Institut de la Francophonie pour l'Informatique
Institut National des Télécommunications

MÉMOIRE DE FIN D'ÉTUDES

MASTER D'INFORMATIQUE

Gestion de groupe et détection de partition en environnement mobile

NGUYEN Tuan Dung

Responsable de stage : Denis CONAN

Ce stage a été réalisé au sein du laboratoire **SAMOVAR**, équipe MARGE du département **Informatique** de l'**Institut National des Télécommunications**

GET/INT CNRS UMR SAMOVAR

Évry, 15 août 2005

Remerciements

Je voudrais tout d'abord remercier le Professeur Guy Bernard pour m'avoir accueilli dans son équipe de recherche de l'Institut National des Télécommunications (INT).

Je tiens à remercier tout particulièrement Denis Conan pour avoir proposé ce sujet de stage et m'avoir encadré pendant ces cinq mois. Je le remercie de son contact chaleureux, ses conseils et encouragements, son soutien permanent et la liberté de recherche qu'il a bien voulu me laisser. Qu'il trouve ici l'expression de ma profonde reconnaissance.

Mes plus sincères remerciements vont également à tous les professeurs et les personnels de l'Institut de la Francophonie pour l'Informatique (IFI) pour m'avoir donné des cours de très bonne qualité et pour leur soutien tout au long de mes études à l'IFI.

Un grand merci aux thésards et aux autres stagiaires dans l'équipe à l'INT pour une ambiance de travail particulièrement favorable.

Je remercie chaleureusement mes camarades de la promotion IX pour leur amitié sans faille et je leur souhaite bonne chance pour la soutenance.

Merci enfin à mes parents, mon frère et mes amis pour leur soutien et leur encouragement à tout l'instant.

Résumé

L'évolution des réseaux sans fil et des équipements a abouti à un nouveau paradigme connu sous le nom « informatique mobile » . Il offre aux utilisateurs la capacité de pouvoir se déplacer tout en restant connecté aux applications répartis et d'être indépendant de la localisation géographique. 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 groupe est un service principal des systèmes de communication de groupe, une brique importante pour construire des applications réparties.

Les travaux dans la littérature ont montré comment et dans quelle mesure des intergiciels existant peuvent être enrichis par quatre détecteurs : défaillance, connectivité, déconnexions, plus un quatrième, partitions. En se basant sur ces détecteurs, nous proposons un nouveau service de gestion de groupe 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. Un prototype est construit pour tester notre approche.

Mots-clefs : mobilité, tolérance aux fautes, algorithmique répartie, gestion de groupe et intergiciels.

Abstract

The advances in wireless networking and portable appliances have engendered a new paradigm of computing, called « mobile computing », in which users have access to data and information services regardless of their physical location or movement behavior. In mobile environment, the mobile terminals could frequently be disconnected. This characteristic requires the specific and adapted mechanisms of disconnection management and fault tolerance.

Group membership is a fundamental service of group communication systems, which are often used like the important bricks to build distributed applications.

Recent research works have showed that the existed middleware can be augmented with four detectors : failure, connectivity, disconnection and partition. Based on these detectors, we propose a new group membership service with new interesting properties. With this service, the processes reach an agreement not only about the set of correct connected processes but also about the set of failure, disconnected and partitioned processes. A prototype is developped to test our approach.

Keywords : mobility, fault tolerance, distributed algorithm, group membership, middleware.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Chapitre 1

Introduction

1.1 Problématique

L'apparition des ordinateurs portables, des assistants personnels numériques (PDA) puissants et des réseaux locaux sans fil (WLAN) dans les années 90s posent de nouveaux problèmes de construction des systèmes répartis avec les clients mobiles [25]. Ceci a engendré un nouveau paradigme appelé « informatique mobile » qui suscite des recherches non seulement pour résoudre les problèmes existants dans les systèmes répartis (p. ex. la tolérance aux fautes) mais aussi pour traiter les problèmes spécifiques en environnement mobile (p. ex. la gestion de déconnexions).

La tolérance aux fautes est un sujet de recherche important pour les systèmes répartis. Ces systèmes disposent de deux propriétés importantes : la sûreté et la vivacité. La première assure que les comportements mauvais n'apparaissent jamais dans le système et la seconde assure que les bons comportements doivent finalement apparaître lors de l'exécution du système. Dans les systèmes réels, la prioritaire entre ces deux propriétés dépend du type de l'application. Les combinaisons de ces deux propriétés nous donnent les quatre formes de tolérance aux fautes présentées dans le tableau 1.1.

	vivace	non vivace
sûre	masquant	fail safe
non sûre	non masquant	rien

TAB. 1.1 – Quatre formes de tolérance aux fautes

La redondance est indispensable pour la tolérance aux fautes. Chaque système réparti doit donc être redondant dans l'espace (p. ex serveurs primaires vs. secondaires) ou dans le temps

d'exécution (p. ex le calcul d'une variable plusieurs fois). Pour assurer la sûreté, il faut détecter les défaillances dans le système grâce aux mécanismes de détection. Par contre, la vivacité est plus difficile à garantir car nous devons corriger les erreurs détectés (la correction) [15].

Dans les systèmes purement asynchrones, c'est-à-dire dans lesquels il n'existe aucune borne sur la vitesse de calcul de chaque processus ou sur le temps de transmission des messages, [14] a montré qu'il est impossible de résoudre le problème du consensus entre processus répartis dès qu'un seul d'entre eux est défaillant (p. ex. par un arrêt franc). La raison est que nous ne pouvons pas dire si un processus est défaillant ou simplement très lent.

Les systèmes de communication de groupe (GCSs pour *Group Communication Systems* en anglais) sont considérés comme des briques fondamentales pour construire les applications réparties. Ils fournissent aux applications la capacité de communication de type *multicast* fiable. Mais le principe d'un GCS s'appuie sur un mécanisme de consensus et donc subit aussi un résultat d'impossibilité dans les systèmes asynchrones [9].

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 [10]. D'autres approches utilisent les algorithmes probabilistes. Grâce à ce modèle, nous pouvons ensuite dénombrer plusieurs résultats de recherche sur les GCSs. Mais, il est montré dans [3] que la plupart d'entre eux possèdent des faiblesses dans leur spécification : ces systèmes peuvent autoriser des exécutions indésirables ou leur spécification peuvent être facilement satisfaites par des solutions triviales sans utilité (p. ex. par les vues capricieuses[1]). La grande difficulté est alors de trouver une spécification à la fois assez forte pour éviter tous ces problèmes et assez faible pour qu'il soit faisable de trouver une solution.

Nous présentons maintenant les caractéristiques de l'environnement mobile et la problématique de notre recherche. Tout d'abord, l'informatique mobile est souvent caractérisée par quatre contraintes [24] :

- les entités mobiles ont des ressources très limitées par rapport aux entités fixes ;
- la mobilité contient elle-même plus de risques (p. ex. perte ou vol) ;

[1] La vue du processus p est le singleton $\{p\}$.

- la connectivité en environnement mobile est hautement variable en performance et en fiabilité ;
- les entités mobiles disposent d'une source limitée d'énergie.

Dans les environnements mobiles, les terminaux mobiles sont sujets à des déconnexions 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 caractéristique donne naissance à de nouveaux types de détecteurs comme les détecteurs de connectivité et de déconnexions [26].

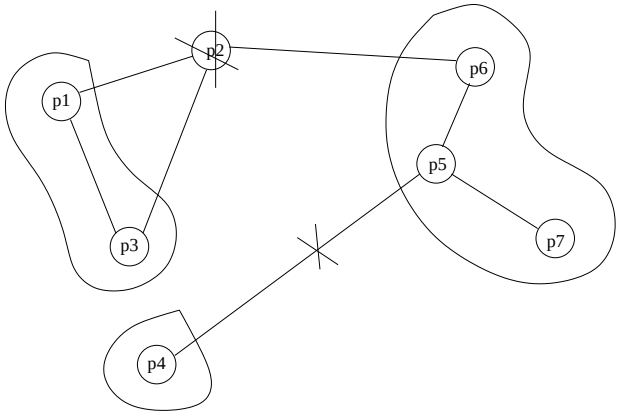

FIG. 1.1 – Partitionnement dans le réseau

Il existe des nœuds et des liens sensibles dans la topologie du réseau car les défaillances ou les déconnexions de ces entités 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 (cf. figure 1.1). Avec l'apparition du détecteur de partition [7], nous pouvons faire la distinction entre les défaillances, les déconnexions et les partitions.

