Détection de partition pour la gestion de groupes en environnement mobile

Muhammad Usman Bhatti et Denis Conan

GET / INT, CNRS UMR SAMOVAR
9 rue Charles Fourier
91011 Évry, France
Usman.Bhatti@gmail.com, Denis.Conan@int-evry.fr

21 Avril 2005

Résumé

Les réseaux sans fil sont de plus en plus omniprésents et introduisent de nouvelles problématiques dans la construction des applications et des systèmes répartis. Les déconnexions en sont un exemple typique. En plus de se déconnecter, les terminaux mobiles et les machines fixes sont sujets à des défaillances. Ces deux entraves créent des situations de partitionnement des entités d'une même application. Dans ce papier, nous présentons un service de détection de partition interopérable avec les systèmes de gestion de groupes. L'idée centrale de la proposition est de permettre au client du service de détection de partition de distinguer les nœuds défaillants, de ceux seulement partitionnés ou déconnectés.

1 Introduction

Les avancées récentes dans les technologies des réseaux sans fil locaux et personnels ainsi que dans les technologies des ordinateurs portables, des assistants personnels numériques et des téléphones mobiles amènent à grands pas les utilisateurs dans le monde de l'informatique mobile. Les utilisateurs demandent à avoir accès aux services et aux informations depuis n'importe quel lieu et tout en se déplaçant, accentuant ainsi leur dépendance vis-à-vis des applications informatiques. En conséquence, l'utilisation de nombreux matériels dans des environnements mobiles variés accroît les probabilités d'apparition de déconnexions, de défaillances et de partitions.

Nous considérons deux types de déconnexions : les déconnexions volontaires et les déconnexions involontaires. Les premières sont décidées par l'utilisateur depuis son terminal mobile ; les secondes sont le résultat de coupures intempestives des connexions physiques. Les travaux de recherche précédents [19] ont mis en exergue un algorithme de détection de connectivité permettant de décider si la requête de l'utilisateur peut être transmise ou non sur le réseau sans fil. Le principe est de surveiller les ressources locales

pour anticiper les manques (de mémoire, de bande passante...) rendant les communications difficiles, voire impossibles. Afin de protéger l'application des variations rapides, fréquentes et non significatives, les détecteurs de connectivité reposent sur un mécanisme d'hystérésis. Les détecteurs de connectivité sont locaux au nœud. Pour cette raison, [19] introduit le concept de détecteur de déconnexions qui diffuse les événements de déconnexion et de reconnexion.

Quant à la tolérance aux fautes, dans un système (complètement) asynchrone, il est impossible d'obtenir un consensus entre processus répartis dès qu'un seul d'entre eux est défaillant (par exemple, par une défaillance franche permanente) [10]. Cette impossibilité résulte de la difficulté de dire si le processus distant est juste lent ou s'il est défaillant. Pourtant, beaucoup de mécanismes de tolérance aux fautes nécessitent de telles décisions, ne serait-ce que pour déterminer de façon cohérente l'ensemble des processus vivants (non défaillants) à un instant donné. Pourtant, de nombreuses solutions de tolérance aux fautes existent déjà ; elles font l'hypothèse d'un modèle de système réparti moins faible appelé partiellement synchrone. Dans ce modèle, la plupart des solutions de tolérance aux fautes utilisent des détecteurs de défaillances non fiables [7]. Dans le même travail précédent [19], nous avons comparé les détecteurs de défaillances et de déconnexions, puis proposé un algorithme de consensus tolérant aussi bien les déconnexions que les défaillances.

L'objectif du travail présenté dans ce papier est la gestion de groupes de terminaux mobiles tolérant les défaillances et les déconnexions. Les systèmes de communication de groupes sont largement reconnus comme fondamentaux dans les solutions de tolérance aux fautes et les applications pour le travail collaboratif. Un tel service se décompose en deux partie : le premier service s'occupe de la gestion des groupes et le second de la diffusion de messages dans les groupes [9]. Dans les systèmes asynchrones, la gestion de groupes subit elle-aussi les effets du résultat d'impossibilité du consensus entre processus répartis dès qu'un seul d'entre eux est défaillant [16]. L'une des leçons des travaux précédents est qu'il est possible et même souhaitable de différencier les défaillances des déconnexions. Comme les déconnexions, les partitionnements du groupe global de terminaux mobiles en plusieurs sous-groupes provoquent une réduction ou une dégradation du service rendu aux utilisateurs, mais pas nécessairement une indisponibilité complète. Les partitions doivent continuer à évoluer comme des systèmes répartis autonomes ; c'est le concept des systèmes de communication de groupes partionnables [9].

Dans les environnements mobiles, les partitionnements sont monnaie courante et doivent être traités comme des situations normales. Dans la suite, nous nous attachons à distinguer les causes des partitionnements : partition créée à la suite soit d'une défaillance, soit d'une déconnexion volontaire ou involontaire, et ainsi à différencier les défaillances des déconnexions des partitions. La suite du papier est organisée comme suit. Après la définition du modèle de systèmes répartis présenté dans la section 2, dans la section 3, nous dessinons l'architecture du système complet avec les détecteurs et le service de gestion de groupes. La gestion de groupe est ensuite discutée plus en détail dans la section 4 et les algorithmes des détecteurs sont développés dans la section 5. Avant de conclure dans la section 7, nous présentons les travaux existants et positionnons notre contribution dans la section 6. Les preuves des algorithmes présentés sont présentés dans l'annexe.

2 Modèles de système réparti

Le système réparti est modélisé par un graphe orienté $G = (\Pi, \Lambda)$ où $\Lambda \subset \Pi \times \Pi$. Le système consiste en un ensemble de n processus $\Pi = \{p_1, p_2 ..., p_n\}$. Nous considérons deux types de liens : les liens équitables et les liens défaillants par arrêt franc et permanent de transmission de message. Un lien équitable peut perdre des messages par intermittence, mais si un processus p émet un message infiniment à destination du processus voisin q alors q reçoit finalement le message une infinité de fois. Chaque paire de processus est connecté par un chemin équitable. À la configuration, chaque processus connaît G. Pour plus de clarté dans la présentation, l'existence d'une horloge globale virtuelle $\mathcal{T}$ est supposée. $\mathcal{T}$ est inaccessible aux processus, elle prend ses valeurs dans l'ensemble des entiers naturels.

Les processus et les liens peuvent être défaillants à cause d'une défaillance franche et permanente. Les processus et les liens défaillants ne sont pas sujet à reprise. En outre, nous supposons qu'il existe toujours au moins un processus correct. Un processus peut se déconnecter volontairement ou involontairement. Il est supposé que tous les processus corrects commencent et terminent leur exécution connecté. Un processus q est accessible à partir d'un processus p (noté $p \rightarrow q$) s'il existe un chemin de p vers q. Si p et q sont mutuellement accessibles (noté $p \leftrightarrows q$) alors ils sont considérés comme appartenant à la même partition (ou groupe). Nous supposons que tous les processus corrects commencent et terminent leur exécution non partitionné : soit les processus partitionnés sont finalement exclus de l'application répartie et assimilés à des processus défaillants, soit les sous-groupes fusionnent en un seul groupe.

