu'il est ordinaire ou parce qu'il peut devenir ordinaire à tout instant. On note,* $\mathsf{QO}(v) \equiv (\mathsf{HS}_v = \mathsf{O}) \vee (\mathsf{HS}_v = \mathsf{NCH}) \vee (\mathsf{HS}_v = \mathsf{NO} \wedge \mathtt{Ready}_v = \mathsf{RO})$

Les nœuds voulant changer de statut devront être à la fois quasi-ordinaire et quasi-leader (agir comme un leader et comme un nœud ordinaire) avant de prendre leur nouveau statut.

Définition 6 (Prédicat de sûreté SP**)** *Chaque quasi-leader v connaît tous les quasi-leaders u, à distance au plus 3 de v, et accessible depuis v par un chemin comportant que des nœuds quasi-ordinaires.*

Complexité en temps. La convergence vers une configuration satisfaisant SP est achevée après 4 cycles asynchrones. Chaque requête de changement de statut provenant du protocole de partitionnement est satisfaite après au plus 9 cycles. Il s'agit du temps nécessaire pour achever une propagation d'information avec acquittement PIF.
Après la dernière requête de changement de statut, CNK converge vers une configuration légitime (qui est aussi terminale) en 4 cycles au plus. Dans une configuration légitime, en plus de la propriété de complétude, la propriété d'exactitude sera vérifiée.

Bibliographie

1. E. M. Belding-Royer. Multi-level hierarchies for scalable ad hoc routing. *Wireless Networks*, 9(5) :461–478, 2003.
2. A. Bui, A. K. Datta, F. Petit, et V. Villain. Snap-stabilization and PIF in tree networks. *Distributed Computing*, 20 :3–19, 2007.
3. S. Dolev et T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.
4. C. Johnen et F. Mekhaldi. Brief announcement : Robust self-stabilizing construction of bounded size weight-based clusters. In *SSS'09, Springer LNCS 5873*, pages 787–788, 2009.
5. C. Johnen et L. H. Nguyen. Robust self-stabilizing weight-based clustering algorithm. *Theoretical Computer Science*, 410(6-7) :581–594, 2009.
6. C. Johnen et S. Tixeuil. Route preserving stabilization. In *SSS'03, Springer LNCS 2704*, pages 184–198, 2003.
7. H. Kakugawa et T. Masuzawa. A self-stabilizing minimal dominating set algorithm with safe convergence. In *APDCM'06*, 2006.
8. S. Kamei et H. Kakugawa. A self-stabilizing approximation for the minimum connected dominating set with safe convergence. In *OPODIS'08, Springer LNCS 5401*, pages 496–511, 2008.