Gestion de groupes tolérant les défaillances et les déconnexions en environnement mobile

Tuan Dung Nguyen, Denis Conan

To cite this version:

Tuan Dung Nguyen, Denis Conan. Gestion de groupes tolérant les défaillances et les déconnexions en environnement mobile. NOTERE 2006 : 6ème Conférence Internationale sur les Nouvelles Technologies de la Répartition, Jun 2006, Toulouse, France. hal-01333299

HAL Id: hal-01333299
https://hal.science/hal-01333299

Submitted on 12 Jun 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Gestion de groupes tolérant les défaillances et les déconnexions en environnement mobile

Tuan Dung Nguyen* **— Denis Conan****

* *GET/ENST Bretagne*
Technopôle de Brest-Iroise, 29238 Brest cedex 3, France
td.nguyen@enst-bretagne.fr

** *GET / INT, CNRS Samovar*
9 rue Charles Fourier, 91011 Évry, France
denis.conan@int-evry.fr

RÉSUMÉ. *L'évolution des réseaux sans fil et des équipements a abouti au paradigme connu sous le nom « informatique mobile ». Il offre aux utilisateurs la capacité de pouvoir se déplacer tout en restant connecté aux applications réparties. Dans un environnement mobile, les terminaux mobiles sont sujets à des déconnexions. Ceci requiert des mécanismes spécifiques de gestion de déconnexions et de tolérance aux fautes. La gestion de groupes est une brique importante pour construire des applications réparties tolérantes aux fautes. Dans des travaux précédents, nous avons montré comment et dans quelle mesure des intergiciels existants peuvent être enrichis par quatre détecteurs : défaillances, connectivité, déconnexions et partitions. En se basant sur ces détecteurs, nous proposons dans cet article un nouveau service de gestion de groupes dans lequel les processus se mettent d'accord non seulement sur l'ensemble des processus corrects connectés du groupe mais aussi sur les ensembles des processus défaillants, déconnectés ou partitionnés.*

ABSTRACT. *The evolution of wireless devices and networks has been marked by the well-known "mobile computing" paradigm that enables users to move around while remaining connected to their distributed applications. In a mobile environment, mobile devices are prone to disconnections that require specific mechanisms for disconnection management and fault tolerance. Group membership is an important building block to construct fault-tolerant distributed applications. In previous works, we have shown how existing middleware can be extended with four detectors: failure, connectivity, disconnection and partition. Based on these detectors, we present in this paper a new group membership service in which processes agree not only on the set of correct connected processes but also on the other sets of faulty, disconnected or partitioned ones.*

MOTS-CLÉS : *mobilité, tolérance aux fautes, algorithmique répartie et gestion de groupes.*

KEYWORDS: *mobility, fault tolerance, distributed algorithms, and group membership.*

1. Introduction

Avec l'informatique mobile [SAT 01], sont apparues de nouvelles problématiques spécifiques aux environnements mobiles telles que la gestion de déconnexions. Les déconnexions sont volontaires ou involontaires. Les premières, décidées par l'utilisateur depuis son terminal mobile, sont justifiées par les bénéfices attendus sur le coût financier des communications, l'énergie, la disponibilité du service applicatif, et la minimisation des désagréments induits par des déconnexions inopinées. Les secondes sont le résultat de coupures intempestives des connexions physiques du réseau, par exemple, lors du passage de l'utilisateur dans une zone d'ombre radio. Cette problématique donne naissance à de nouveaux types de détecteurs comme les détecteurs de connectivité et de déconnexions [TEM 04].

La tolérance aux fautes est un sujet de recherche important pour les systèmes répartis [GÄR 99]. Ces systèmes disposent de deux propriétés importantes : la sûreté (un mauvais comportement n'apparaît jamais) et la vivacité (un bon comportement apparaît ultimement). Dans les systèmes purement asynchrones, c.-à-d. dans lesquels il n'existe aucune borne sur la vitesse de calcul de chaque processeur ou sur le temps de transmission des messages, [FIS 85] a montré qu'il est impossible de résoudre le problème du consensus entre processus répartis dès qu'un seul d'entre eux est défaillant par arrêt franc. La raison est que nous ne pouvons pas dire si un processus est défaillant ou simplement très lent. Au dessus des détecteurs de défaillances, les systèmes de communication de groupe (GCS pour *Group Communication Systems* en anglais) sont des briques fondamentales pour construire les applications réparties. Mais le principe d'un GCS s'appuie sur le consensus et donc subit aussi un résultat d'impossibilité dans les systèmes asynchrones [CHA 96a]. Pourtant, le problème du consensus peut être résolu dans un modèle partiellement synchrone où nous ne connaissons pas les délais de calcul ou de transmission mais dans lequel il existe un instant et une borne tels que, après cet instant, les délais sont inférieurs à cette borne. Un exemple du consensus dans ce modèle est construit en utilisant le concept de détecteur de défaillances non fiable [CHA 96b].

Certaines défaillances ou déconnexions créent des situations de partitionnement dans lesquelles le réseau est divisé en partitions et la communication est réalisable seulement entre processus de la même partition. Avec un détecteur de partitions, nous faisons la distinction entre les défaillances, les déconnexions et les partitions [BHA 05]. L'objectif de ce travail est donc de construire un service de gestion de groupes de processus partitionnables exploitant les détecteurs existants pour définir de nouvelles propriétés et de nouveaux services répartis, par exemple, en s'accordant sur l'ensemble des processus déconnectés, défaillants et partitionnés.

La suite du papier est organisée comme suit. Après l'étude bibliographique sur les détecteurs existants pour environnements mobiles dans la section 2 et sur les systèmes de communication de groupes dans la section 3, nous spécifions un nouveau service de gestion de groupes dans les sections 4 et 5. Ensuite, dans la section 6, nous présentons les algorithmes de la gestion de groupes. Enfin, nous concluons et donnons les perspectives de nos travaux dans la section 7.

2. Détecteurs en environnement mobile

Cette section présente les principes des quatre détecteurs pour environnements mobiles. Elle se termine en motivant l'idée de les utiliser pour un service de gestion de groupes.