Toutes ces propriétés extra-fonctionnelles requièrent donc l'adaptation des applications existantes. L'adaptation aux caractéristiques de l'environnement mobile peut être réalisée par les applications (l'approche *laissez faire*), par le système (l'approche *transparence aux applications*) ou par les deux (l'approche *collaboration*) [18]. La dernière approche dans laquelle l'application

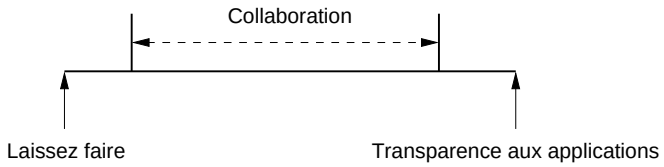

FIG. 1.2 – Stratégies de l'adaptation

décide de l'adaptation en se basant sur l'information fournie par l'intergiciel convient mieux aux environnements mobiles. Elle permet aux applications de déterminer la meilleure adaptation et préserve en même temps la capacité du système à surveiller les ressources [24]. Certaines fois, c'est l'intergiciel qui agit seul (il est dit proactif). Il peut y avoir une approche pessimiste et une approche optimiste pour la décision de l'application. Dans l'approche pessimiste, l'application continue à fonctionner en acceptant une dégradation de service. Dans l'approche optimiste, l'application peut donner des contres-mesures différentes : par exemple, attendre la reconnexion des processus ou encore l'ouverture de nouveaux liens [7].

1.2 Motivation et objectifs du stage

Ce travail de stage est la suite des travaux précédents sur les détecteurs en environnement mobile [26, 7] dans lequel nous essayons de construire une application concrète et prometteuse de ces nouveaux détecteurs.

Le premier objectif du stage est donc de construire un service de gestion de groupe de processus partitionables tolérant la mobilité des terminaux. Le second objectif est d'exploiter 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.

1.3 Environnement de travail

Le laboratoire SAMOVAR (*Services répartis Architectures, MOdélisation, Validation, Administration des Réseaux*) regroupe plusieurs enseignants-chercheurs de l'Institut National des Télécommunications (INT) travaillant dans le domaine des services, des réseaux et des télécommunications. L'unité est organisée autour de 4 équipes : ARMOR, AVERSE, MARGE et

CINA. Il est reconnu comme l'Unité Mixe de Recherche de CNRS labellisée n° 5157 depuis 2003.

Le département Informatique participe à SAMOVAR par le biais de l'équipe MARGE (*Middleware pour Applications Réparties avec Gestion de l'Environnement*) dans le domaine des systèmes distribués, avec une orientation vers la prise en compte de la mobilité.

Un des thèmes de recherche de l'équipe est l'adaptation dynamique aux caractéristiques des environnements mobiles. Dans ce travail, l'équipe s'intéresse à la détection de déconnexions, à la gestion de cache de composants logiciels et à la réconciliation pour la gestion de déconnexions. Ce travail de stage se situe dans cet axe de recherche.

1.4 Contribution

La principale contribution de ce travail est la réalisation d'un nouveau service de gestion de groupe avec de nouvelles propriétés grâce aux trois détecteurs utilisés dans l'environnement mobile. La spécification est basée sur celle d'un système de communication de groupe existant [5] et le prototype est développé en utilisant le système *Open Source* Jgroup [17].

1.5 Plan du document

La suite du rapport est organisée comme suit. Après l'étude bibliographique sur les détecteurs existants pour environnements mobiles dans le chapitre 2, dans le chapitre 3, nous présentons les concepts, les caractéristiques et une classification des systèmes de communication de groupes. Nous étudions plus en détail dans ce chapitre 3 deux implantations existantes en Java. Ensuite, dans le chapitre 4, nous proposons un nouveau service de gestion de groupe avec sa spécification formelle. L'architecture, l'algorithme et l'implantation d'un prototype de ce service sont présentés dans le chapitre 5. Enfin, le chapitre 6 donne la conclusion et les perspectives du travail. Les preuves des algorithmes présentés sont insérées en annexe.

Chapitre 2

Détecteurs en environnement mobile

Ce chapitre a pour but de faire une étude sur les différents détecteurs proposés en environnement mobile. Il commence par un rappel sur le modèle général de système réparti(cf. section 2.1). Ensuite, nous présentons les principes des quatres détecteurs pour environnements mobiles (dans les sections 2.2, 2.3, 2.4 et 2.5). La section 2.6 donne une discussion sur quelques limites de ces détecteurs et l'idée de les utiliser pour un service de gestion de groupe. Enfin, nous concluons le chapitre dans la section 2.7.

2.1 Modèle général

Le système consiste en un ensemble de n processus $\Pi = \{p_1, p_2, \ldots, p_n\}$ où chaque paire de processus est connectée par un lien de communication. Nous pouvons le modéliser par un graphe orienté $G = (\Pi, \Lambda)$ où $\Lambda \subset \Pi \times \Pi$ est l'ensemble de liens entre les processus.

Pour plus de clarté dans la présentation, nous considérons l'existence d'une horloge globale virtuelle $\mathcal{T}$, mais elle n'est pas accessible aux processus et elle prend ses valeurs dans l'ensemble des entiers naturels.

Un modèle plus détaillé augmenté par les détecteurs en environnement mobile est présenté plus tard dans la section 4.1.

2.2 Détecteur de défaillances non fiable

Pour résoudre le problème de l'impossibilité du consensus dans les systèmes asynchrones dans lesquels les liens entre processus sont fiables [14], [10] a proposé pour la première fois 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 actuellement suspectés d'être défaillants. Ce détecteur n'est pas fiable car il peut faire des erreurs. Quand il détecte que la suspicion d'un processus q est inexacte, il enlève q de sa liste de suspects. Grâce à l'apparition de ce détecteur, nous obtenons un modèle de système réparti partiellement synchrone et c'est le modèle qui est utilisé dans la suite de notre travail.

Le détecteur de défaillances dans [10] est présenté en donnant ses propriétés abstraites et sans aucune implantation spécifique. Cela rend cette approche indépendante des systèmes réels. Les deux propriétés importantes sont les suivantes : *complétude* et *précision*. La première assure la capacité à suspecter les processus défaillants et la seconde restreint les erreurs qu'un détecteur peut faire [10, 12, 23].

Nous avons deux possibilités pour la *complétude* et quatre pour la *précision* :

- *complétude forte* : chaque processus incorrect est suspecté par tous les processus corrects ;
- *complétude faible* : chaque processus incorrect est suspecté par au moins un processus correct ;
- *précision forte* : aucun processus n'est suspecté avant qu'il ne soit défaillant ;
- *précision faible* : au moins un processus correct n'est jamais suspecté ;
- *précision forte finale* : il existe un instant après lequel aucun processus correct n'est suspecté par les processus corrects ;
- *précision faible finale* : il existe un instant après lequel au moins un processus correct n'est jamais suspecté par les processus corrects.

En combinant chaque propriété de complétude avec chaque propriété de précision, nous obtenons huit classes de détecteurs possibles comme présenté dans le tableau 2.1. Grâce à ces détecteurs, [10] a montré que $\diamondsuit\mathcal{S}$ est le plus faible détecteur de défaillances permettant de résoudre le problème du consensus (avec une majorité de processus corrects).

	Précision			
Complétude	Forte	Faible	Forte finale	Faible finale
Forte	Parfait $\mathcal{P}$	Fort $\mathcal{S}$	Parfait Final $\diamondsuit\mathcal{P}$	Fort Final $\diamondsuit\mathcal{S}$
Faible	$\mathcal{Q}$	Faible $\mathcal{W}$	$\diamondsuit\mathcal{Q}$	Faible Final $\diamondsuit\mathcal{W}$

TAB. 2.1 – Détecteurs de défaillances

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) [1, 2], le plus faible détecteur de défaillances qui permet de résoudre le problème du consensus est $\Diamond\mathcal{P}$. 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 » noté $\mathcal{HB}$. 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 compteur d'un processus ne cesse d'augmenter tant qu'il n'est pas défaillant. C'est l'application qui décide quel processus est suspecté d'être défaillant grâce à cette information fournie par $\mathcal{HB}$.

Le détecteur de défaillances « battements de cœur » pour les réseaux partitionables $\mathcal{HBP}$ est proposé par [1]. Le compteur d'un processus q de p ne cesse d'augmenter tant que q est dans la partition de p.