L'information supplémentaire utilisée dans le détecteur de partitions (cf. section 5.4) est la connaissance de la topologie globale au lancement de l'application répartie. En outre, nous supposons que la topologie ne change pas de manière implicite, un processus doit se reconnecter aux mêmes voisins. Sinon, de nouveaux liens sont créés et les anciens supprimés ; ces changements dans la topologie doivent être validés par un consensus. Dans cette étude, nous ne traitons pas le cas où le processus déconnecté n'arrive pas à contacter ses anciens voisins, qui se trouvent dans une autre partition après la reconnexion. Ces limites indiquent clairement que les travaux présentés ici sont une première approche. Nous travaillons actuellement sur une amélioration qui consiste à détecter les partitions uniquement via la connaissance des voisins, c'est-à-dire sans construction de la topologie globale. Ces résultats feront l'objet de publications ultérieures. En revanche, nous n'empêchons pas la découverte dynamique de nouveaux participants ni le départ volontaire du groupe, qui sont du ressort du gestionnaire de groupe. Dans ce cas, ce dernier avertit les détecteurs (de défaillances, de déconnexions. . .) du changement du graphe G. Ainsi, dans ce papier, nous nous focalisons sur la gestion d'un groupe fixe de processus Π. En conclusion, nous nous intéressons aux opérations *déconnecter* et *reconnecter*, et ne traitons pas les opérations *joindre* et *quitter* un groupe.

3 Architecture générale

L'architecture que nous avons choisie suit classiquement une orientation service dans laquelle l'une des briques de base est le détecteur de défaillances. En effet, les plateformes existantes dans la littérature commencent la plupart du temps par augmenter le système asynchrone d'un détecteur de défaillances afin de proposer aux couches supérieures un

système partiellement asynchrone. Par analogie, les déconnexions sont détectées dans un service dédié (détecteur de déconnexions). Comme représenté dans la figure 1, ces deux événements sont retravaillés par un service de détection de partitions pour être présentés au gestionnaire de groupes. L'algorithme de détection de partitions s'appuie sur la connaissance de la topologie complète au lancement de l'application répartie et maintient cette topologie pour en déduire les partitions. Par conséquent, le détecteur de partitions ne donne qu'une indication sans prendre de décision, par exemple, pour que deux groupes fusionnent en créant de nouveaux liens.

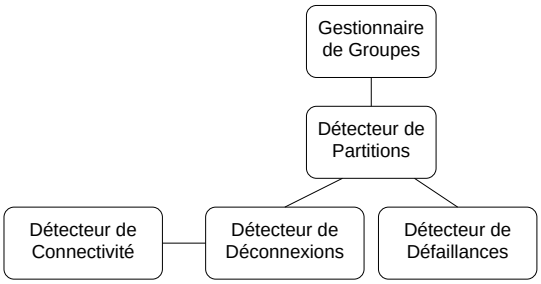

FIG. 1 – Architecture

Dans le détecteur de défaillances choisi : détecteur de défaillances à battements de cœur pour réseaux partitionnables [2], les processus envoient des messages vers les autres processus à une fréquence donnée, d'où le nom « battements de cœur ». Dans la suite, nous l'appellerons $\mathcal{HBP}$. Lorsque les processus sont vivants, la fréquence d'émission correspond naturellement à la fréquence de réception. Aussi, l'absence de réception de messages pendant une certaine durée permet de déclarer un processus défaillant. La détection de défaillances reste non fiable car ce sont peut-être les liens ou les nœuds intermédiaires dans le chemin qui sont défaillants. La figure 2-a montre un exemple de perte des battements de cœur suite à la défaillance du lien entre p et q, alors que les deux processus sont corrects. Dans la figure 2-b, sans connaissance supplémentaire, les processus q et t, et les processus r et s se suspectent réciproquement alors qu'ils sont tous corrects. Selon qu'ils sont optimistes ou pessimistes, les processus de l'application q et t tenterons ou non un possible changement de la topologie pour rejoindre r et s en créant de nouveaux liens. En conclusion, dans tous les cas, le détecteur de défaillances seul (sans les autres détecteurs) assimile les partitionnements, comme les déconnexions [19], à des défaillances.

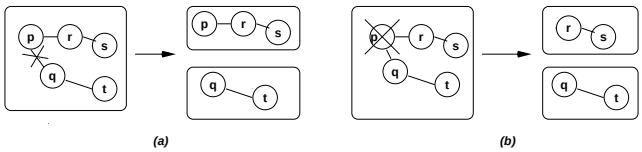

FIG. 2 – Partition suite à la défaillance *(a)* d'un lien ou *(b)* d'un processus

Alors que les détecteurs de défaillances ne permettent pas de différencier les défaillances des partitions, les détecteurs de déconnexions ne souffrent pas de cette impossibilité. En effet, une déconnexion est souvent prévisible et le processus qui se déconnecte

essaie de diffuser un message d'alerte [19]. Donc, mise à part dans le cas des déconnexions soudaines, les processus ont la possibilité de prévoir si la déconnexion provoque un partitionnement comme montré dans la figure 3. En outre, puisque nous supposons que la topologie ne change pas de manière implicite, le processus p de la figure 3 doit se reconnecter aux mêmes voisins q et r.

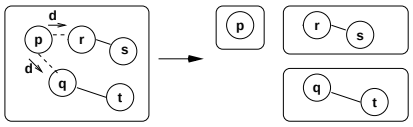

FIG. 3 – Partition suite à une déconnexion

4 Gestion de groupes

Dans cette section, nous utilisons le détecteur de partitions pour proposer le concept d'accord de partition (cf. section 4.1). La section section 4.2 montre ensuite l'utilisation de ce concept.

4.1 Accords de groupe et de partition

En plus de la gestion de groupes classique qui construit l'ensemble des processus accessibles ou appartenant au groupe (« accord de groupe »), nous mettons en exergue un nouveau concept algorithmique que nous appelons « accord de partition ». Il s'agit de mettre d'accord tous les processus d'un groupe, non seulement sur les processus vivants accessibles, mais aussi sur les processus défaillants, déconnectés et partitionnés. Nous proposons ce nouveau concept car, bien que dans les algorithmes existants les processus s'accordent sur l'ensemble des processus vivants accessibles, ils ne cherchent pas un consensus sur les autres processus. Dans les algorithmes de la littérature, chaque processus exclut du groupe un processus soit parce qu'il est défaillant soit parce qu'il est partitionné. Nous cherchons donc à ce qu'ils ajoutent le cas de la déconnexion, et en plus, qu'ils se mettent d'accord sur la cause de l'exclusion. Ensuite, des contre-mesures coordonnées deviennent applicables : attente, nouveau lien... Par conséquent, le gestionnaire de groupes dessiné dans la figure 1 au dessus du détecteur de partitions est plus qu'un gestionnaire de groupe classique. L'idée force de ce papier que nous développons dans la section suivante est donc la collaboration entre l'application et l'intergiciel dans la gestion de groupes. Par analogie avec la gestion du cache de composants logiciels [11] et la gestion de la réconciliation [8] pour la gestion des déconnexions, nous appliquons de manière systématique la stratégie de collaboration entre l'application et l'intergiciel dans les environnements mobiles [18]. La démarche est que l'intergiciel rende visible les abstractions du système réparti selon le bon niveau d'abstraction : dans notre cas, de la défaillance et de la connectivité, à la déconnexion, au partitionnement, et enfin, à l'accord de groupe et de partition.