Pour résoudre le problème de l'impossibilité du consensus dans les systèmes asynchrones dans lesquels les liens entre processus sont fiables [FIS 85], [CHA 96b] a proposé le concept de détecteur de défaillances non fiable. Chaque processus p est enrichi par un module de détection de défaillances local $\mathcal{FD}$ qui surveille l'ensemble des processus du système et donne à p une liste de processus suspectés d'être défaillants. Ce détecteur n'est pas fiable car il peut faire des erreurs : lorsqu'il détecte que la suspicion d'un processus q est inexacte, il enlève q de sa liste de suspects. Le détecteur de défaillances dans [CHA 96b] est présenté en donnant ses propriétés abstraites : *complétude* et *précision*. La première assure la capacité à suspecter les processus défaillants et la seconde restreint les erreurs qu'un détecteur peut faire. Dans le cas où les processus et les liens peuvent être défaillants et si nous ne cherchons que les algorithmes silencieux[1] (*quiescent* en anglais) [AGU 99], le plus faible des détecteurs de défaillances qui permet de résoudre le problème du consensus est le détecteur parfait ultime $\diamond\mathcal{P}$: complétude forte et précision forte ultime. Malheureusement, ce détecteur n'est pas réalisable, donc un nouveau type de détecteur de défaillances est proposé : le détecteur de défaillances dit « battements de cœur pour réseaux partitionnables » noté $\mathcal{HBP}$. Ce type de détecteur n'utilise pas de délai de garde (*timeout* en anglais) et sa sortie est un tableau de compteurs de battements de cœur de tous ses voisins au lieu d'une liste bornée de suspects. Le détecteur de défaillances pour réseaux partitionnables $\mathcal{HBDP}$ [BHA 05] se base sur $\mathcal{HBP}$ et ajoute en plus les informations sur la topologie du réseau en sauvegardant le chemin parcouru par le battement de cœur.

Le détecteur de connectivité $\mathcal{CD}$ [TEM 04] est introduit pour surveiller les ressources locales (niveau de la batterie du terminal mobile ou pourcentage de la bande passante du réseau sans fil) afin d'anticiper les déconnexions. L'algorithme est basé sur un mécanisme d'hystérésis à deux doubles seuils pour lisser les variations du niveau de disponibilité d'une ressource et donc évite l'effet *ping-pong*, c.-à-d. un niveau de disponibilité de ressource qui oscille autour d'un seuil provoque un seul changement de mode.

Les informations de déconnexions et de reconnexions fournies par le détecteur de connectivité $\mathcal{CD}$ restent toutefois locales. Le détecteur de déconnexions pour réseaux partitionnables $\mathcal{DDP}$ [TEM 04, BHA 05] est alors introduit pour échanger ces informations entre processus. Si un processus p reçoit un message de déconnexion (resp. reconnexion) d'un processus q, il ajoute (resp. enlève) q de son ensemble des processus vus déconnectés. Si la déconnexion est trop rapide, le processus est considéré comme défaillant : cela exprime le fait que nous sommes bien sûr limités par le résultat d'impossibilité de [FIS 85], mais essayons de construire la vue la plus pré-

1. Les processus arrêtent ultimement de retransmettre (indéfiniment) leurs messages en cas de défaillance de liens.

cise possible du système. Comme le détecteur de défaillances $\mathcal{HBP}$, $\mathcal{DDP}$ est décrit de manière abstraite : complétude de déconnexion forte et précision de déconnexion forte [BHA 05].

La partition peut survenir suite à la défaillance d'un lien (cf. figure 1.a) ou d'un processus (cf. figure 1.b). Elle peut aussi être la conséquence d'une déconnexion d'un processus (cf. figure 1.c). Comme les deux détecteurs précédents, le détecteur de partitions est présenté avec deux propriétés abstraites de complétude et de précision : complétude de partition forte et précision de partition forte finale [BHA 05]. L'algorithme $\mathcal{PDG}$ s'appuie sur les connaissances sur la topologie globale du système. Il construit et maintient un graphe de tous les processus et de tous les liens et l'utilise pour détecter les partitions.

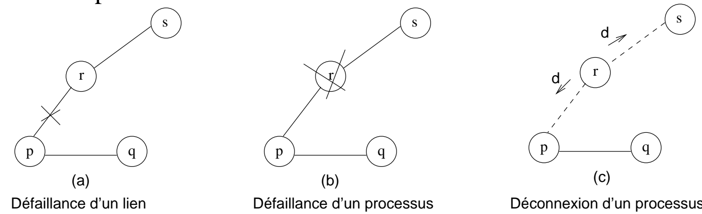

Figure 1. *Situations de partitionnement*

Toutefois, le détecteur de partitions n'assurer pas la cohérence entre différents processus. Dans la figure 1.c, nous pouvons considérer plusieurs scénarios qui provoquent une partition. Par exemple, les sorties des détecteurs de partitions des processus p et q peuvent être incohérentes comme dans le tableau 1 : la déconnexion de r crée une situation de partitionnement ; p détecte que r est déconnecté et alors s devient partitionné ; par contre, q ne reçoit pas le message de déconnexion de r avant de considérer s défaillant.

	vus défaillants	vus déconnectés	vus partitionnés
p	$\emptyset$	$\{r\}$	$\{s\}$
q	$\{s\}$	$\emptyset$	$\emptyset$

Tableau 1. *Incohérence des détecteurs de partitions*

Pour résoudre ce problème, nous utilisons un service de gestion de groupes au dessus de $\mathcal{PDG}$. En plus de construire l'ensemble des processus accessibles ou appartenant au groupe (« accord de groupe »), nous définissons un nouveau concept algorithmique appelé « 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 [BHA 05]. Ainsi, dans le scénario du tableau 1, p et q se mettent d'accord par exemple pour considérer r vu déconnecté et s vu partitionné.