La *complétude* et la *précision* s'exprime de la façon suivante :

- $\mathcal{HB}$-*complétude* : Pour tout processus correct p, les compteurs de battements de cœur de tous les processus n'étant pas dans la partition de p sont bornés ;
- $\mathcal{HB}$-*précision* :
  - Pour tout processus p, les compteurs de battements de cœur de tous les processus croissent de façon monotone ;
  - Pour tout processus correct p, les compteurs de battements de cœur de tous les processus dans la partition de p sont non bornés.

Le détecteur de défaillances pour réseaux partitionnables $\mathcal{HBDP}$ est présenté dans [6, 7] se base sur l'algorithme de $\mathcal{HBP}$ et ajoute en plus les informations sur la topologie du réseau en sauvegardant le chemin parcouru par le battement de cœur.

2.3 Détecteur de connectivité

Le détecteur de connectivité $\mathcal{CD}$ [26] est introduit pour traiter les problèmes spécifiques des déconnexions. L'idée est de surveiller les ressources locales (niveau de la batterie du terminal mobile ou pourcentage de la bande passante du réseau sans fil) pour anticiper les déconnexions.

Un algorithme basé sur un mécanisme d'hystérésis pour construire un détecteur de connectivité est introduit dans [26]. Dans la figure 2.1, quand le niveau augmente et est inférieur au seuil

[1] Les processus arrêtent finalement d'envoyer des messages concernant la transmission d'un message m.

lowUp (resp. *highUp*), le terminal mobile est déconnecté (resp. partiellement connecté). Quand le niveau de ressource diminue mais est plus élevé que la valeur *highDown* (resp. *lowDown*), le terminal mobile est connecté (resp. partiellement connecté). L'algorithme utilise deux doubles seuils pour lisser les variations du niveau de disponibilité d'une ressource et donc éviter l'effet *ping-pong*. Cet effet intervient si le niveau de disponibilité de la ressource oscille autour d'une valeur frontière entre deux modes de connectivité, provoquant des transferts d'états répétés [26].

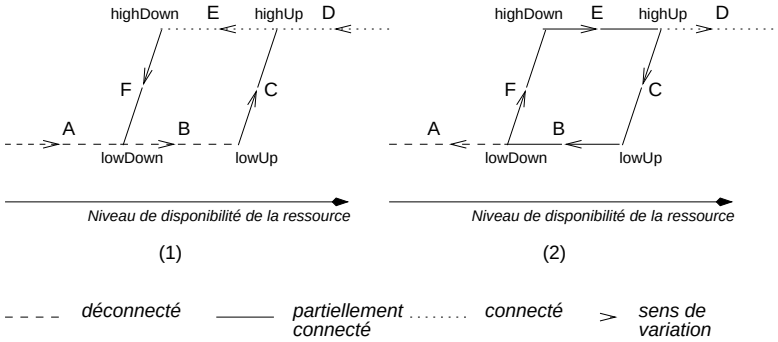

FIG. 2.1 – Hystérésis du gestionnaire de connectivité

2.4 Détecteur de déconnexions

Les informations de déconnexions et de reconnexions fournies par le détecteur de connectivité $\mathcal{CD}$ restent toutefois locales. Le détecteur de déconnexions $\mathcal{DD}$ [26] 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 peut être considérée comme défaillant.

Le détecteur de déconnexions pour réseaux partitionables $\mathcal{DDP}$ est proposé dans [7] comme une version étendue de $\mathcal{DD}$. Les changements apportés permettent de supporter les réseaux orientés non complets et les liens non fiables équitables.

Comme le détecteur de défaillances $\mathcal{FD}$, il est décrit de manière abstraite sans aucune implantation concrète. Nous avons aussi deux propriétés de *complétude de déconnexion* et de *précision de déconnexion*.

- *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 ;
- *précision de déconnexion forte* : aucun processus p n'est vu déconnecté par un processus correct connecté avant que p ne soit déconnecté.

2.5 Détecteur de partitions

La partition peut survenir suite à la défaillance d'un lien (cf. figure 2.2.a) ou d'un processus (cf. figure 2.2.b). Elle peut être aussi la conséquence d'une déconnexion d'un processus (cf. figure 2.2.c).

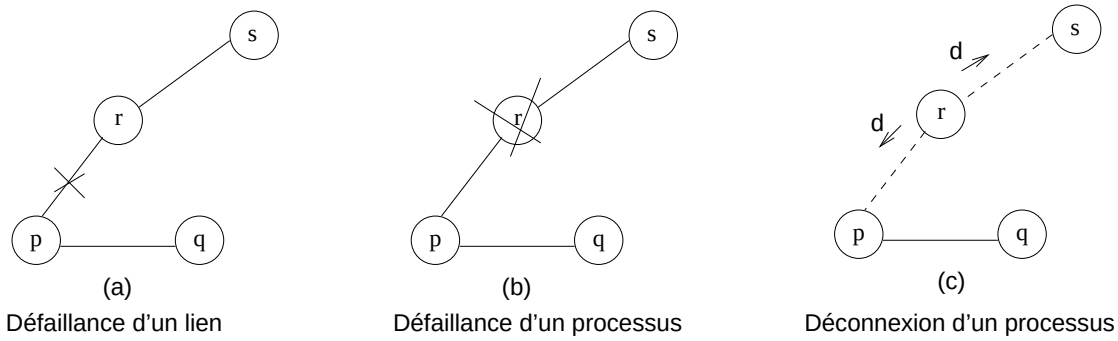

FIG. 2.2 – Situations de partitionnement

Comme les deux détecteurs précédents, le détecteur de partitions est présenté avec deux propriétés abstraites de complétude et de précision : *complétude de partition* et *précision de partition* [7].

- *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 ;
- *précision de partition forte finale* : il existe un instant après lequel aucun processus correct n'est vu partitionné avant qu'il ne soit partitionné.

Deux algorithmes sont proposés pour le détecteur de partitions : $\mathcal{PDG}$ et $\mathcal{PDN}$. $\mathcal{PDG}$ 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. Par contre, $\mathcal{PDN}$ ne requiert pas ces informations. Il n'utilise que les informations sur ses voisins fournies par le

détecteur de défaillances $\mathcal{HBDP}$. Cela explique la raison pour laquelle l'algorithme $\mathcal{PDN}$ est beaucoup plus compliqué que $\mathcal{PDG}$.

Dans ce travail, nous n'utilisons que l'algorithme $\mathcal{PDG}$. Donc, dans la suite de ce rapport, nous utilisons $\mathcal{PDG}$ au lieu du détecteur de partitions en général.

2.6 Discussion

La sortie du détecteur de partitions reste toutefois approximative et la cohérence entre différents processus n'est pas assurée. Dans la figure 2.2, nous pouvons constater plusieurs scénarios qui provoquent une partition. Les sorties des détecteurs de partitions des processus p et q, dans la situation (c), par exemple, peuvent être incohérentes comme dans le tableau 2.2. Dans ce cas, la déconnexion du processus r crée une situation de partitionnement. Le processus p détecte que q est déconnecté et alors s devient partitionné. Par contre, dans l'exemple du tableau 2.2, le processus q ne reçoit pas le message de déconnexion de p avant de considérer s défaillant.

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

TAB. 2.2 – Incohérence des détecteurs de partitions

Pour résoudre ce problème, nous construisons un service de gestion de groupe au dessus de $\mathcal{PDG}$ comme présenté dans la figure 2.3. 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 [7]. Ainsi, dans le scénario du tableau 2.2, p et q se mettent d'accord par exemple pour considérer r vu déconnecté et s vu partitionné.

D'autre part, ces détecteurs sont construits avec l'hypothèse que les processus et les liens défaillants ne sont pas sujets à reprise et que la topologie ne change pas de manière implicite. Cela veut dire que chaque processus se déconnecte et se reconnecte aux même voisins. La non-mobilité des nœuds est donc une limitation à lever, mais elle n'est pas dans le cadre de ce travail. En outre, excepté le détecteur de défaillances, les travaux existants sur les trois autres

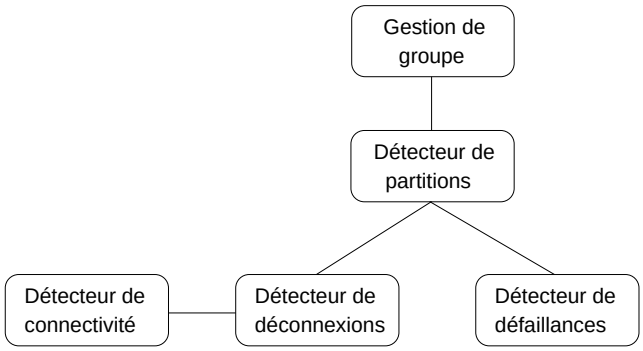

FIG. 2.3 – Détection de partitions pour la gestion de groupe

détecteurs de connectivité, de déconnexions et de partitions sont encore purement théoriques.