4.2 Décision de l'application

Dans un système partiellement synchrone, le détecteur de défaillances n'est pas fiable, c'est-à-dire une suspicion peut être fausse et remise en cause. En conséquence, les détections de partitions ne sont qu'approximatives. Toute latitude est donc laissée à l'application cliente pour interpréter les spéculations de défaillances et de partitions. La seule assurance est que la détection se stabilise si le système réparti se stabilise : finalement, toutes les défaillances et partitions, et uniquement les vraies défaillances et partitions sont détectées. Il peut y avoir une approche pessimiste et une approche optimiste. Dans l'approche pessimiste, les processus de l'application constituant un « groupe majoritaire » appliquent des décisions conservatrices en décidant de ne pas attendre la confirmation de la détection de partitions et d'ignorer dorénavant les processus partitionnés comme s'ils étaient collectivement défaillants. Ils essayent de continuer à fournir les services dans un mode dégradé de manière auto-suffisante, hypothéquant ainsi une possible future fusion des deux sous-groupes. Les regroupements sont alors traités comme des changements de topologie avec ajouts de processus et ajouts de liens.

En revanche, dans l'approche optimiste, les processus peuvent appliquer des heuristiques optimistes envers les processus suspectés de partitionnement. L'idée sous-jacente est de parier sur les processus « placés derrière » les processus défaillants pour qu'ils soient encore vivants, et ainsi lorsqu'il y a un doute, de préférer voire un processus partitionné plutôt que défaillant. C'est l'approche qui est généralement employée pour les processus partitionnés à cause d'une déconnexion. Cela revient donc à assimiler les défaillances au même niveau de gravité que les déconnexions. Les processus d'une partition s'attendent ainsi à une fusion ultérieure des sous-groupes. Par exemple, ils gardent toujours en vue la possibilité d'une réconciliation, les décisions par consensus ne sont que des tentatives et l'utilisateur est averti que les actions restent à valider. Lorsque la partition fait suite à une déconnexion, l'espoir peut résider dans le fait que le terminal de l'utilisateur revienne dans le réseau sans fil et que le processus incriminé se reconnecte. Dans le cas d'une défaillance, les processus des deux côtés du processus fautif peuvent essayer de ré-ouvrir une connexion et ajouter un nouveau lien pour reconnecter les sous-groupes. Bien sûr, cette dernière mesure est aussi applicable en cas de déconnexion : il ne faut pas oublier qu'une déconnexion est provoquée suite à une notification par le détecteur de connectivité d'un manque de ressource telle que la bande passante, mais aussi telle que la batterie. Si c'est la bande passante du réseau sans fil local comme IEEE 802.11b qui pose problème, l'utilisateur peut commuter sur un réseau sans fil global comme GPRS. Si c'est la batterie qui est en cause, rien n'empêche alors de passer par un autre terminal mobile pour relier les sous-groupes. En conclusion, ces perspectives ouvrent un axe de recherche sur la modélisation/spécification des décisions/politiques des applications réparties face aux partitionnements, en s'appuyant sur le concept de détecteur de partitions.

5 Algorithmes

Maintenant que nous avons introduit les différentes catégories de détecteurs, nous précisons un peu plus le modèle de système réparti en donnant le schéma de défaillance et de déconnexion dans la section 5.1. Puis, nous présentons les trois détecteurs pour réseaux partitionnables tour à tour : détecteurs de défaillances, de déconnexions et de

partitions en sections 5.2, 5.3 et 5.4, respectivement.

5.1 Schéma de défaillance et de déconnexion

Le schéma de défaillance F_P est une fonction de $\mathcal{T}$ vers 2^{Π}, où $F_P(t)$ est l'ensemble des processus défaillants jusqu'à l'instant t. $crashed(F_P) = \bigcup_{t \in \mathcal{T}} F_P(t)$ représente l'ensemble des processus défaillants dans F_P et $correct(F_P) = \Pi \setminus crashed(F_P)$ représente l'ensemble des processus corrects dans F_P.

Le schéma de défaillance F_L est une fonction de $\mathcal{T}$ vers 2^{Λ}, où F_L est l'ensemble des liens défaillants jusqu'à l'instant. $crashed(F_L) = \bigcup_{t \in \mathcal{T}} F_L(t)$ représente l'ensemble des liens défaillants dans F_L et $correct(F_L) = \Pi \setminus crashed(F_L)$ représente l'ensemble des liens corrects dans F_L. Si $p \rightarrow q \in crashed(F_L)$, nous disons que le chemin entre p et q est défaillant dans F_L. Si $p \rightarrow q \notin crashed(F_L)$, nous disons que le chemin entre p et q est équitable dans F_L.

Le schéma de défaillance $F = (F_P, F_L)$ combine le schéma de défaillance des processus et celui des liens. L'historique H_{FD} d'un détecteur de défaillances est une fonction de l'ensemble $\Pi \times \mathcal{T}$ vers l'ensemble 2^{Π} où $H_{FD}(p,t)$ est la valeur du détecteur de défaillances du processus p à l'instant t dans H_{FD}. Si $q \in H_{FD}(p,t)$ alors p suspecte q à l'instant t dans H_{FD}. Un détecteur de défaillances $\mathcal{FD}$ est une fonction qui associe à chaque schéma de défaillance F_{FD} un ensemble d'historiques de détecteurs de défaillances $\mathcal{FD}(F)$. $\mathcal{FD}(F)$ est l'ensemble de tous les historiques pouvant exister durant les exécutions avec l'ensemble des défaillances F et le détecteur de défaillances $\mathcal{FD}$.

Le schéma de déconnexion D est une fonction de $\mathcal{T}$ vers 2^{Π}, où $D(t)$ est l'ensemble des processus déconnectés à l'instant t. $disconnected(D)$ représente l'ensemble des processus déconnectés dans D, et $connected(D) = \Pi \setminus disconnected(D)$ représente les processus connectés. En outre, si $p \in disconnected(D)$, nous disons que p est vu déconnecté dans D, et si $p \in connected(D)$, que p est vu connecté dans D. Nous supposons aussi que tous les processus de l'application répartie sont démarrés et terminés alors que la connectivité est bonne (mode « connecté » défini dans [19]). L'historique H_{DD} d'un détecteur de déconnexions est une fonction de l'ensemble $\Pi \times \mathcal{T}$ vers l'ensemble 2^{Π}, où $H_{DD}(p,t)$ est la valeur du détecteur de déconnexions du processus p à l'instant t dans H_D. Si $q \in H_{DD}(p,t)$ alors p voit q déconnecté à l'instant t. Un détecteur de déconnexions $\mathcal{DD}$ est une fonction qui associe à chaque schéma de déconnexion D un ensemble d'historiques de détecteurs de déconnexions $\mathcal{DD}(D)$. $\mathcal{DD}(D)$ est l'ensemble de tous les historiques pouvant exister durant les exécutions avec l'ensemble des déconnexions D et le détecteur de déconnexions $\mathcal{DD}$.

La partition du processus p relativement à F est dénommé $partition(p)$. Si p est défaillant ou déconnecté, l'ensemble $partition(p)$ est égal à l'ensemble $\{p\}$. L'historique H_{PD} d'un détecteur de partitions est une fonction de l'ensemble $\Pi \times \mathcal{T}$ vers l'ensemble 2^{Π}, où $H_{PD}(p,t)$ est la valeur du détecteur de partitions du processus p à l'instant t dans H_{PD}. Si $q \in H_{PD}(p,t)$ alors nous disons que p voit q partitionné à l'instant t.