3. Système de communication de groupes

Dans les systèmes de communication de groupes (GCSs), un groupe est un ensemble de processus, appelés les membres du groupe. Les processus dans un groupe communiquent en envoyant un message au groupe identifié par un nom unique ; le GCS s'occupe de transmettre le message à tous les membres. Un processus peut

joindre un groupe existant, quitter son groupe ou bien être éliminé du groupe à cause d'une défaillance, d'une déconnexion ou d'une partition. Un GCS se décompose en deux services imbriqués [CHO 01, DEL 01] : gestion de groupes et diffusion de messages. Le premier service s'occupe de former et de maintenir la composition des groupes au cours de l'exécution. La sortie de ce service est une vue qui contient une liste des processus du groupe et un identificateur unique. Cette vue est dynamique et l'objectif de ce service est d'assurer la cohérence de ces vues. Pour ce faire, la gestion de groupes s'appuie sur un algorithme de consensus entre les membres. Le deuxième service diffuse des messages aux membres du groupe en assurant les propriétés comme la diffusion fiable, causale ou atomique. Dans le cadre de ce travail, nous ne nous intéressons qu'à la gestion de groupes.

Les canevas logiciels les plus souvent cités sont Isis, Horus, Totem, Phoenix, Ensemble, Relacs, Jgroup, Javagroup (cf. [CHO 01] pour les références). Le premier critère de classification nous intéressant ici est le support des partitionnements : GCSs à composant primaire (p. ex. Isis) et GCSs partitionnables (p. ex. Ensemble, Javagroup, Jgroup, Totem). Les GCSs à composant primaire maintiennent une seule vue de la composition courante du groupe. En revanche, les GCSs partitionnables autorisent la coexistence de plusieurs vues afin de modéliser les partitions dans le réseau. En ce qui concerne la structure, nous distinguons deux classes [MEN 03] : monolithique (p. ex. Isis, Totem) et modulaire (p. ex. Ensemble, Javagroup, Jgroup). Les systèmes modulaires sont plus faciles à modifier pour ajouter de nouveaux composants. Nous cherchons donc un GCS partitionnable et modulaire dont la spécification ne subit pas de faiblesses déjà connues [ANC 95].

Ensuite, un autre critère important est la facilité d'étude et d'adaptation pour ajouter de nouvelles fonctionnalités. Nous devons donc regarder le langage de programmation utilisé, et la disponibilité du code source et de la documentation (spécification, conception et implantation). Les deux candidats satisfaisant aussi ces derniers critères et parmi les implantations les plus actives sont Jgroup et Javagroup. Ils sont développés en Java avec une licence logiciel libre. Après une étude de cas, nous avons choisi Jgroup qui est un GCS partitionnable et modulaire ayant une spécification formelle compacte [BAB 01].

4. Modèle de système réparti

Avant de présenter la spécification du nouveau service de gestion de groupes, nous présentons le modèle de système réparti repris en partie de [CHA 96b]. Le système réparti est modélisé par un graphe orienté $G = (\Pi, \Lambda)$, où $\Lambda \subset \Pi \times \Pi$, Π est l'ensemble de n processus $\Pi = \{p_1, p_2, \ldots, p_n\}$ et Λ est l'ensemble des liens entre eux. Chaque paire de processus est connectée par un lien équitable non fiable, c.-à-d. ce lien peut perdre des messages par intermittence, mais si un processus p émet un message m une infinité de fois à destination du processus voisin q alors q reçoit ultimement m une infinité de fois. Un chemin équitable est un chemin dont tous les liens sont équitables. Pour plus de clarté dans la présentation, nous considérons l'existence d'une horloge globale virtuelle $\mathcal{T}$ qui prend ses valeurs dans l'ensemble des entiers naturels, mais elle n'est pas accessible aux processus. En outre, les processus et les liens sont sujets

à des défaillances franches et permanentes. Un processus peut se déconnecter volontairement ou involontairement. Un processus q est accessible à partir d'un processus p (noté $p \rightarrow q$) s'il existe un chemin équitable de p vers q. S'il n'en existe pas, on dit que q n'est pas accessible à partir de p ($p \nrightarrow q$). Si p et q sont mutuellement accessibles (noté $p \leftrightarrow q$) alors ils sont considérés comme appartenant à la même partition. Sinon, ils sont considérés comme n'appartenant pas à la même partition ($p \nleftrightarrow q$).

Comme dans [BAB 01], l'exécution d'un programme réparti résulte en ce que chaque processus exécute un événement (pouvant être l'événement nul noté ϵ), choisi à partir d'un ensemble $\mathcal{S}$, à chaque top d'horloge. $\mathcal{S}$ contient au moins deux événements $send()$ et $recv()$, et la notification de l'installation d'une nouvelle vue $vchg()$. L'historique global d'une exécution est une fonction de $\Pi \times \mathcal{T}$ vers $\mathcal{S} \cup \{\epsilon\}$. Si un processus p exécute un événement e à l'instant t, nous notons $\sigma(p,t) = e$. Sinon, $\sigma(p,t) = \epsilon$, c.-à-d. p n'exécute aucun événement à l'instant t.

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 [TEM 04]). La partition du processus p est dénommée $partition(p)$. Si p est défaillant ou déconnecté, l'ensemble $partition(p)$ est égal au singleton $\{p\}$.

5. Propriétés du nouveau service de gestion de groupes

Nous présentons dans cette section la spécification d'un service de gestion de groupes augmenté de nouvelles propriétés grâce aux détecteurs en environnement mobile. Dans la littérature, un service de gestion de groupes classique s'occupe de former et de maintenir une *vue*, c.-à-d. une liste de processus corrects et connectés[2] appartenant à un groupe. Chaque changement dans la liste des membres du groupe provoque l'installation d'une nouvelle vue. Pour les distinguer entre elles, chaque vue dispose d'un identificateur unique. Formellement :

– $V(p,t)$ est la vue courante du processus p à l'instant t, $V(p,t) = \left\langle VID, \overline{V(p,t)} \right\rangle$, dans laquelle VID est l'identificateur de la vue $V(p,t)$ et $\overline{V(p,t)}$ est l'ensemble des processus du groupe.