2.7 Conclusion

Dans ce chapitre, nous avons présenté les principes des détecteurs proposés en environnement mobile. Nous avons aussi discuté leur limitation (p. ex. l'incohérence entre différents processus) et introduit l'idée d'utiliser un nouveau service de gestion de groupe au dessus de ces détecteurs. Dans le chapitre suivant, nous allons présenter le concept et les propriétés du service de gestion de groupe avec les systèmes de communications de groupe existants dans la littérature.

Chapitre 3

Système de communication de groupe

Dans ce chapitre, nous présentons le concept (cf. section 3.1), la classification (cf. section 3.2) et les propriétés principales (cf. section 3.3) des systèmes de communication de groupe. Ensuite, nous étudions plus en détail deux implantations existantes et argumentons notre choix pour développer notre prototype (cf. section 3.4). Ce chapitre se termine par une conclusion dans la section 3.5.

3.1 Concept

Les systèmes de communication de groupe (GCSs) sont les briques fondamentales pour construire les applications réparties. Ils fournissent un mécanisme de communication de type *multicast* fiable. Nous pouvons les comparer avec d'autres protocoles existants (UDP, TCP, IP Multicast) dans le tableau 3.1 [16]. De nombreux types d'application des GCSs sont dénombrés dans la littérature [11] : les serveurs hautement disponibles, les plate-formes de travail collaboratif, les jeux multi-joueurs, etc.

	Non Fiable	Fiable
Unicast	UDP	TCP
Multicast	IP Multicast	**GCS**

TAB. 3.1 – Comparaison entre système de communication de groupe et autres protocoles

Dans ces systèmes, un groupe est un ensemble de processus qui coopèrent pour atteindre un objectif. Ils sont appelés les membres du groupe. Chaque groupe se distingue par son nom unique. Les processus dans un groupe communiquent en envoyant un message au nom du groupe ; le GCS s'occupe de transmettre le message vers tous les membres. Un processus peut

joindre un groupe existant. Il peut aussi 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 [11, 12] : le service de gestion de groupe et le service de diffusion de messages. Le premier service s'occupe de former et de maintenir la composition des groupes au cours de l'exécution du système. 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 groupe 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. Il est normalement construit en s'appuyant sur le premier, mais il y a aussi des travaux qui proposent un service de diffusion directement au dessus un détecteur de défaillance non fiable [13]. Dans le cadre de ce travail, nous ne nous intéressons qu'au premier service : la gestion de groupe. Plus de détails sur le deuxième service peuvent être consultés dans [11].

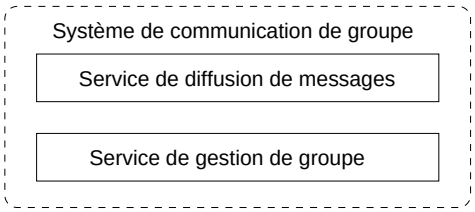

FIG. 3.1 – Architecture d'un système de communication de groupe

3.2 Classification

Dans la littérature, les GCSs sont souvent classés en deux catégories [11] : GCSs à composant primaire (p. ex. ISIS, Phoenix) et GCSs partitionables (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 partitionables autorisent la coexistence de plusieurs vues afin de modéliser les partitions dans le réseau.

Au point de vue de la structure des GCSs, nous les classons en deux types [20] : monolithique (p. ex. ISIS, Phoenix, Totem) et modulaire (p. ex. Ensemble, Javagroup, Jgroup). Les systèmes monolithiques ne permettent pas de s'adapter facilement aux besoins des utilisateurs.

Au contraire, les systèmes modulaires ayant une architecture en couches peuvent être plus faciles à modifier pour par exemple, ajouter de nouveaux composants.

La plupart des recherches concernent les réseaux d'entreprise (LAN), mais il y a aussi l'intérêt de construire des GCSs dans les réseaux à grande échelle (WAN) [19] ou en environnement mobile [22].

Si nous regardons plus spécialement l'algorithme de la gestion de groupe, nous pouvons distinguer les GCSs selon le type de l'algorithme (utilisant un coordinateur ou non, le nombre de messages envoyés avant sa terminaison ...).

Dans notre travail, nous cherchons une implantation d'un GCS partitionable et modulaire car il convient mieux aux caractéristiques de l'environnement mobile et c'est plus facile à l'adapter pour ajouter de nouvelles propriétés.

3.3 Propriétés

Les propriétés principales d'un service de gestion de groupe sont les suivantes [11, 12, 5] :

- auto-inclusion (*self inclusion*) : chaque vue installée par un processus p doit l'inclure lui même ;
- monotonie locale (*local monotonicity*) : si un processus p installe une vue V après avoir installé une autre vue V' alors l'identificateur de V doit être supérieur à celui de V' ;
- vue initiale (*initial view event*) : tous les événements (p. ex *send*, *recv*) doivent apparaître dans une vue ;
- complétude des vues (*view completeness*) : s'il existe un instant après lequel tout processus d'une partition Θ devient inaccessible à partir des autres processus du système alors ultimement la vue V de tous les processus corrects hors de Θ ne contient aucun processus de Θ ;
- précision des vues (*view accuracy*) : 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 de p.

3.4 Études de deux implantations existantes

Les GCSs ont attiré beaucoup de travaux de recherche [11] : ISIS, Horus, Totem, Transis, Ensemble, Javagroup, Relacs, Jgroup, Appia, ... Pour nous, le critère du choix d'un système pour le développement de notre prototype est tout d'abord la spécification et l'algorithme de gestion de groupe. Nous cherchons donc un GCS partitionable et modulaire dont la spécification ne subit pas de faiblesses déjà connues dans [3]. 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. Les trois candidats parmi les implantations plus récentes et encore actives sont donc Jgroup [17], Javagroup [16] et Appia [4]. Ils sont tous des GCSs partitionnables développés en Java avec la licence *Open Source*, mais Javagroup et Appia s'appuient sur un même système plus ancien développé en Caml (Ensemble). C'est la raison pour laquelle nous avons choisi le deux systèmes Jgroup et Javagroup pour notre étude ci-après.

3.4.1 Javagroup

Javagroup [16] est un GCS construit par un groupe de recherche à l'université de Cornell, aux États-Unis. Javagroup est développé purement en Java en se basant sur Ensemble (un GCS développé en Caml). L'algorithme de gestion de groupe de Javagroup se base sur l'algorithme publié dans [8]. C'est un GCS partitionable dont l'architecture modulaire est présentée dans la figure 3.2.

Au point de vue du développement d'applications, parmi les projets développés utilisant Javagroup, nous trouvons JBoss, JOnAS, TomcatHTTP. Il y a aussi le projet JGroups-ME qui essaie de fait porter Javagroup sur les téléphones mobiles et PDA en utilisant l'architecture J2ME [16].

Pourtant, Javagroup a une longue histoire de développement (Ensemble, Horus) donc il faut étudier aussi ces systèmes pour savoir comment il fonctionne. Le manque de documentation est une autre difficulté si nous choisissons ce système.

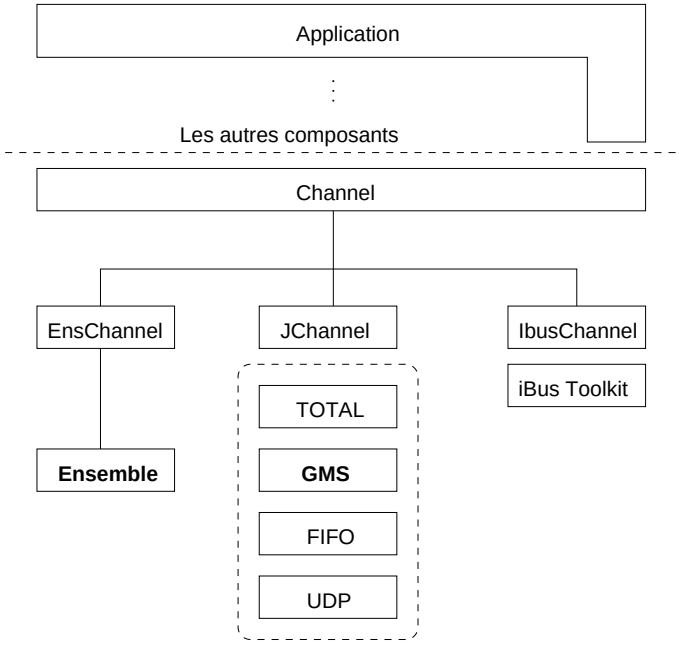

FIG. 3.2 – Architecture de Javagroup

3.4.2 Jgroup

Jgroup [17] est un résultat de recherche de l'université de Bologne, en Italie. Jgroup se base sur Relacs, un système construit précédemment par le même groupe de recherche. C'est un GCS partitionable ayant une spécification formelle assez claire [5]. Il est développé avec une structure modulaire dont l'architecture en couches est montrée dans la figure 3.3.