5.2 Détecteur de défaillances

Pour des raisons de complétude et de meilleure compréhension, nous présentons rapidement le détecteur de défaillances $\mathcal{HBP}$. Lorsque les processus sont vivants, la fréquence

d'émission correspond naturellement à la fréquence de réception. Aussi, l'absence de réception de messages pendant une certaine durée permet de déclarer un processus défaillant. La détection de défaillances reste non fiable car ce sont peut-être les liens ou les nœuds intermédiaires dans le chemin qui sont défaillants. Dans l'algorithme 1, tout processus p exécute deux tâches concurrentes. Dans la première tâche, p émet périodiquement un message de la forme $(\mathsf{HB}, path)$ à tous ses voisins. La seconde tâche manipule la réception des messages $(\mathsf{HB}, path)$. Quand un tel message est reçu de q, p incrémente le compteur de tous les processus qui apparaissent après p dans $path$. Ensuite, p s'ajoute à $path$. $path$ représente donc le chemin parcouru par le battement de cœur. p fait suivre le battement de cœur à tous ses voisins s'il n'est pas présent plus d'une fois dans $path$. Dans le cas contraire, cela indique que le message est parti une première fois de p et qu'il a traversé certains nœuds pour revenir à p, auquel cas p en déduit que les processus après lui dans $path$ sont mutuellement accessibles. Bien sûr, pour construire l'ensemble des processus suspects, $\mathcal{HBP}$ doit être appelé par un algorithme introduisant la notion de temps : le nombre de battements de cœur reçus pendant une période est normalisé et l'ensemble des processus suspectés est construit. Cette tâche est accomplie par le détecteur de partitions.

Algorithme 1: Détecteur de défaillance pour réseaux partionnables $\mathcal{HBP}$

1 **for** every process p :
2 **initialisation** :
3 **for all** $q \in \Pi$ **bf** $D_p[q] \leftarrow 0$
4 **cobegin** :
5 $\|$ **task 1** : repeat periodically
6 $D_p[p] \leftarrow D_p[p] + 1$
7 **for all** $q \in neighbour(p)$ **do send**(HB, p) to q
8 $\|$ **task 2** : **upon receive**$(\mathsf{HB}, path)$ from q
9 **for all** $r \in \Pi$ such that r appears after p in $path$ **do** $D_p[r] \leftarrow D_p[r] + 1$
10 $path \leftarrow path.p$
11 **for all** $r \in neighbour(p)$ such that r appears at most once in $path$ **do send**$(\mathsf{HB}, path)$ to r
12 **coend**

5.3 Détecteur de déconnexions

Le détecteur de déconnexions pour les réseaux partitionnés $\mathcal{DDP}$ est présenté dans l'algorithme 2. C'est une version étendue de celui présenté dans [19] : les changements apportés permettent de supporter les réseaux orientés non complets et les liens non fiables équitables. Pour ce faire, nous définissons la nouvelle primitive *qrn_send*[1]. En utilisant cette primitive, un processus peut transmettre un message de manière fiable sur un lien équitable : l'émetteur envoie périodiquement son message jusqu'à la réception d'un acquittement par le récepteur, qui envoie un acquittement pour chaque message reçu. Dans

[1]La première lettre du préfixe *qrn* signifie en anglais *quiescent* pour silencieux : en un mot, la primitive arrête finalement d'envoyer des messages et se termine [1].

l'algorithme 2, la première tâche s'exécute suite à la notification d'un changement de mode émise par le détecteur de connectivité [19]. *mode* = 'd' indique le mode déconnecté ; un message de déconnexion est diffusé. *mode* ='c' indique le mode connecté ; un message de reconnexion est diffusé. La deuxième (*resp.* troisième) tâche est en écoute des messages de déconnexion (*resp.* reconnexion). Le processus ajoute l'émetteur à (*resp.* enlève l'émetteur de) son ensemble des processus vus déconnectés. Pour tolérer les messages qui se doublent, les déconnexions et les reconnexions sont numérotées et appariées. $\mathcal{DDP}$ implante un détecteur de déconnexions respectant les deux propriétés suivantes :

Complétude de déconnexion forte : il existe un instant après lequel tout processus correct qui se déconnecte est vu déconnecté par tous les processus corrects connectés. Formellement :
$\forall D,\ \forall H_D \in \mathcal{DD}(D),\ \exists t \in \mathcal{T},$
$\forall p \in \mathit{disconnected}(D),\ \forall q \in \mathit{connected}(D),\ \forall t' \geq t :$
$p \in H_D(q,\ t')$

Précision de déconnexion forte : Aucun processus p n'est vu déconnecté par un processus correct connecté avant que p ne soit déconnecté. Formellement :
$\forall D,\ \forall H_D \in \mathcal{DD}(D),\ \forall t \in \mathcal{T},\ \forall p,\ q \in \Pi - D(t) :$
$p \notin H_D(q,\ t)$

5.4 Détecteur de partitions

De manière similaire aux détecteurs de défaillances et de déconnexions, les détecteurs de partitions ne sont pas définis pour une implantation spécifique. Pour cela, nous définissons les propriétés abstraites suivantes de complétude et de précision : « complétude de partition » et « précision de partition ».

Complétude de partition forte : il existe un instant après lequel tout processus dans une partition est vu défaillant, déconnecté ou partitionné par tous les processus d'une autre partition. Formellement :
$\forall F = (F_P, F_L),\ \forall H_{FD} \in \mathcal{FD}(F),\ \forall H_{DD},\ \forall H_{PD},$
$\forall p, q \in \mathit{correct}(F_P),\ q \notin \mathit{partition}(p),\ \exists t \in \mathcal{T},$
$\forall t' \geq t : q \in H_{FD}(p, t') \vee q \in H_{DD}(p, t') \vee q \in H_{PD}(p, t')$

Précision de partition forte finale : il existe un instant après lequel aucun processus correct n'est vu partitionné avant qu'il ne soit partitionné. Formellement :
$\forall F = (F_P, F_L),\ \forall H_{FD} \in \mathcal{FD}(F),\ \forall H_{DD},\ \forall H_{PD},$
$\exists t \in \mathcal{T},\ \forall t' \geq t,\ \forall p, q \in \mathit{correct}(F_P),\ q \in \mathit{partition}(p)\ :$
$q \notin H_{PD}(p, t')$

Le détecteur de partitions $\mathcal{PDG}$ maintient la topologie complète du système réparti. La première tâche de l'algorithme 3 opère la détection de défaillances en comparant le nombre de battements de cœur au seuil k. Lorsqu'un processus est suspecté, il est aussitôt mis dans l'ensemble des processus défaillants f_p. Ensuite, le graphe est parcouru pour détecter si des processus deviennent partitionnés à cause de la nouvelle défaillance : la recherche est effectuée en enlevant les processus déconnectés (ensemble d_p), défaillants (ensemble f_p) et déjà partitionnés (ensemble p_p) à Π. S'il n'existe plus dans ce graphe de chemin équitable entre p et un processus q, alors q est ajouté à l'ensemble p_p. La boucle suivante effectue l'opération similaire, mais cette fois-ci pour détecter les fausses suspicions. Dans ce cas, le processus faussement suspecté q est enlevé de f_p et l'algorithme

Algorithme 2: Détecteur de déconnexion pour réseaux partitionnables $\mathcal{DDP}$