Nous proposons dans ce travail le nouveau concept de « vue augmentée » qui se compose de la vue classique (ici appelée la vue du groupe V_C) plus les trois ensembles des processus défaillants, déconnectés et partitionnés. Formellement :

– $V_a(p,t)$ est la vue augmentée courante du processus p à l'instant t avec la relation $\overline{V_a(p,t)} = \overline{V_C(p,t)} \cup \overline{V_F(p,t)} \cup \overline{V_D(p,t)} \cup \overline{V_P(p,t)}$ dans laquelle :

- $\overline{V_C(p,t)}$ est l'ensemble des processus corrects et connectés. Si p est défaillant ou déconnecté, $\overline{V_C}$ de p est égal à $\{p\}$;
- $\overline{V_F(p,t)}$ est l'ensemble des processus défaillants ;
- $\overline{V_D(p,t)}$ est l'ensemble des processus vus déconnectés ;

2. La notion de processus connecté n'existe pas dans ces travaux car les déconnexions ne sont pas traitées. Cela revient ici à dire que les processus de la vue sont connectés.

- $\overline{V_P(p,t)}$ est l'ensemble des processus vus partitionnés.

L'ensemble des processus corrects et connectés V_C de la vue augmentée courante est appelé la *vue du groupe*, son ensemble des processus défaillants V_F la *vue des défaillances*, son ensemble des processus vus déconnectés V_D la *vue des déconnexions* et son ensemble des processus vus partitionnés V_P la *vue des partitions*.

Dans les cinq propriétés présentées ci-après, les modifications par rapport à la solution d'origine [BAB 01] sont constatées dans AGM1, AGM2 et AGM5 ; AGM3 et AGM4 restent inchangées. Pour les deux premières propriétés sur la précision et la complétude des vues, nous remplaçons la vue dans [BAB 01] par la vue du groupe et ajoutons de nouvelles sous-propriétés concernant les autres vues. En outre, une nouvelle sous-propriété pour l'intégrité des vues est ajoutée dans AGM5.

AGM1 : Précision des vues
(1) S'il existe un instant après lequel un processus correct q devient accessible à partir du processus correct p, alors ultimement q est inclus dans la vue du groupe V_C de p. Formellement : $\exists t_0 \in \mathcal{T}, \forall t \geq t_0 : p, q \in correct(F_P) \wedge p \rightarrow q \Rightarrow (\exists t_1 \in \mathcal{T}, \forall t \geq t_1 : q \in \overline{V_C(p,t)})$
(2) Il existe un instant après lequel aucun processus correct n'apparaît dans les vues des défaillances, des déconnexions ou des partitions avant qu'il ne soit défaillant, déconnecté ou partitionné. Formellement :
$\exists t_0 \in \mathcal{T}, \forall t \geq t_0, \forall p, q \in correct(F_P) : q \in partition(p) \Rightarrow (\exists t_1 \in \mathcal{T}, \forall t \geq t_1 : q \notin \overline{V_F(p,t)} \wedge q \notin \overline{V_D(p,t)} \wedge q \notin \overline{V_P(p,t)})$

AGM2 : Complétude des vues
(1) S'il existe un instant après lequel tout processus d'une partition Θ devient inaccessible à partir des autres processus de Π, alors ultimement la vue du groupe V_C de tous les processus corrects hors de Θ ne contient aucun processus de Θ. Formellement :
$\exists t_0 \in \mathcal{T}, \forall t \geq t_0, \forall q \in \Theta, \forall p \notin \Theta : p \nrightarrow q \Rightarrow (\exists t_1 \in \mathcal{T}, \forall t \geq t_1, \forall r \in correct(F_P) - \Theta : \overline{V_C(r,t)} \cap \Theta = \emptyset)$
(2) Il existe un instant après lequel tout processus correct dans une partition est inclus dans la vue des défaillances, des déconnexions ou des partitions des processus des autres partitions. Formellement :
$\exists t_0 \in \mathcal{T}, \forall t \geq t_0, \forall p, q \in correct(F_P), q \notin partition(p) \Rightarrow (\exists t_1 \in \mathcal{T}, \forall t \geq t_1 : q \in \overline{V_F(p,t)} \vee q \in \overline{V_D(p,t)} \vee q \in \overline{V_P(p,t)})$

Les deux propriétés sur la précision et la complétude des vues sont importantes pour tous les GCSs. Sans la propriété AGM1, la spécification est facilement satisfaite par l'installation des vues capricieuses [ANC 95]. D'autre part, l'absence de AGM2 permet des GCSs dans lesquels les vues contiennent toujours tous les membres du groupe [BAB 01].

AGM3 : Cohérence des vues
(1) Si un processus correct p installe la vue du groupe v alors tous les processus dans v installent aussi v ou p installe ultimement un successeur immédiat de v. Formellement : $p \in correct(F_P) \wedge vchg(v) \in \sigma(p, \mathcal{T}) \wedge q \in \overline{v} \Rightarrow (vchg(v) \in \sigma(q, \mathcal{T})) \vee (\exists w : v \prec_p w)$
(2) Si deux processus p et q installent initialement la même vue du groupe v, puis p installe un successeur immédiat de v alors ultimement soit q installe aussi un succes-

seur immédiat de v, soit q est défaillant. Formellement :
$vchg(v) \in \sigma(p, \mathcal{T}) \wedge vchg(v) \in \sigma(q, \mathcal{T}) \wedge v \prec_p w_1 \wedge q \in correct(F_P) \Rightarrow \exists w_2 : v \prec_q w_2$
(3) Quand un processus p installe une vue du groupe w comme successeur immédiat de la vue du groupe v, tous les processus survivants de v à w avec p avaient précédemment installé v. Formellement :
$\sigma(p, t_0) = vchg(w) \wedge v \prec_p w \wedge q \in \overline{v} \cap \overline{w} \Rightarrow vchg(v) \in \sigma(q, [0, t_0[)$

AGM4 : Ordre des vues
L'ordre dans lequel les processus installent successivement des vues du groupe est une relation d'ordre partiel. Formellement : $v \prec^* w \Rightarrow w \not\prec^* v$

Avec les GCSs partitionnables, il est impossible d'obtenir un ordre total sur l'installation des vues. La propriété AGM4 signifie que si deux vues sont déjà installées par un processus dans un ordre, elles ne peuvent pas être installées dans un ordre opposé par un autre processus.