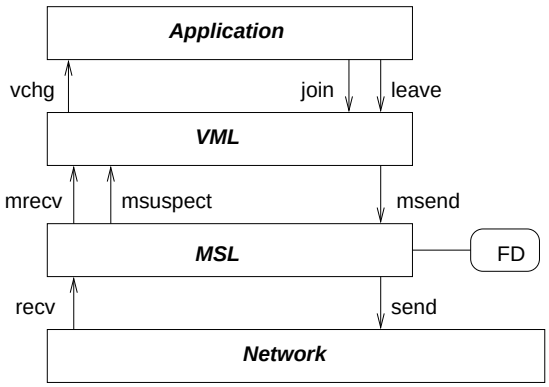

FIG. 3.3 – Architecture de Jgroup

Les deux couches principales sont les suivantes :

- MSL (pour *Multi Send Layer* en anglais) : il est difficile de construire un service de gestion de groupe directement au dessus d'un service de communication point-à-point non fiable fourni par le réseau. Cette couche réalise donc un protocole de communication un-vers-plusieurs de type « au mieux » (*best-effort* en anglais), c'est-à-dire qu'il n'y a aucune garantie sur la qualité de service. Elle filtre aussi la sortie du détecteur de défaillances $\mathcal{FD}$ pour enlever de la liste de suspects tous les processus accessibles de façon indirecte. Pour ce faire, MSL calcule la clôture transitive[1] des ensembles de processus accessibles.
- VML (pour *View Membership Layer* en anglais) : Cette couche utilise le service fourni par MSL pour construire et installer des vues. Ceci est fait grâce à un algorithme de gestion de groupe utilisant un coordinateur et ayant des mécanismes pour assurer sa terminaison dans n'importe quelle situation de défaillance de processus ou de perte de messages.

3.4.3 Choix de l'implantation

Nous constatons que Javagroup et Jgroup sont tous les deux GCSs partitionables ayant une architecture modulaire. Nous avons choisi ce dernier pour développer notre prototype pour les raisons suivantes :

- La documentation de Jgroup est plus détaillée ;
- L'architecture de Jgroup est plus facile à modifier et adapter à nos besoins ;

3.5 Conclusion

Ce chapitre a donné une présentation sur le concept, la spécification des systèmes de communication de groupe et les propriétés principales de son service de gestion de groupe. Nous avons fait une étude sur les deux implantations existantes pour choisir un système pour notre recherche après. Le chapitre suivant présente la spécification d'un nouveau service de gestion de groupe ayant de nouvelles propriétés grâce aux nouveaux détecteurs en environnement mobile présentés dans le chapitre 2.

[1] La clôture transitive d'un graphe (orienté) est un graphe ayant les même sommets et dont deux sommets sont reliés par une arête s'il y a un chemin les reliant dans le graphe de départ.

Chapitre 4

Spécification d'un nouveau service de gestion de groupe

Nous présentons dans ce chapitre la spécification formelle d'un service de gestion de groupe augmenté de nouvelles propriétés grâce aux détecteurs en environnement mobile. La présentation commence par un rappel sur le modèle de système réparti (cf. section 4.1), puis les historiques globaux (cf. section 4.2) et les schémas de défaillance (cf. section 4.3), de déconnexion (cf. section 4.4) et de partitionnement (cf. section 4.5). Ensuite, les propriétés du nouveau service de gestion de groupe sont présentées dans la section 4.6. Nous concluons ce chapitre dans la section 4.7.

4.1 Modèle de système réparti

Nous reprenons le modèle proposé dans [7]. 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 $\Pi = \{p_1, p_2, \ldots, p_n\}$ et un ensemble Λ de liens entre eux. Chaque paire de processus est connectée par un lien équitable non fiable, c'est-à-dire 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.

Les processus et les liens sont sujets à des défaillance franches et permanentes. Les pro-

cessus et les liens défaillants ne sont pas sujets à reprise. 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 ($p \nleftrightarrow q$).

Ce système est augmenté par les détecteurs de défaillances $\mathcal{FD}$, de déconnexions $\mathcal{DD}$ et de partitions $\mathcal{PD}$ définis ci-après.

4.2 Historiques globaux

Comme dans [5], 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'est-à-dire p n'exécute aucun événement à l'instant t.

4.3 Schéma de défaillance

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 $correct(F_P) = \Pi \backslash crashed(F_P)$ représente l'ensemble de processus corrects dans F_P.

Le schéma de défaillance F_L est une fonction de $\mathcal{T}$ vers 2^Λ où $F_L(t)$ est l'ensemble des liens défaillants jusqu'à l'instant t. $crashed(F_L) = \bigcup_{t \in \mathcal{T}} F_L(t)$ représente l'ensemble des liens défaillants dans F_L et $correct(F_L) = \Pi \backslash crashed(F_L)$ représente l'ensemble de 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 avec 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 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 [7].

Nous rappelons et définissons de manière formelle les propriétés des détecteurs de défaillances $\mathcal{HB}$:

- $\mathcal{HB}$-*complétude* : pour tout processus correct p, les compteurs de battements de cœur de tous les processus n'étant pas dans la partition de p sont bornés. Formellement :
  $\forall F = (F_P, F_L), \forall H \in \mathcal{HB}(F), \forall p \in \mathit{correct}(F_P), \forall q \in \Pi \setminus \mathit{partition}(p), \exists K \in \mathbb{N}, \forall t \in \mathcal{T} :$
  $H(p,t)[q] \leq K$
- $\mathcal{HB}$-*précision* :
  - pour tout processus p, les compteurs de battements de cœur de tous les processus croissent de façon monotone. Formellement :
    $\forall F, \forall H \in \mathcal{HB}(F), \forall p \in \Pi, \forall q \in \Pi, \forall t \in \mathcal{T} : H(p,t)[q] \leq H(p,t+1)[q]$
  - pour tout processus correct p, les compteurs de battements de cœur de tous les processus dans la partition de p sont non bornés. Formellement :
    $\forall F = (F_P, F_L), \forall H \in \mathcal{HB}(F), \forall q \in \mathit{partition}(p), \forall K \in \mathbb{N}, \forall t \in \mathcal{T} : H(p,t)[q] > K$

4.4 Schéma de déconnexion

Le schéma de déconnexion D est une fonction de $\mathcal{T}$ vers 2^{Π}, où $D(t)$ est l'ensemble des processus vus déconnectés à l'instant t. $\mathit{disconnected}(D)$ représente l'ensemble des processus vus déconnectés dans D, et $\mathit{connected}(D) = \Pi \setminus \mathit{disconnected}(D)$ représente les processus vus connectés. En outre, si $p \in \mathit{disconnected}(D)$, nous disons que p est vu déconnecté dans D, et si $p \in \mathit{connected}(D)$, 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 [26]). 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 détecteur 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 $\mathcal{DD}$ [7].

Nous rappelons et définissons de manière formelle les propriétés des détecteurs de déconnexions :

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

4.5 Schéma de partitionnement

La partition du processus p relativement à F et à D est dénommée $\mathit{partition}(p)$. Si p est défaillant ou déconnecté, l'ensemble $\mathit{partition}(p)$ est égal au singleton $\{p\}$. 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. La partition est créée à la suite soit d'une défaillance d'un lien ou d'un processus soit d'une déconnexion volontaire ou involontaire. Grâce au détecteur de partitions, nous pouvons distinguer l'ensemble de processus corrects connectés des trois ensembles des processus défaillants, déconnectés ou partitionnés [7].