1 **for** every process p :
2 **initialisation** :
3 $disc_p \leftarrow \emptyset$ {Set of processes seen disconnected}
4 **for all** $q \in \Pi$ **do** $dn_p[q] \leftarrow 0$ {Vector of received disconnection numbers}
5 **cobegin** :
6 $\|$ **task 1** : **upon** change mode notification with ($mode_p$, $voluntary_p$)
7 **if** $mode_p =$ 'd' $\vee$ $voluntary_p$
8 **for all** $q \in neighbour(p)$ **do qrn_send**($\mathsf{DISC}, p, dn_p[p], p$) to q
9 $dn_p[p] \leftarrow dn_p[p] + 1$; $disc_p \leftarrow disc_p \cup \{p\}$; **notify** disconnection of p
10 **else if** $mode_p =$ 'c' $\wedge$ $\neg voluntary_p$
11 **for all** $q \in neighbour(p)$ **do qrn_send**($\mathsf{REC}, p, dn_p[p], p$) to q
12 $dn_p[p] \leftarrow dn_p[p] + 1$; $disc_p \leftarrow disc_p \setminus \{p\}$; **notify** reconnection of p
13 $\|$ **task 2** : **upon receive**($\mathsf{DISC}, q, n_q, path$)
14 **if** $q \notin disc_p \wedge dn_p[q] < n_q$
15 $path \leftarrow path.p$
16 **for all** r such that $r \in neighbour(p)$ and r appears at most once in $path$ **do qrn_send**($\mathsf{DISC}, q, n_q, path$)
17 $disc_p \leftarrow disc_p \cup \{q\}$; $dn_p[q] \leftarrow n_q$; **notify** disconnection of q
18 $\|$ **task 3** : **upon** receive($\mathsf{REC}, q, n_q, path$)
19 **if** $q \in disc_p \wedge dn_p[q] < n_q$
20 $path \leftarrow path.p$
21 **for all** r such that $r \in neighbour(p)$ and r appears at most once in $path$ **do qrn_send**($\mathsf{REC}, q, n_q, path$)
22 $disc_p \leftarrow disc_p \setminus \{q\}$; $dn_p[q] \leftarrow n_q$; **notify** reconnection of q
23 **coend**

teste si des processus partitionnés suite à cette fausse suspicion sont dans p_p par erreur : il s'agit de trouver un chemin entre p et un processus r de p_p en supposant que r est à nouveau vivant. À la fin de cette première tâche, si l'un des ensembles f_p, d_p ou p_p a été modifié, alors le détecteur de partitions envoie une notification de possible changement de vue de la partition. Enfin, la détection de défaillances doit respecter les propriétés de complétude forte et de précision forte finale [7].

Les deuxième et troisième tâches traitent respectivement les notifications de détection de déconnexions et de reconnexion en provenance du détecteur de déconnexions. Il est à noter que, contrairement aux détections de défaillances, ce sont de « vraies » déconnexions et reconnexions. Comme dans le cas des défaillances, les ensembles f_p, d_p et p_p sont mis à jour. Puisque les déconnexions sont sûres et que les défaillances approximatives, lorsqu'un processus q est dit défaillant et qu'une déconnexion est détectée alors q est enlevé de f_p et ajouté à d_p. Il en est de même pour les partitions. Par conséquent, un processus est d'abord vu déconnecté, puis défaillant ou partitionné. Certains processus voient q d'abord défaillant, d'autres d'abord partitionné, d'où la nécessité d'un consensus si certaines décisions en dépendent (cf. section 4.1). Enfin, il est aisé de diviser p_p en deux si l'on désire connaître la cause du partionnement d'un processus : défaillance ou déconnexion.

D'une manière générale, le détecteur de partitions $\mathcal{PDG}$ est utilisable par les gestionnaires de groupes de la littérature. Un algorithme tel que celui proposé dans [3] construit la gestion de groupes sur un ensemble de processus accessibles appelé $reachable_p$. Dans notre cas, $reachable_p$ est égal à $\Pi \setminus (d_p \cup f_p \cup p_p)$. Dans leur algorithme, les auteurs n'utilisent qu'un détecteur de défaillances pour construire $reachable_p$ et tolèrent le recouvrement des liens équitables. Dans la section 5.2, il est bien établi que les déconnexions et les partitions sont vues comme des défaillances par $\mathcal{HBP}$. Mise à part l'hypothèse de défaillance non permanente des liens, $\mathcal{HBP}$ serait donc remplaçable par $\mathcal{PDG}$ et $\mathcal{PDG}$ fournirait deux ensembles supplémentaires : d_p et p_p.

6 Travaux connexes

Les algorithmes de détection de défaillances sont le plus souvent conçus pour les réseaux locaux. C'est pourquoi la plupart d'entre eux ne passent pas à l'échelle, aussi bien en terme de distance entre les nœuds qu'en terme de nombre de participants. Concernant le premier aspect, par analogie avec le calcul des temporisations du protocole `TCP`, [4] montre comment implanter et adapter dynamiquement le détecteur de défaillances selon les paramètres du réseau, la vitesse de détection, la durée entre deux erreurs consécutives et la durée de correction des erreurs. Le détecteur de défaillances est structuré en deux couches : la première fournit une estimation des temps d'arrivée des messages pour optimiser le temps de détection et la seconde adapte le service de détection fourni par la première aux besoins de l'application. [12] modélise et évalue le fonctionnement des détecteurs de défaillances en environnement mobile. Les auteurs insistent sur l'asymétrie des communications sans fil en terme de performance : un PDA en réception est un goulot d'étranglement. Pour répondre au problème du nombre de participants, [5] agence les détecteurs de défaillances en deux niveaux : dans les réseaux locaux, les détecteurs de défaillances utilisent IP *multicast*, et entre réseaux locaux, ils utilisent UDP.

Algorithme 3: Détecteur de partition $\mathcal{PDG}$

1 **for** every process p :
2 **initialisation** :
3 **for all** $q \in \Pi$ **do** $oldD_p[q] \leftarrow 0$; $D_p[q] \leftarrow 0$ {Previous and current outputs of $\mathcal{HBDP}$}
4 $f_p \leftarrow \emptyset$; $d_p \leftarrow \emptyset$; $p_p \leftarrow \emptyset$ {Sets of faulty, disconnected, and partitioned processes}
5 $fchg \leftarrow false$ {Boolean stating whether f_p changes}
6 **cobegin** :
7 $\|$ **task 1** : repeat periodically
8 **if** $p \notin d_p$ {Current process not disconnected}
9 $D_p \leftarrow getD_p()$
10 **for all** $q \in \Pi \setminus (d_p \cup p_p)$
11 **if** $|D_p[q] - oldD_p[q]| \leq k \ \wedge \ q \notin f_p$ {k is the failure detection threshold}
12 $fchg \leftarrow true$; $f_p \leftarrow f_p \cup \{q\}$
13 **while** $\exists r \in \Pi \setminus (d_p \cup f_p \cup p_p) \ : \ \neg p \leftrightarrows q$ in $(\Pi \setminus (d_p \cup f_p \cup p_p), E)$ **do** $p_p \leftarrow p_p \cup \{r\}$
14 **for all** $q \in f_p$
15 **if** $|D_p[q] - oldD_p[q]| > k$ {False suspicion}
16 $fchg \leftarrow true$; $f_p \leftarrow f_p \setminus \{q\}$
17 **while** $\exists r \in p_p \ : \ p \leftrightarrows q$ in $(\Pi \setminus (d_p \cup f_p \cup p_p) \cup \{r\}, E)$ **do** $p_p \leftarrow p_p \setminus \{r\}$
18 **if** $fchg$ **then notify** new view {Possible new view}
19 $oldD_p \leftarrow D_p$; $fchg \leftarrow false$
20 $\|$ **task 2** : **upon** disconnection notification of q
21 $d_p \leftarrow d_p \cup \{q\}$
22 **if** $q \in f_q$ **then** $f_q \leftarrow f_q \setminus \{q\}$; **if** $q \in p_q$ **then** $p_q \leftarrow p_q \setminus \{q\}$ {q or the fair path was erroneously suspected}
23 **while** $\exists r \in \Pi \setminus (d_p \cup f_p \cup p_p) \ : \ \neg p \leftrightarrows q$ in $(\Pi \setminus (d_p \cup f_p \cup p_p), E)$ **do** $p_p \leftarrow p_p \cup \{r\}$
24 **notify** new view {Possible new view}
25 $\|$ **task 3** : **upon** reconnection notification of q
26 $d_p \leftarrow \emptyset$; **while** $\exists r \in p_p \ : \ p \leftrightarrows q$ in $(\Pi \setminus (d_p \cup f_p \cup p_p) \cup \{r\}, E)$ **do** $p_p \leftarrow p_p \setminus \{r\}$
27 **notify** new view {Possible new view}
28 **coend**