AGM5 : Intégrité des vues
(1) Chaque vue augmentée installée par un processus p doit inclure ce processus dans la vue du groupe. Formellement : $vchg(V) \in \sigma(p, \mathcal{T}) \Rightarrow p \in \overline{V_C}$
(2) L'intersection des vues du groupe, des défaillances, des déconnections et des partitions est l'ensemble vide. Formellement : $\forall t \in \mathcal{T} : \overline{V_C(p,t)} \cap \overline{V_F(p,t)} \cap \overline{V_D(p,t)} \cap \overline{V_P(p,t)} = \emptyset$

6. Algorithmes

Dans cette section, nous présentons tout d'abord l'idée principale et la structure générale de l'algorithme de gestion de groupes qui se base sur celui de Jgroup [BAB 01]. Ensuite, nous donnons une explication détaillée du fonctionnement et nos modifications par rapport à l'original. Pour faciliter la comparaison, nous utilisons la même notation et la même organisation que dans [BAB 01].

6.1. *Présentation générale*

La structure générale de l'algorithme se trouve dans l'algorithme 1 qui est déclenché par les événements générés par $\mathcal{PDG}$. Chaque processus entre dans la procédure *AgreementPhase* (cf. algorithme 2). Cette procédure se compose de deux sous-procédures : *SynchronisationPhase* et *EstimateExchangePhase*. La première a pour but de réaliser une synchronisation avec les processus qui ne sont pas encore dans la même phase de l'algorithme. La deuxième essaie d'obtenir un consensus entre les membres du groupe sur la composition d'une nouvelle vue en échangeant leur estimation. La décision est donnée à la fin par un coordinateur choisi entre les membres du groupe (cf. algorithme 3).

L'algorithme 4 donne quelques procédures et fonctions supplémentaires utilisées dans les algorithmes : *InitiateEstimatePhase* est utilisée pour envoyer une estimation initiale ; *SendEstimate* est utilisée pour mettre à jour son estimation et l'échanger avec d'autres processus ; *CheckAgreement* vérifie la condition d'accord entre les membres ; *InstallView* installe une nouvelle vue et informe les autres de cette installation.

Algorithme 1: Algorithme principal d'un processus p

```
1  initialisation :
2    reachable ← {p} ; failure ← ∅ ; disconnect ← ∅ ; partition ← ∅
3    version ← (0, . . . , 0) ; symset ← ({p}, . . . , {p})
4    view ← (UniqueID(), ({p}, ∅, ∅, ∅)) ; cview ← view                     % initial view
5    generate vchg(view)                                                     % install the initial view
6    while true do
7      wait until event
8      case event of
9        msuspect(f_p, d_p, p_p) :                                           % from PDG
10         P ← f_p ∪ d_p ∪ p_p
11         for all r ∈ (Π \ P) \ reachable do symset[r] ← reachable         % new reachable process
12         msend(⟨SYMMETRY, version, reachable⟩, (Π \ P) \ reachable)
13         reachable ← Π \ P ; failure ← f_p ; disconnect ← d_p ; partition ← p_p   % update relevant sets
14         AgreementPhase()
15       mrecv(⟨SYNCHRONIZE, V_p, V_q, P⟩, q) :                              % synchronization request from q
16         if version[q] < V_q then                                          % avoid obsolete messages
17           version[q] ← V_q
18           if q ∈ reachable then AgreementPhase()
```

Algorithme 2: *AgreementPhase* et *SynchronizationPhase*

```
1  procedure AgreementPhase()
2  repeat
3    estimate ← (reachable, failure, disconnect, partition)                % initial estimate
4    version[p] ← version[p] + 1                                             % new agreement phase
5    SynchronisationPhase()
6    EstimateExchangePhase()
7  until stable
8  procedure SynchronisationPhase()
9    synchronized ← {p}                                                      % set of synchronized processus
10   for all r ∈ estimate.comp \ {p} do                                      % send synchronization request
11     msend(⟨SYNCHRONIZE, version[r], version[p], symset[r]⟩, {r})
12   while (estimate.comp ⊈ synchronized) do                                 % wait for synchronization with others
13     wait until event
14     case event of
15       msuspect(f_p, d_p, p_p) :                                           % From PDG
16         P ← f_p ∪ d_p ∪ p_p
17         for all r ∈ (Π \ P) \ reachable do symset[r] ← reachable
18         msend(⟨SYMMETRY, version, reachable⟩, (Π \ P) \ reachable)
19         reachable ← Π \ P ; failure ← f_p ; disconnect ← d_p ; partition ← p_p
20         estimate.comp ← estimate.comp ∩ reachable ; estimate.fail ← estimate.fail ∪ failure
21         estimate.disc ← estimate.disc ∪ disconnect ; estimate.part ← estimate.part ∪ partition
22       mrecv(⟨SYMMETRY, V, P⟩, q) :                                        % correct asymmetry problem
23         if (version[p] = V[p]) and (q ∈ estimate.comp) then
24           estimate.comp ← estimate.comp \ P
25       mrecv(⟨SYNCHRONIZE, V_p, Vq, P⟩, q) :                               % synchronization request from q
26         if version[p] = V_p then                                          % avoid obsolete messages and send back response
27           synchronized ← synchronized ∪ {q}
28         if version[q] < V_q
29           version[q] ← V_q ; agreed[q] ← V_q
30           msend(⟨SYNCHRONIZE, version[q], version[p], symset[q]⟩, {q})
31       mrecv(⟨ESTIMATE, V, E⟩, q)                                          % Estimation from q
32         version[q] = V[q]
33         if q ∉ estimate.comp then                                         % correct asymmetry problem
34           msend(⟨SYMMETRY, version, estimate.comp⟩, {q})
35         else if (version[p] = V[p]) and (p ∈ E.comp) then                 % update the own estimation
36           estimate.comp ← estimate.comp ∩ E.comp ; estimate.fail ← estimate.fail ∪ E.fail
37           estimate.disc ← estimate.disc ∪ E.disc ; estimate.part ← estimate.part ∪ E.part
38           if ∃r ∈ estimate.fail : r ∈ estimate.disc then estimate.fail ← estimate.fail \ {r}
39           if ∃r ∈ estimate.fail : r ∈ estimate.part then estimate.fail ← estimate.fail \ {r}
40           synchronized ← E.comp ; agreed ← V
```

6.2. *Présentation détaillée*

Nous présentons d'abord les événements, les variables globales et les messages nécessaires aux algorithmes. Ensuite, nous expliquons comment chaque procédure fonctionne et les modifications par rapport à l'original. Nous abordons enfin le problème de l'incohérence des détecteurs de partitions à travers quatre règles nouvelles et un scénario d'exécution.