Nous rappelons et définissons de manière formelle les propriétés des détecteurs de partitions :

- *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. 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 :

$\forall F = (F_P, F_L), \forall H_{FD} \in \mathcal{FD}(F), \forall H_{DD}, \forall H_{PD},$

$\exists t \in \mathcal{T}, \forall t' \geq t, \forall p, q \in correct(F_P), q \in partition(p) : q \notin H_{PD}(p, t')$

4.6 Propriétés du service de gestion de groupe

Dans la littérature, un service de gestion de groupe classique s'occupe de former et de maintenir une *vue*, c'est-à-dire une liste de processus corrects et connectés[1] appartenant à un groupe. Chaque changement dans la liste des membres du groupe lors de l'exécution 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 $V(p,t)$;
  - $\overline{V(p,t)}$ est l'ensemble des processus du groupe dans la vue $V(p,t)$.

Nous proposons dans ce travail un nouveau concept, c'est la « 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 du groupe dans la vue augmentée $V_a(p,t)$. Si un processus p est défaillant ou déconnecté, nous considérons que la vue du groupe $\overline{V_C}$ de p est égal au singleton $\{p\}$;
  - $\overline{V_F(p,t)}$ est l'ensemble des processus défaillants dans la vue augmentée $V_a(p,t)$;
  - $\overline{V_D(p,t)}$ est l'ensemble des processus vus déconnectés dans la vue augmentée $V_a(p,t)$;
  - $\overline{V_P(p,t)}$ est l'ensemble des processus vus partitionnés dans la vue augmentée $V_a(p,t)$.

L'ensemble des processus corrects 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*.

[1] 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.

Pour la gestion de groupe classique, il faut faire un consensus sur l'ensemble des processus appartenant au groupe (nous l'appelons « accord de groupe »). Dans notre travail, nous proposons un nouveau concept algorithmique que nous appelons « accord de partitions ». Il s'agit de mettre d'accord tous les processus d'un groupe, non seulement sur les processus corrects accessibles, mais aussi sur les processus défaillants, vus déconnectés et vus partitionnés.

Dans les cinq propriétés présentées ci-après, les modifications par rapport à [5] sont constatées dans AGM1, AGM2 et AGM5. Pour les deux premières propriétés sur la précision et la complétude des vues, nous remplaçons la vue dans le travail existant par la vue du groupe et ajoutons de nouvelles sous propriétés pour 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 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 vue 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 (cf. section 1.1). D'autre part, l'absence de AGM2 permet des GCSs dans lesquels les vues contiennent toujours tous les membres du groupe [5].

AGM3 Cohérence des vues

1. Si un processus correct p installe la vue 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 installe initialement la même vue v, puis p installe un successeur immédiat de v alors ultimement soit q installe aussi un successeur 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 w comme successeur immédiat de la vue 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 est une relation d'ordre partiel. Formellement : $v \prec^* w \Rightarrow w \not\prec^* v$

Avec les GCSs partitionables, il est impossible d'obtenir un ordre total sur l'installation des vues. La propriété AGM4 veut dire 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 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 :
   $\forall t \in \mathcal{T} : \overline{V_C(p,t)} \cap \overline{V_F(p,t)} \cap \overline{V_D(p,t)} \cap \overline{V_P(p,t)} = \emptyset$

La propriété AGM5 donne une contrainte sur l'intégrité des compositions des vues installées par un processus.

4.7 Conclusion

Nous avons présenté dans ce chapitre la spécification d'un nouveau service de gestion de groupe. Nous proposons le nouveaux concept de « vue augmentée » qui se compose de quatre ensembles de processus : la vue du groupe, la vue des défaillances, la vue des déconnexions et la vue des partitions. La spécification de ce service est basée sur celle de Jgroup [5] en ajoutant les nouvelles propriétés pour ces nouvelles compositions de la vue augmentée. Dans le chapitre suivant, nous présentons l'architecture, les algorithmes et l'implantation d'un prototype de ce service de gestion de groupe.

Chapitre 5

Architecture, algorithmes et prototype

Dans ce chapitre, nous présentons l'architecture (cf. section 5.1), les algorithmes (cf. section 5.2) et l'implantation d'un prototype (cf. section 5.3) du nouveau service de gestion de groupe. La présentation se termine par un état d'avancement du développement du prototype dans la section 5.4.

5.1 Architecture

L'architecture de notre service de gestion de groupe est illustrée dans figure 5.1. Par rapport de celle de Jgroup, nous insérons une couche intermédiaire entre les deux couches existantes MSL et VML. Cette couche réalise les fonctionnalités des trois nouveaux détecteurs de défaillances, de déconnexions et de partitions. Elle intercepte le message *msuspect* généré par MSL et fournit un nouveau message *msuspect** à VML. Par rapport à l'original, *msuspect** contient deux ensembles en plus : celui des processus déconnectés et celui des processus partitionnés.

Cependant, les travaux existants sur les détecteurs en environnement mobile, excepté le détecteur de défaillances, sont purement théoriques. Il n'existe pas encore d'implantation de ces détecteurs et donc il faut compter du temps pour réaliser ce travail. Ainsi, pour simplifier le travail de l'implantation, nous utilisons une simulation au lieu d'une vraie implantation des détecteurs $\mathcal{CD}$, $\mathcal{DD}$, $\mathcal{PDG}$. Cette simulation est réalisée d'une manière simple avec le schéma client-serveur. $\mathcal{PDG}$ écoute sur une porte et l'utilisateur lui donne l'ensemble des processus déconnectés et partitionnés selon un scénario prédéfini. Ce composant peut être facilement remplacé par une implantation réelle de ces détecteurs dans un travail à venir. L'objectif ici est de concevoir le fonctionnement du nouveau service de gestion groupe implémenté dans VML.

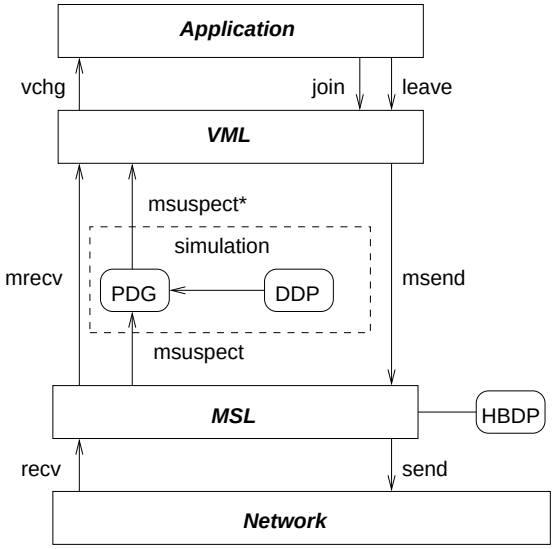

FIG. 5.1 – Architecture du système proposé

Dans la couche VML, l'algorithme pour la gestion de groupe est modifiée selon les modifications présentées dans la section suivante.

5.2 Algorithmes

Dans cette section, nous présentons tout d'abord l'idée principale et la structure générale de l'algorithme de gestion de groupe qui se base sur celui de la couche VML de Jgroup [5]. 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 [5].

5.2.1 Présentation générale

La structure générale de l'algorithme se trouve dans l'algorithme 1 qui est déclenché par les événements générés par MSL (p. ex *msuspect*, *mrecv*). Chaque processus entre dans la procédure *AgreementPhase* (cf. algorithme 2). Cette procédure se compose de deux sous-procédures : *SynchronisationPhase* et *EstimateExchangePhase* (cf. figure 5.2). La première a pour but de réaliser une synchronisation avec les autres 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). Le principe de cet algorithme est l'utilisation d'un coordinateur accessible choisi de façon fixe pour mettre d'accord les processus.

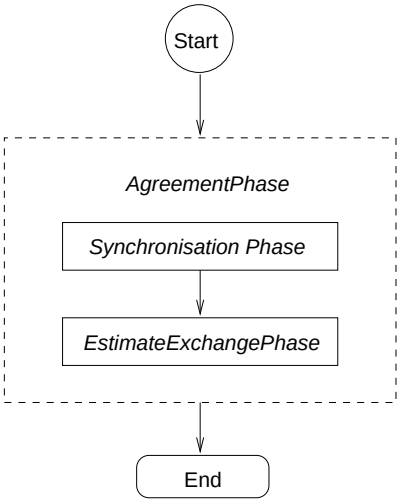

FIG. 5.2 – Algorithme de gestion de groupe

L'algorithme 4 donne quelques procédures et fonctions supplémentaires utilisées dans les algorithmes : $InitiateEstimatePhase$, $SendEstimate$, $CheckAgreement$ et $InstallView$:

- $InitiateEstimatePhase$ 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.