Une autre solution de la littérature pour le passage à l'échelle, cette fois-ci sans organiser les processus en une hiérarchie de groupes de détection, est dite à base de propagation de rumeurs (en anglais, *gossiping*) [15]. Les processus choisissent au hasard certains voisins pour transmettre les battements de cœurs, évitant ainsi la diffusion complète de tous les battements de cœurs. [6] oriente la rumeur en exploitant les informations sur les domaines et sous-domaines Internet. Notre solution peut aisément bénéficier de ces travaux à partir du moment où le détecteur de défaillances est conçu comme un composant, donc remplaçable, moyennant l'adaptation des interfaces. D'ailleurs, pour l'implantation en cours, nous étudions le prototype de [4, 5].

Parmi les domaines porteurs grands demandeurs de solutions de détection passant à l'échelle cités dans [9], on trouve les applications scientifiques diffusées sur de très nombreux nœuds répartis dans de nombreux réseaux locaux (en anglais, *grid computing*). En cas de défaillances et de partitions, les processus d'un sous-groupe doivent prendre la décision de redistribuer les calculs précédemment confiés aux processus partitionnés. Notre détecteur de partitions permettant de caractériser les causes de partitionnement peut aider dans la prise de décision.

Un autre domaine de l'informatique mobile pouvant bénéficier des travaux présentés dans ce papier et cité dans [9] est le domaine du travail collaboratif [20]. Dans ces applications, contrairement aux applications précédentes, les utilisateurs finaux veulent connaître les participants accessibles, ceux qui se sont déconnectés, ceux qui sont partis pour cause de défaillance du réseau ou de leur terminal mobile. Dans ces applications, les participants forment souvent un réseau logique au dessus d'Internet. Il s'ensuit que certains participants peuvent être momentanément inaccessibles à cause d'un autre participant déconnecté ou défaillant. Notre solution autorise une grande latitude dans les décisions suite à ces entraves, soit de la part de l'application de manière pro-active, soit à l'initiative des utilisateurs finaux.

Parmi les applications mobiles réparties fortement demandeuses de nos détecteurs et prometteuses en terme de popularité, les jeux mobiles multijoueurs sont en très bonne place. Dans [13], les joueurs sont regroupés selon l'état du jeu, demandant ainsi de fréquents changements de topologie. Ce critère de regroupement possède pour objectif de diminuer la latence du jeu et l'utilisation de la bande passante. L'hypothèse de ce jeu est que les joueurs ne peuvent agir que sur des éléments physiques (personnages, objets...) physiquement proches. Les groupes sont alors organisés hiérarchiquement, avec un nœud responsable des communications avec les groupes de la hiérarchie et de la gestion des changements de topologie. Les communications intra-groupe et inter-groupe utilisent soit un réseau IEEE 802.11b soit un réseau GPRS pour la couverture globale, ce dernier étant plus difficile d'utilisation à cause d'une latence de plusieurs secondes. Les auteurs prévoient de dédier la technologie GPRS pour la gestion des déconnexions. La zone du jeu est couverte par de nombreux réseaux sans fil, l'étude consistant à fournir un réseau logique performant avec *Mobile* IPv6. Les auteurs indiquent que les défaillances sont détectées localement. Le détecteur de partitions permettrait d'étendre la détection de défaillances à tous les participants et d'introduire la détection de déconnexions.

Toujours dans le domaine des jeux mobiles, dans [17], les auteurs cherchent à détecter les partitions pour savoir combien d'occurrences du jeu lancées, avec une occurrence par sous-groupe. Les nœuds « en bordure » d'un groupe sont responsables de la détection de partitions et plus particulièrement surveillés quant à leur défaillance. L'inconvénient

de cette approche est l'affectation de rôles spécifiques à certains nœuds car ces nœuds sont des points sensibles et le système réparti doit être reconfiguré à chaque défaillance d'un de ces nœuds et à chaque partitionnement. Au contraire, notre solution s'appuie sur les détecteurs de défaillances et de déconnexion pour fournir nativement la détection de partitions sans configuration spécifique.

7 Conclusions

Dans les environnements mobiles, les déconnexions et les partitionnements causés aussi bien par des déconnexions que par des défaillances ne sont pas des événements exceptionnels. Leur détection doit donc être effectuée par les intergiciels et être transparente aux applications. Il est alors du ressort de l'application de prendre les décisions de changements de structure ou de comportement. Parmi les applications clientes d'un mécanisme de détection de défaillances, de déconnexions et de partitions, ce papier étudie plus précisément les systèmes de communication de groupes. Ces systèmes sont déjà bien connus pour leur utilité dans le domaine de la tolérance aux fautes. Nous réutilisons donc les travaux existants pour y ajouter le traitement de la mobilité.

Dans ce papier, nous construisons une approche qui permet de distinguer et donc de traiter différemment les défaillances, les déconnexions et les partitions. Plus particulièrement, nous avons adaptés le détecteur de déconnexions présenté dans [19] pour les réseaux partitionnables tolérants les défaillances de liens équitables. Le détecteur de partitions récupère les événements de détections de défaillances et de déconnexions pour maintenir une vue complète du système réparti en classant les processus dans quatre ensembles : corrects, défaillants, déconnectés et partitionnés. Le dernier ensemble peut au besoin être divisé en deux sous-ensembles en distinguant les causes des partitionnements : partitionnés suite à des défaillances et partitionnés suite à des déconnexions. Ces ensembles forment une partition de l'ensemble des processus.

Puisque les événements de détection de défaillances et donc de partitions ne sont que des spéculations, les applications clientes du détecteur de partitions peuvent adapter leur comportement de façon optimiste ou pessimiste. L'application cliente ciblée dans ce papier est la gestion de groupes. Le paradigme gestion de groupes est naturellement très présent et très prisé dans les environnements mobiles avec par exemple les applications de travail collaboratif et les jeux multijoueurs en réseau. Nous avons montré comment le détecteur de partitions s'insère dans l'architecture d'un service de communication de groupes.