Événements. $msuspect$ est généré par le détecteur de partitions. Il donne trois ensembles de processus : défaillances f_p, déconnexions d_p et partitions p_p. L'ensemble des proccessus corrects, connectés et accessibles est calculé à partir de ces ensembles : $reachable = \Pi \setminus (f_p \cup d_p \cup p_p)$. $msend$ et $mrecv$ sont utilisés pour envoyer et recevoir des messages de diffusion. En outre, $vchg$ sert à informer les applications de l'installation d'une nouvelle vue. Pour des raisons de simplicité, dans la présentation, nous ignorons les entrées ($join$) et les sorties ($leave$) volontaires de l'application.

Variables. $view$ représente une vue dans laquelle $view.id$ est l'identificateur de la vue et $view.comp$, $view.fail$, $view.disc$, $view.part$ est l'ensemble des processus membres du groupe, défaillants, vus déconnectés, et vus partitionnés respectivement. $cview$, ayant les mêmes champs que $view$, représente une vue complète. Normalement, elle a la même valeur que $view$, mais quand l'installation d'une vue complète ne satisfait pas la propriété sur la cohérence des vues (AGM3), le processus installe alors provisoirement une vue partielle de la vue complète. En outre, les quatre variables $reachable$, $failure$, $disconnect$, $reachable$ sont obtenues directement de la sortie du détecteur de partitions et ne contiennent pas toujours la même valeur que les compositions de la vue $view$ Enfin, $installed$ indique si la condition d'accord de groupe et de partition est atteinte, et $stable$ si la composition de la vue augmentée correspond bien aux ensembles fournis par le détecteur de partitions.

Pour la synchronisation des processus, $version$ est un tableau indexé par Π dans lequel pour chaque $q \in \Pi$, $version[q]$ est la dernière version de $AgreementPhase$ de q connue de p. Il est utilisé pour savoir si les processus sont dans la même phase $AgreementPhase$. $agreed$ est aussi un tableau indexé par Π utilisé pour savoir si les messages envoyés sont dans la même phase *EstimateExchangePhase*. $synchronized$ est un ensemble de processus déjà synchronisés avec p, c.-à-d. p a reçu leur réponse à sa demande de synchronisation.

Les dernières variables globales sont $symset$ et $ctbl$. $symset$ est un tableau indexé par Π dans lequel pour chaque $q \in \Pi$, $symset[q]$ contient la dernière valeur de $reachable$ tels que $q \notin reachable$ (cf. ligne 11 de l'algorithme 1). $ctbl$ est un tableau des enregistrements indexé par Π dont les champs de chaque enregistrement sont $cview$, $agreed$ et $estimate$. Pour chaque entrée q, $ctbl[q]$ représentant la perception par p de la valeur de ces variables de q est utilisé pour tester la condition d'accord par le coordinateur dans l'algorithme 3.

Types de message. $\langle \text{SYNCHRONIZE}, V_p, V_q, P \rangle$ permet de synchroniser les processus qui ne sont pas dans la même phase de l'algorithme. V_p et V_q sont les deux valeurs des numéros de version de l'émetteur et du récepteur respectivement ; P re-

présente la dernière valeur de $symset$ associée au processus destinataire de l'émetteur. $\langle \text{SYMMETRY}, V, P \rangle$ sert à résoudre le problème de l'asymétrie temporaire[3] entre les ensembles $reachable$ des différents processus. V est un tableau des numéros de version connus par l'émetteur et P est l'approximation de l'ensemble des processus accessibles connus par p. $\langle \text{ESTIMATE}, V, E \rangle$ contient l'estimation de chaque processus sur la composition de la nouvelle vue. Ce message est échangé entre les processus pour obtenir le consensus sur les ensembles des processus corrects accessibles, défaillants, déconnectés, et partitionnés. V est un tableau des versions et E est une estimation. $\langle \text{PROPOSE}, S \rangle$ transmet au coordinateur une proposition pour la nouvelle vue. Ce dernier s'appuie sur ces informations pour tester la condition d'accord de groupe. S est la valeur de *ctbl* qui représente une proposition au coordinateur. $\langle \text{VIEW}, w, C \rangle$ est un message diffusé aux membres du groupe pour annoncer l'installation d'une nouvelle vue. Ce message possède deux arguments : w est la valeur représentant l'identificateur de la vue et C est une copie de la variable $ctbl$.

Algorithmes. Dans l'algorithme 1, après l'initialisation des variables (lignes 2–4), p installe une vue initiale (ligne 5). Ensuite, chaque processus p entre dans une boucle infinie et attend des événements. Il est réactivé soit quand il reçoit un message de suspicion $msuspect$ de $\mathcal{PDG}$ soit quand il reçoit un message de synchronisation SYNCHRONIZE venant d'un autre processus q. Dans le premier cas (lignes 9–14), la variable $symset$ est mise à jour et p envoie un message SYMMETRY aux nouveaux processus accessibles pour corriger le problème de l'assymétrie. Puis, p entre dans la phase *AgreementPhase*. Dans l'autre cas, p entre dans la phase *AgreementPhase* si sa valeur de $version$ est moins élevée que celle du message en provenance de q pour éviter les messages obsolètes. Par rapport à l'original, ce qui est nouveau est l'apparition des ensembles des processus défaillants, vus déconnectés et vus partitionnés. La vue originale est remplacée dans notre algorithme par la vue augmentée.

L'algorithme 2 présente la structure de *AgreementPhase*. Outre l'apparition des nouveaux ensembles des processus défaillants, vus déconnectés et vus partitionnés, la modification se trouve dans la partie de traitement du message ESTIMATE, c.-à-d. quand les processus échangent leur estimation de la nouvelle vue augmentée. En échangeant les messages ESTIMATE, les processus mettent à jour leur propre estimation en utilisant les règles suivantes :

R1 : l'ensemble des processus vivants accessibles $estimate.comp$ est calculé comme l'intersection des ensembles. La taille de cet ensemble est alors diminuée de façon monotone : c'est une condition nécessaire pour assurer la terminaison de l'exécution de l'algorithme ;

R2 : l'ensemble des processus défaillants (resp. déconnectés ou partitionnés) est calculé comme l'union des ensembles.