5.2.2 Présentation détaillée

Dans la première partie, nous présentons les primitives, les variables et les messages nécessaires aux algorithmes. Ensuite, nous expliquons comment chaque procédure fonctionne et les modifications par rapport à l'original. Nous abordons aussi le problème de l'incohérence avec les détecteurs de partitions (c.f section 2.6) pour montrer comment il est résolu avec notre algorithme.

PRIMITIVES

msuspect est généré par la couche en dessous (dans notre architecture, c'est le détecteur de partition $\mathcal{PDG}$). Il donne trois ensembles de processus de suspect : défaillances f_p, déconnexions d_p et partitions p_p. L'ensemble des proccessus corrects, connectés et accessibles peut être calculé à partir de ces ensembles ($reachable = \Pi \setminus (f_p \cup d_p \cup p_p)$).

msend est implémenté par MSL et utilisé pour envoyer un message m à un ensemble de processus destinataires (de type *best-effort*).

mrecv est généré par MSL quand il reçoit un message m (envoyé par *msend*) à partir d'un autre processus.

vchg est généré dans l'algorithme de VML pour informer les applications de l'installation d'une nouvelle vue.

VARIABLES

view représente une vue dans laquelle :

- $view.id$ est l'identificateur de la vue ;
- $view.comp$ est l'ensemble des processus membres du groupe ;
- $view.fail$ est l'ensemble des processus défaillants ;
- $view.disc$ est l'ensemble des processus vus déconnectés par les membres du groupe ;
- $view.part$ est l'ensemble des processus vus partitionnés par les membres du groupe.

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 une vue partielle de la vue complète. Donc, nous pouvons considérer que *cview* est une tentative de vue et *view* est la vue effectivement installée.

$failure$ représente l'ensemble des processus défaillants.

disconnect représente l'ensemble des processus vus déconnectés.

partition représente l'ensemble des processus vus partitionnés.

reachable représente l'ensemble des processus vus corrects, connectés et accessibles.

Ces quatre variables ($reachable$, $failure$, $disconnect$, $reachable$) sont obtenues directement de la sortie du détecteur de partition $\mathcal{PDG}$ et ne contiennent pas toujours la même valeur que les compositions de la vue ($view.comp$, $view.fail$, $view.disc$, $view.part$) qui sont obtenues après un accord avec les autres 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 et $version[p]$ est la version actuelle de p. Elle est utilisée 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 $EstimateExchangePhase$.

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. lignes 18–19 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, $ctbl[q]$ représente la perception par p de la valeur de ces variables de q. Ce tableau est utilisé pour tester la condition d'accord par le coordinateur.

synchronized est un ensemble de processus déjà synchronisés avec p, c'est-à-dire p a reçu leur réponse à sa demande de synchronisation.

stable est une variable booléenne qui indique si la composition de la vue complète correspond bien aux ensembles fournis par la couche en dessous ($\mathcal{PDG}$).

installed est une variable booléenne qui indique si la condition d'accord est atteinte.

Types de message

Lors de l'exécution de l'algorithme, plusieurs messages typés sont échangés. Chaque type de message dispose alors de différents composants. Voici ces différents messages :

$\langle \text{SYNCHRONIZE}, V_p, V_q, P \rangle$ est un message de synchronisation utilisé dans le début de l'algorithme 1) et dans $AgreementPhase$ (cf. algorithmes 2 et 3). Son but est de synchroniser les processus qui ne sont pas dans la même phase de l'algorithme. Ce message possède trois arguments : V_p, V_q sont les deux valeurs des numéros de version de l'émetteur et du récepteur respectivement ; P représente la dernière valeur de *symset* associée au processus destinataire de l'émetteur .

$\langle \text{SYMMETRY}, V, P \rangle$ est un message utilisé au début de l'algorithme 1 et dans la procédure $AgreementPhase$ (cf. algorithmes 2 et 3). L'objectif est de résoudre le problème de l'asymétrie temporaire[1] entre les ensembles *reachable* des différents processus. Ce message possède deux arguments : V est un tableau des numéros de version connu par l'émetteur et P est

[1] 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 l'accord avec p.

l'approximation de l'ensemble des processus accessibles connu par p à cet instant.

$\langle \text{ESTIMATE}, V, E \rangle$ est un message utilisé dans $AgreementPhase$ (cf. algorithmes 2, 3 et 4) qui contient l'estimation propre de chaque processus pour la composition de la nouvelle vue. Ce message est échangé entre les processus pour obtenir deux consensus sur l'ensemble des processus corrects accessibles et sur l'ensemble des processus défaillants, déconnectés et partitionnés. Ce message possède deux arguments : V est un tableau des versions et E est une estimation.

$\langle \text{PROPOSE}, S \rangle$ est un message utilisé dans $AgreementPhase$ (cf. algorithmes 3 et 4) qui contient une proposition de la nouvelle vue envoyée au coordinateur. Ce dernier s'appuie sur ces informations pour tester la condition d'accord. Ce message possède un seul argument : 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*.

Dans l'algorithme 1, après l'initialisation des variables (lignes 1–9), p installe une vue initiale (ligne 10). Ensuite, chaque processus p entre dans une boucle infinie et attend des événements. Il est reactivé 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 16–25), la variable *symset* est mise à jour et p envoie un message SYMMETRY aux nouveaux processus accessibles pour corriger le problème de l'asymétrie. 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.

Algorithme 1: Algorithme principal d'un processus p

```
1   initialisation :
2       reachable ← {p}
3       failure ← ∅
4       disconnect ← ∅
5       partition ← ∅
6       version ← (0, ..., 0)
7       symset ← ({p}, ..., {p})
8       view ← (UniqueID(), ({p}, ∅, ∅, ∅))                    % Initial view
9       cview ← view
10      generate vchg(view)                                    % Install the initial view
11
12      while true do
13          wait until event
14          case event of
15
16              msuspect(f_p, d_p, p_p) :                      % From PDG
17                  P ← f_p ∪ d_p ∪ p_p
18                  for all r ∈ (Π \ P) \ reachable do
19                      symset[r] ← reachable                  % new reachable process
20                  msend(⟨SYNCHRONIZE, version, reachable⟩, (Π \ P) \ reachable)
21                  reachable ← Π \ P                          % update relevant sets
22                  failure ← f_p
23                  disconnect ← d_p
24                  partition ← p_p
25                  AgreementPhase()
26
27              mrecv(⟨SYNCHRONIZE, V_p, V_q, P⟩, q) :         % Synchronization request from q
28                  % avoid obsolete messages
29                  if version[q] < V_q then
30                      version[q] ← V_q
31                      if q ∈ reachable then AgreementPhase()
32
```

L'algorithme 2 présente la structure de *AgreementPhase*. Chaque processus entre dans cette phase avec sa propre estimation des ensembles corrects et accessibles, défaillants, déconnectés, et partitionnés. Il y a deux sous procédures : la procédure *SynchronisationPhase* est présentée dans l'algorithme 2 et la procédure *EstimateExchangePhase* est donnée plus tard dans l'algorithme 3.

Au début de *SynchronisationPhase*, p réalise une demande de synchronisation avec tous les processus accessibles dans son estimation en envoyant les messages SYNCHRONIZE (lignes 10–12). Ensuite, p entre dans une boucle et attend jusqu'à l'obtention de cette synchronisation (ligne 15). p quitte cette procédure soit quand il reçoit toutes les réponses des processus accessibles dans son estimation (ligne 37) soit quand il reçoit un message ESTIMATE d'un processus qui est déjà dans la phase *EstimateExchangePhase* (ligne 45). Dans les deux cas, p met à jour la variable *agreed* pour l'utiliser dans la phase suivante. Pendant l'attente, il peut aussi traiter la primitive *msuspect* générée par $\mathcal{PDG}$ ou le message SYNCHRONIZE venant d'un autre processus. 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'est-à-dire 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 accessible *estimate.comp* est calculé comme l'intersection de ces ensembles des membres du groupe. La taille de cet ensemble est alors diminuée de façon monotone et c'est une condition nécessaire pour assurer la terminaison de l'exécution de l'algorithme ;
- **R2**. L'ensemble des processus défaillants (déconnectés ou partitionnés) est calculé comme l'union de ces ensembles des membres du groupe.