En guise de perspectives, comme les matériels ciblés sont des terminaux mobiles possédant peu de ressources, nous étudions la possibilité de rassembler les trois détecteurs en un seul pour optimiser les transmissions de messages. Nous devons aussi considérer les modèles de systèmes répartis autorisant les recouvrements des liens, voire des processus. En ce qui concerne la mobilité, nous devons traiter le problème du terminal mobile qui se déconnecte dans une partition et se reconnecte dans une autre partition, donc sans pouvoir faire un consensus sur le changement de topologie incluant les anciens voisins et les nouveaux. Enfin, concernant le prototypage, les trois détecteurs sont en cours de réalisation avec le modèle de composants Fractal [14].

Références

[1] M. Aguilera, W. Chen, and S. Toueg. Heartbeat : A Timeout-Free Failure Detector for Quiescent Reliable Communication. In M. Mavronicolas and P. Tsigas, editors, *Proc. 11th WDAG*, volume 1320 of *LNCS*, pages 126–140. Springer-Verlag, Sept. 1997.

[2] M. Aguilera, W. Chen, and S. Toueg. Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks. *Theoretical Computer Science*, 220(1) :3–30, June 1999.

[3] Z. Babaoglu, R. Davoli, and A. Montresor. Group communication in partitionable systems : Specification and algorithms. *IEEE TOSE*, 27(4) :308–336, 2001.

[4] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proc. 4th IEEE DSN*, June 2002.

[5] M. Bertier, O. Marin, and P. Sens. Performance Analysis of a Hierarchical Failure Detector. In *Proc. 4th IEEE DSN*, June 2003.

[6] M. Burns, A. George, and B. Wallace. Simulative performance analysis of gossip failure detection for scalable distributed systems. *Cluster Computing*, 2(3) :207–217, 1999.

[7] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *JACM*, 43(2), Mar. 1996.

[8] L. Chateigner, S. Chabridon, and G. Bernard. Intergiciel pour l'informatique nomade : réplication optimiste et réconciliation. In *Actes UbiMob*, June 2004.

[9] G. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications : A Comprehensive Study. *ACM CS*, 33(4) :427–469, Dec. 2001.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2) :374–382, Apr. 1985.

[11] N. Kouici, D. Conan, and G. Bernard. Caching Components for Disconnection Management in Mobile Envrionments. In Z. Tari *et al*, editor, *Proc. 6th DOA*, volume 3291 of *LNCS*, pages 1322–1379. Springer-Verlag, Oct. 2004.

[12] C. Marchand and J.-M. Vincent. Détecteurs de défaillances et qualité de service dans un réseau ad-hoc hétérogène. In *Actes de la 3è CFSE*, pages 525–536, Oct. 2003.

[13] K. Mitchell, D. McCaffery, G. Metaxas, J. Finney, S. Schmid, and A. Scott. Six in the City : Introducing Real Tournament—A mobile IPv6 Based Context-Aware Multiplayer Game. In *Proc. ACM Games*, pages 91–100, May 2003.

[14] ObjectWeb Consortium. Fractal Home Page. http ://fractal.objectweb.org, 2004.

[15] R. V. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical Report TR98-1687, Department of Computer Science, Cornell University, Ithaca, New-York, USA, 28, 1998.

[16] A. Ricciardi, A. Schiper, and K. P. Birman. Understanding Partitions and the "No Partition" Assumption. In *Proc. 4th IEEE FTDCS*, Sept. 1993.

[17] H. Ritter, R. Winter, J. Schiller, and T. Zippan. A Partition Detection System for Distributed Mobile Games. In *Proc. ACM Games*, Aug. 2004.

[18] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proc. 15th ACM PODC*, May 1996.

[19] L. Temal and D. Conan. Détections de défaillances, de connectivité et de déconnexions. In *Actes UbiMob*, pages 90–97. ACM International Conference Proceedings Series, June 2004.

[20] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proc. ACM CSCW*, pages 171–180, 2000.

Nous rassemblons dans cette section les preuves des nouveaux algorithmes présentés dans le papier : le détecteur de déconnexion $\mathcal{DDP}$ pour les réseaux partitionnables dans la section A et le détecteur de partition $\mathcal{PDG}$ dans la section B.

A Preuve de $\mathcal{DDP}$

Dans cette section, nous présentons la preuve de l'algorithme $\mathcal{DDP}$, donné en section 5.3.

Lemme 1 *Si deux processus p et q sont dans la même partition alors tout message émis par p est reçu par q.*

Preuve. Ce lemme fait parti des preuves données dans [2], dans le lemme 25 de leur algorithme présenté dans la figure 4. Nous la réutilisons ici pour nos besoins. Si p sont dans la même partition alors il existe un chemin équitable constitué de un ou plusieurs liens équitables de p vers q et chaque message émis une infinité de fois par p vers q est reçu une infinité de fois par q. Par conséquent, tous les messages émis par p avec la primitive *qrn_send* sont finalement reçus.

Lemme 2 ***Complétude de déconnexion forte*** *Il existe un instant après lequel tout processus correct qui se déconnecte est vu déconnecté par tous les processus corrects connectés.*

Preuve. Tout d'abord, regardons le scénario suivant. Un processus p émet un message de déconnexion DISC mais, puisque le lien n'est qu'équitable, le lien perd le message. Aucun processus ne reçoit le message de déconnexion avant la déconnexion effective et p est vu comme défaillant. Pour résoudre ce problème et pour rendre la transmission de ce message fiable, $\mathcal{DDP}$ utilise la primitive *qrn_send* qui, comme l'indique le lemme 1, rend la transmission sur les liens équitables fiable.

Nous devons considérer trois cas. Dans le premier, p envoie avec succès le message de déconnexion à tous ses voisins corrects connectés avec l'appel de la primitive *qrn_send* de la ligne 8 de l'algorithme 2. Toujours selon le lemme 1, ce message est finalement diffusé (ligne 16) à tous les processus accessibles, c'est-à-dire à tous les processus corrects connectés de la partition dans laquelle se trouvait p. Notons que la condition sur r et *path* dans la boucle $for\ all$ de la ligne 16 évite que le message de déconnexion soit transmis indéfiniment et que la diffusion ne s'arrête pas (ne soit pas « silencieuse »). Puisque le modèle de système réparti suppose que l'application répartie se termine avec une seule partition et tous les processus connectés, tous les processus corrects reçoivent finalement le message de déconnexion.

Dans le deuxième cas, p se déconnecte effectivement juste avant de réussir à transmettre un message DISC. Toujours selon le lemme 1, ce message est effectivement transmis lorsque p se reconnecte. Donc, p diffuse finalement le message de déconnexion à tous les processus de la partition. Notons cependant qu'il s'agit de la partition de la reconnexion, non celle de la déconnexion. Comme dans le premier cas, tous les processus corrects reçoivent finalement le message de déconnexion.

Dans le troisième cas, p réussit à transmettre son message de déconnexion à au moins un voisin correct connecté q. Si q ne se déconnecte pas juste après avoir reçu le message

de déconnexion de p, q diffuse finalement ce message de déconnexion à tous les processus corrects connectés de la partition. Si q se déconnecte avant de faire suivre le message de déconnexion de p, nous nous retrouvons dans le second cas. Enfin, si q arrive à transmettre le message de déconnexion de p à au moins un de ses voisins corrects connectés, nous nous retrouvons récursivement dans ce troisième cas. Clairement, le message de déconnexion de p est finalement reçu par tous les processus corrects connectés de la partition, puis tous les processus corrects de l'application répartie. Ceci termine la preuve du lemme.