3. Les ensembles $reachable$ obtenus sont temporairement asymétriques (p. ex. p considère que q est défaillant mais q considère que p est accessible). Dans cette situation, q pourrait être bloqué en attendant un accord avec p.

Algorithme 3: *EstimateExchangePhase*

1 **procedure** $EstimateExchangePhase()$
2 $installed \leftarrow false$
3 $InitializeEstimatePhase()$ % initialisation
4 **repeat**
5 **wait until** $event$
6 **case** $event$ **of**
7 $msuspect(f_p, d_p, p_p)$: % from $\mathcal{PDG}$
8 $P \leftarrow f_p \cup d_p \cup p_p$
9 **for all** $r \in (\Pi \setminus P) \setminus reachable$ **do** $symset[r] \leftarrow reachable$
10 $msend(\langle \text{SYMMETRY}, version, reachable\rangle, (\Pi \setminus P) \setminus reachable)$
11 $msend(\langle \text{ESTIMATE}, agreed, estimate\rangle, (\Pi \setminus P) \setminus reachable)$
12 $reachable \leftarrow \Pi \setminus P$
13 **if** $estimate.comp \cap P \neq \emptyset$ **then**$SendEstimate(estimate.comp \cap P, f_p, d_p, p_p)$
14 $mrecv(\langle \text{SYMMETRY}, V, P\rangle, q)$: % correct assymmetry problem
15 **if** $(agreed[p] = V[p]$ **or** $agreed[q] \leq V[q])$ **and** $(q \in estimate.comp)$ **then**
16 $SendEstimate(estimate.comp \cap P, estimate.comp \cap P, \emptyset, \emptyset)$
17 $mrecv(\langle \text{SYNCHRONIZE}, V_p, V_q, P\rangle, q)$: % synchronization request from q
18 $version[q] \leftarrow V_q$
19 **if** $(agreed[q] < V_q)$ **and** $(q \in estimate.reachable)$ **then**
20 $SendEstimate(estimate.comp \cap P, estimate.comp \cap P, \emptyset, \emptyset)$
21 $mrecv(\langle \text{ESTIMATE}, V, E\rangle, q)$: % estimation from q
22 **if** $(q \in estimate.comp)$ **then**
23 **if** $(p \notin E.comp)$ **and** $(agreed[p] = V[p]$ **or** $agreed[q] \leq V[q])$ **then**
24 $SendEstimate(estimate.comp \cap P.comp, P.fail, P.disc, P.part)$
25 **else if** $(p \in E.reachable)$ **and** $(\forall r \in estimate.comp \cap E.comp : agreed[r] = V[r])$
26 $SendEstimate(estimate.comp \setminus P.comp, P.fail, P.disc, P.part)$
27 $mrecv(\langle \text{PROPOSE}, S\rangle, q)$: % coordinator's work
28 $ctbl[q] \leftarrow S$
29 **if** $(q \in estimate.comp)$ **and** $CheckAgreement(ctbl)$ **then** % check agreement condition
30 $InstallView(UniqueID(), ctbl)$; $installed \leftarrow true$
31 $mrecv(\langle \text{VIEW}, w, C\rangle, q)$: % receive request to install a new view
32 **if** $(C[p].cview.id = cview.id)$ **and** $(q \in estimate.comp)$ **then**
33 $InstallView(w, C)$; $installed \leftarrow true$ % set terminate condition
34 **until** $installed$

Algorithme 4: Procédures et fonctions supplémentaires

1 **procedure** $InitializeEstimatePhase()$
2 $SendEstimate(\emptyset, \emptyset, \emptyset, \emptyset)$
3 **procedure** $SendEstimate(P, P_f, P_d, P_p)$
4 $estimate.comp \leftarrow estimate.comp \setminus P$; $estimate.fail \leftarrow estimate.fail \cup P_f$
5 $estimate.disc \leftarrow estimate.disc \cup P_d$; $estimate.part \leftarrow estimate.part \cup P_p$
6 **if** $\exists r \in estimate.fail : r \in estimate.disc$ **then** $estimate.fail \leftarrow estimate.fail \setminus \{r\}$
7 **if** $\exists r \in estimate.fail : r \in estimate.fail$ **then** $estimate.fail \leftarrow estimate.fail \setminus \{r\}$
8 $msend(\langle \text{ESTIMATE}, agreed, estimate\rangle, reachable \setminus \{p\})$
9 $msend(\langle \text{PROPOSE}, (cview, agreed, estimate)\rangle, Min(estimate.comp))$ % send a proposal to the coordinator
10 **function** $CheckAgreement(C)$
11 **return** $(\forall q \in C[p].estimate : C[p].estimate = C[q].estimate)$ **and**
12 $(\forall q, r \in C[p].estimate : C[p].agreed[r] = C[q].agreed[r])$
13 **procedure** $InstallView(w, C)$
14 $msend(\langle \text{VIEW}, w, C\rangle, C[p].estimate.comp \setminus \{p\})$
15 **if** $\exists q, r \in C[p].estimate.comp : q \in C[r].cview.comp \wedge C[q].cview.id \neq C[r].cview.id$ **then**
16 $view \leftarrow ((w, view.id), \{r | r \in C[p].estimate \wedge C[r].cview.id = cview.id\})$
17 **else** $view \leftarrow ((w, \bot), C[p].estimate)$
18 **generate** $vchg(view)$
19 $cview \leftarrow (w, C[p].estimate)$; $stable \leftarrow (view.comp = reachable)$ **and** $(\forall q, r \in C[p].estimate : C[p].agreed[r] = agreed[r])$

L'algorithme 3 présente la phase *EstimateExchangePhase* dans laquelle les processus essaient d'échanger leur estimation et d'obtenir un accord sur cette estimation en envoyant les messages ESTIMATE et PROPOSE (lignes 21 et 27). Il faut que tous les processus disposent de la même estimation et de la même version indiquée par la variable $agreed$. Cette condition est vérifiée dans la procédure *CheckAgreement(C)* de l'algorithme 4.

Dans l'algorithme 4, dans les trois premières procédures *InitializeEstimatePhase()*, *EstimateExchangePhase* et *CheckAgreementPhase(C)*, nous remplaçons la vue originale par la vue augmentée. La dernière procédure, *SendEstimate*(P, P_f, P_d, P_p) est modifiée pour assurer la convergence des ensembles de défaillances, de déconnexions et de partitions en plus de l'accord sur l'ensemble des processus vivants accessibles. Pour assurer la propriété d'intégrité des vues (AGM5), en plus des règles R1 et R2, nous proposons les règles suivantes permettant de passer un processus d'un ensemble à l'autre :

R3 : si un processus p voit un processus q à la fois dans son ensemble des processus défaillants et dans son ensemble des processus déconnectés alors p enlève q de celui des processus défaillants ;

R4 : si un processus p voit un processus q à la fois dans son ensemble des processus défaillants et dans son ensemble des processus partitionnés alors p enlève q de celui des processus défaillants.

Scénario d'exécution. Revenons à la situation présentée dans la section 2 dans laquelle les détecteurs de partitions ne donnent pas en sortie les même ensembles. Le message $msuspect$ contient par exemple les ensembles suivants :
- pour p : $f_p = \emptyset, d_p = \{r\}, p_p = \{s\}$; pour q : $f_q = \{s\}, d_q = \emptyset, p_q = \emptyset$.

En appliquant les règles R1–R2, nous obtenons alors :
- pour p : $f_p = \{s\}, d_p = \{r\}, p_p = \{s\}$; pour q : $f_q = \{s\}, d_q = \{r\}, p_q = \{s\}$.

En appliquant les règles R3–R4, nous obtenons enfin :
- pour p : $f_p = \emptyset, d_p = \{r\}, p_p = \{s\}$; pour q : $f_q = \emptyset, d_q = \{r\}, p_q = \{s\}$.

7. Conclusions et perspectives

Dans nos travaux précédents, nous avons montré comment et dans quelle mesure des intergiciels existants peuvent être enrichis par quatre détecteurs : défaillances, connectivité, déconnexions et partitions. En se basant sur ces détecteurs, nous proposons dans ce papier un nouveau service de gestion de groupes ayant de nouvelles propriétés intéressantes. Avec ce service, les processus se mettent d'accord non seulement sur l'ensemble des processus corrects connectés du groupe mais aussi sur les ensembles des processus défaillants, déconnectés ou partitionnés. La spécification est basée sur celle d'un système de communication de groupe existant [BAB 01] et le prototype est en cours de construction en utilisant le logiciel libre Jgroup [JGR 05] pour montrer la faisabilité de notre approche.

En guise de perspectives, nous devons compléter le développement du prototype. D'autre part, nous étudions également la possibilité de réaliser une implantation de

ce service de gestion de groupes sur les terminaux mobiles (p. ex. avec J2ME). Pour ce faire, il faut améliorer les algorithmes des détecteurs et de la gestion de groupes pour réduire leur complexité et leur nombre des messages envoyés. Cette amélioration est particulièrement importante en environnement mobile car les ressources des terminaux (p. ex. batterie, puissance de calcul) sont très limitées. En outre, il reste encore quelques limitations des détecteurs à lever (p. ex. les recouvrements des liens, des processus ou la mobilité des nœuds).

8. Bibliographie

[AGU 99] AGUILERA M., CHEN W., TOUEG S., « Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks », *Theoretical Computer Science*, vol. 220, n° 1, 1999, p. 3–30.

[ANC 95] ANCEAUME E., CHARRON-BOST B., MINET P., TOUEG S., « On the Formal Specification of Group Membership Services », rapport n° TR 95-1534, août 1995, Department of Computer Science, Cornell University, Ithaca, New-York (USA).

[BAB 01] BABAOĞLU Ö., DAVOLI R., MONTRESOR A., « Group communication in partitionable systems : specification and algorithms », *IEEE Transactions on Software Engineering*, vol. 27, n° 4, 2001, p. 308–336, IEEE Press.

[BHA 05] BHATTI M. U., CONAN D., « Détections de partition pour la gestion de groupes en environnement mobile », *Actes de Ubimob'05*, Grenoble, France, mai 2005, ACM Press.

[CHA 96a] CHANDRA T., HADZILACOS V., TOUEG S., CHARRON-BOST B., « On the Impossibility of Group Membership », *Proc. ACM PODC'96*, Philadeplhia, USA, mai 1996.

[CHA 96b] CHANDRA T. D., TOUEG S., « Unreliable Failure Detectors for Reliable Distributed Systems », *Journal of the ACM*, vol. 43, n° 2, 1996, p. 225–267, ACM Press.

[CHO 01] CHOCKLER G., KEIDAR I., VITENBERG R., « Group Communication Specifications : A Comprehensive Study », *ACM Comp. Surv.*, vol. 33, n° 4, 2001, p. 427–469, ACM Press.

[DEL 01] DELPORTE-GALLET C., « Sur l'algorithmique distribué tolérant aux pannes », décembre 2001, Thèse HDR, Université Paris VII.

[FIS 85] FISCHER M. J., LYNCH N. A., PATERSON M. S., « Impossibility of distributed consensus with one faulty process », *Journal of the ACM*, vol. 32, n° 2, 1985, p. 374–382, ACM Press.

[GÄR 99] GÄRTNER F. C., « Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments », *ACM Comp. Surv.*, vol. 31, n° 1, 1999, p. 1–26, ACM Press.

[JGR 05] JGROUP, « Jgroup home page », http://jgroup.sourceforge.net, 2005.

[MEN 03] MENA S., SCHIPER A., WOJCIECHOWSKI P., « A Step Towards a New Generation of Group Communication Systems », *Proc. of Middleware'03*, Springer Verlag, juin 2003.

[SAT 01] SATYANARAYANAN M., « Pervasive Computing : Vison and Challenges », *IEEE Personal Communications*, vol. 8, n° 4, 2001, p. 10–17, IEEE Press.

[TEM 04] TEMAL L., CONAN D., « Détections de défaillances, de connectivité et de déconnexions », *Actes de Ubimob'04*, Nice, France, juin 2004, ACM Press, p. 90–97.