Lemme 3 ***Précision de déconnexion forte*** *Aucun processus p n'est vu déconnecté par un processus correct connecté avant que p ne soit déconnecté.*

Preuve. Nous pouvons voir dans l'algorithme 2 qu'un processus n'ajoute un autre processus que lui-même à l'ensemble des processus déconnectés qu'à la ligne 17. Cette instruction n'est exécutée que lorsque le processus reçoit un message de déconnexion d'un processus qui n'est pas déjà dans $disc_p$ et si c'est un nouveau message de déconnexion : c'est-à-dire que ce n'est pas une « ancienne » déconnexion. En outre, le message de déconnexion était soit émis par le processus déconnecté (ligne 8) soit transmis par un processus de la partition qui fait suivre le message de déconnexion (lignes 13 et 16), construisant ainsi une vague dont l'initiateur est le processus déconnecté.

Théorème 1 *L'algorithme $\mathcal{DDP}$ implante un détecteur de déconnexion satisfaisant la complétude de déconnexion forte et la précision de déconnexion forte.*

Preuve. Immédiatement par les lemmes 2 et 3.

B Preuve de $\mathcal{PDG}$

Dans cette section, nous présentons la preuve de l'algorithme $\mathcal{PDG}$, donné en section 5.4. Nous devons montrer que $\mathcal{PDG}$ satisfait les propriétés de complétude forte et de précision forte finale, et de complétude de partition forte et de précision de partition forte finale.

Lemme 4 ***Complétude forte*** *Il existe un instant après lequel tout processus défaillant est suspecté par tous les processus de partition(p).*

Preuve. Dans [2], les auteurs démontrent que $\mathcal{HBP}$ satisfait la propriété suivante : à tout processus correct $p \in partition(p)$, le compteur de battement de cœur de tout processus $q \notin partition(p)$ est borné. Tout d'abord, la condition du test de la ligne 9 de l'algorithme 3 est vraie puisque $p \in partition(p)$. Ensuite, la première condition du test de la ligne 11 est vraie pour q puisque le compteur $D_p[q]$ est borné. En outre, puisque q était précédemment dans $partition(p)$, il n'était ni déconnecté (pas dans d_p), ni défaillant (pas dans f_p), ni partitionné (pas dans p_p). Donc, tous les processus dans $partition(p)$ exécutent finalement la ligne 12 et suspecte q.

Lemme 5 ***Précision forte finale*** *Il existe un instant après lequel tous les processus corrects ne sont suspectés par aucun processus correct de partition(p).*

PREUVE. La preuve est par contradiction. Supposons qu'un processus correct $q \in partition(p)$ est suspecté par p. Tout d'abord, dans [2], les auteurs démontrent que $\mathcal{HBP}$ satisfait la propriété suivante : à tout processus correct $p \in partition(p)$, le compteur de battement de cœur de tout processus correct $q \in partition(p)$ n'est pas borné. Comme il est suspecté, il existe un instant auquel la condition $(D_p[q] - oldD_p[q]) \leq k$ est vraie. Donc, les conditions des tests aux lignes 8, 10 et 12 de l'algorithme 3 sont vraies et q est ajouté à f_p. Puisque $D_p[q]$ n'est pas borné, la condition du test à la ligne 15 est vraie et q est enlevé de f_p à la ligne 16.

Théorème 2 *$\mathcal{PDG}$ implante un détecteur de défaillance qui satisfait les propriétés de complétude forte et de précision forte finale.*

PREUVE. Des lemmes 4 et 5.

Lemme 6 *Tout processus qui se déconnecte est finalement vu déconnecté et seulement déconnecté par tous les processus de $partition(p)$.*

PREUVE. Si un processus se déconnecte alors il envoie un message de déconnexion à tous ses voisins qui font suivre le message pour que tous les processus de la partition le reçoivent finalement. Selon le lemme 2, tous les détecteurs de déconnexion de la partition voient finalement la déconnexion et notifient leur détecteur de partition (ligne 17) de l'algorithme 2. Tous les détecteurs de partition de la partition exécutent donc finalement la tâche 2 de l'algorithme 3. Le processus déconnecté est finalement ajouté à l'ensemble d_p à la ligne 21 et enlevé des deux autres ensembles à la ligne 22. Seule la tâche 1 pourrait changer cette affectation, mais les lignes 8 et 10 excluent les processus déconnectés de la détection de défaillance.

Lemme 7 *Tout processus correct et connecté $q \notin partition(p)$ est finalement vu comme défaillant ou partitionné par tous les processus de $partition(p)$.*

PREUVE. La preuve est par contradiction. Supposons qu'un processus $q \notin partition(p)$ n'est ni dans f_p ni dans p_p. Deux cas sont à considérer : p est soit connecté soit déconnecté. Dans le premier cas, les conditions des tests aux lignes 8 et 10 de l'algorithme 3 sont vérifiées. Ensuite, la partition est due à une déconnexion ou due à la défaillance d'un processus r d'un chemin allant de p à q ou encore due à la défaillance d'un lien d'un chemin allant de p à q. Dans le premier sous-cas, r est vu soit défaillant soit déconnecté par p et est finalement ajouté à p_p à la ligne 13 ou à la ligne 23. Dans le deuxième sous-cas, q est finalement vu défaillant et est finalement ajouté à f_p à la ligne 12. Dans le troisième sous-cas, q est aussi finalement vu défaillant et ajouté à f_p à la ligne 12.

Dans le second cas, quand p se déconnecte, p ajoute finalement q à p_p à la ligne 23.

Lemme 8 ***Complétude de partition forte*** *Il existe un instant après lequel tout processus correct dans une partition est vu défaillant, déconnecté ou partitionné par tous les processus d'une autre partition.*

PREUVE. Supposons un processus correct $r \notin partition(p)$. Quatre cas sont à considérer : r est déconnecté (formant une partition à lui seul), r est connecté et un processus entre

r et p est déconnecté, r est connecté et un processus entre r et p est défaillant, ou r est connecté et un lien entre r et p est défaillant. Dans le second cas, le lemme 2 assure de manière évidente que r est finalement vu déconnecté par tous les processus corrects de $partition(p)$. Dans les trois derniers cas, les lemmes 6 et 7 assurent que r est finalement vu défaillant, déconnecté ou partitionné par tous les processus de $partition(p)$.

Lemme 9 ***Précision de partition forte finale*** *Il existe un instant après lequel aucun processus correct n'est vu partitionné avant qu'il ne soit partitionné.*

PREUVE. Supposons un processus r accessible du processus p via le processus q. Un processus r devient partitionné du processus p dans trois cas : q se déconnecte ou défaille, ou un lien entre r et p défaille, ou encore r se déconnecte. Le second cas est finalement considéré comme une défaillance, le résultat est le même que dans le premier cas. Si le processus q reste correct et qu'il n'y a pas de défaillances de liens, aucun message de déconnexion n'est généré. Les messages de déconnexion sont générés seulement lorsque le processus se déconnecte. La tâche 2 de l'algorithme 3 n'est donc pas exécutée et finalement aucun processus ne considère r comme partitionné. Si tous les processus reçoivent les battements de cœur en provenance de r, il ne sera pas suspecté et par conséquent la tâche 1 ne considère finalement pas r défaillant. Dans le dernier cas, la précision de déconnexion forte finale assure que r n'est pas partitionné avant qu'il ne soit déconnecté.

Théorème 3 $\mathcal{PDN}$ *implante un détecteur de partition satisfaisant la complétude de partition forte et la précision de partition forte finale.*

PREUVE. Des lemmes 8 et 9.