Pour assurer la propriété d'intégrité des vues (AGM5), nous proposons les règles pour passer un processus d'un ensemble à l'autre afin d'assurer la convergence. Ces règles de convergence (R3–R4) sont données avec la description de l'algorithme 4.

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
9   procedure SynchronisationPhase()
10      synchronized ← {p}                                            % set of synchronized processus
11      % send synchronization request
12      for all r ∈ estimate.comp \ {p} do
13          msend(⟨SYNCHRONIZE, version[r], version[p], symset[r]⟩, {r})
14
15
16      % wait for synchronization with others
17      while (estimate.comp ⊈ synchronized) do
18          wait until event
19          case event of
20
21              msuspect(f_p, d_p, p_p) :                             % From PDG
22                  P ← f_p ∪ d_p ∪ p_p
23                  for all r ∈ (Π \ P) \ reachable do
24                      symset[r] ← reachable
25                  msend(⟨SYMMETRY, version, reachable⟩, (Π \ P) \ reachable)
26                  reachable ← Π \ P
27                  failure ← f_p
28                  disconnect ← d_p
29                  partition ← p_p
30                  estimate.comp ← estimate.comp ∩ reachable
31                  estimate.fail ← estimate.fail ∪ failure
32                  estimate.disc ← estimate.disc ∪ disconnect
33                  estimate.part ← estimate.part ∪ partition
34
35              mrecv(⟨SYMMETRY, V, P⟩, q) :                          % correct asymmetry problem
36                  if (version[p] = V[p]) and (q ∈ estimate.comp) then
37                      estimate.comp ← estimate.comp \ P
38
39              mrecv(⟨SYNCHRONIZE, V_p, Vq, P⟩, q) :                 % synchronization request from q
40                  % avoid obsolete messages and send back response
41                  if version[p] = V_p then
42                      synchronized ← synchronized ∪ {q}
43                  if version[q] < V_q then
44                      version[q] ← V_q
45                      agreed[q] ← V_q
46                      msend(⟨SYNCHRONIZE, version[q], version[p], symset[q]⟩, {q})
47
48              mrecv(⟨ESTIMATE, V, E⟩, q)                            % Estimation from q
49                  version[q] = V[q]
50                  % correct asymmetry problem
51                  if q ∉ estimate.comp then
52                      msend(⟨SYMMETRY, version, estimate.comp⟩, {q})
53                  % update the own estimation
54                  elseif (version[p] = V[p]) and (p ∈ E.comp) then
55                      estimate.comp ← estimate.comp ∩ E.comp
56                      estimate.fail ← estimate.fail ∪ E.fail
57                      estimate.disc ← estimate.disc ∪ E.disc
58                      estimate.part ← estimate.part ∪ E.part
59                      if ∃r ∈ estimate.fail : r ∈ estimate.disc then
60                          estimate.fail ← estimate.fail \ {r}
61                      if ∃r ∈ estimate.fail : r ∈ estimate.part then
62                          estimate.fail ← estimate.fail \ {r}
63                      synchronized ← E.comp
64                      agreed ← V
```

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 9–32). L'accord est décidé par un coordinateur choisi entre les processus accessibles dans l'estimation (p.ex. par celui qui possède le plus petit index). 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. Quand elle est satisfaite, le coordinateur envoie un message VIEW aux autres processus de l'ensemble de processus accessibles de l'estimation. Quand un processus reçoit ce dernier message, il installe la nouvelle vue et retransmet le message aux autres processus de la vue. Cela est nécessaire car le coordinateur peut aussi être défaillant ou déconnecté avant d'avoir transmis tous les messages.

Une autre modification par rapport à l'original est l'apparition des trois ensembles de défaillances, de déconnexions et de partitions. La procédure *SendEstimate* 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.

Algorithme 3: *EstimateExchangePhase*

```
1   procedure EstimateExchangePhase()
2   installed ← false
3   InitializeEstimatePhase()                                   % Initialisation
4
5   repeat
6       wait until event
7       case event of
8
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
12                  symset[r] ← reachable
13              msend(⟨SYMMETRY, version, reachable⟩, (Π \ P) \ reachable)
14              msend(⟨ESTIMATE, agreed, estimate⟩, (Π \ P) \ reachable)
15              reachable ← Π \ P
16              if estimate.comp ∩ P ≠ ∅ then
17                  SendEstimate(estimate.comp ∩ P, f_p, d_p, p_p)
18
19          mrecv(⟨SYMMETRY, V, P⟩, q) :                        % correct asymmetry problem
20              if (agreed[p] = V[p] or agreed[q] ≤ V[q]) and (q ∈ estimate.comp) then
21                  SendEstimate(estimate.comp ∩ P, estimate.comp ∩ P, ∅, ∅)
22
23          mrecv(⟨SYNCHRONIZE, V_p, V_q, P⟩, q) :              % synchronization request from q
24              version[q] ← V_q
25              if (agreed[q] < V_q) and (q ∈ estimate.reachable) then
26                  SendEstimate(estimate.comp ∩ P, estimate.comp ∩ P, ∅, ∅)
27
28          mrecv(⟨ESTIMATE, V, E⟩, q) :                        % Estimation from q
29              if (q ∈ estimate.comp) then
30                  if (p ∉ E.comp) and (agreed[p] = V[p] or agreed[q] ≤ V[q]) then
31                      SendEstimate(estimate.comp ∩ P.comp, P.fail, P.disc, P.part)
32                  elseif (p ∈ E.reachable) and (∀r ∈ estimate.comp ∩ E.comp : agreed[r] =
                    V[r]) then
33                      SendEstimate(estimate.comp \ P.comp, P.fail, P.disc, P.part)
34
35          mrecv(⟨PROPOSE, S⟩, q) :                            % coordinator's work
36              ctbl[q] ← S
37              % check agreement condition
38              if (q ∈ estimate.comp) and CheckAgreement(ctbl) then
39                  InstallView(UniqueID(), ctbl)
40                  installed ← true
41
42          mrecv(⟨VIEW, w, C⟩, q) :                            % receive request to install a new view
43              if (C[p].cview.id = cview.id) and (q ∈ estimate.comp) then
44                  InstallView(w, C)
45                  % set terminate condition
46                  installed ← true
47
48  until installed
```

Dans l'algorithme 4, nous trouvons trois procédures et une fonction utilisées dans l'algorithme de gestion de groupe :

- *InitializeEstimatePhase*() est utilisée pour envoyer une estimation initiale au début de la procédure *EstimateExchangePhase* ;
- *CheckAgreementPhase*(C) est utilisée pour tester la condition d'accord. Il faut que tous les membres aient la même estimation et soit dans la même phase de l'algorithme

(c'est-à-dire ils ont la même valeur de $agreed$) ;

- $InstallView(w, C)$ est utilisée pour installer une nouvelle vue ;
- $SendEstimate(P, P_f, P_d, P_p)$ est utilisée pour mettre à jour l'estimation. Ensuite, cette estimation est envoyée aux autres processus dans l'estimation de la vue de groupe et proposée au coordinateur.

Les trois premières procédures utilise le même principe que l'original. Nous devons seulement remplacer la vue originale par la vue augmentée. Dans la dernière procédure, nous utilisons les règles R1–R4 pour mettre à jour les ensembles dans l'estimation et assurer la convergence. Les deux premières règles R1–R2 ont été décrites dans la description de l'algorithme 2.

Les deux dernières règles R3–R4 sont construites en se basant sur le principe des détecteurs de déconnexions et de partitions (cf. sections 2.4 et 2.5). Si les messages de déconnexion et de réconnexion venant d'un processus sont perdus, il serais considéré par les autres processus comme être défaillant. Les règles suivantes sont proposées pour traiter ce problème avec la gestion de groupe.

- **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 défaillances ;
- **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 défaillances.

Revenons à la situation présentée dans la section 2.6. À cause du problème de l'incohérence entre les différents processus, les détecteurs de partitions $\mathcal{PDG}$ ne donnent pas en sortie les même ensembles de suspect. 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 alors :

- pour p : $f_p = \emptyset, d_p = \{r\}, p_p = \{s\}$;

– pour q : $f_q = \emptyset, d_q = \{r\}, p_q = \{s\}$.

Nous voyons que grâce à ces règles, nous obtenons un consensus entre les membres de différents ensembles de processus en assurant la convergence de ces ensembles. Cela est nécessaire pour satisfaire la propriété AGM5 sur l'intégrité des vues.

Algorithme 4: Procédures et fonctions supplémentaires

```
1   procedure InitializeEstimatePhase()
2   SendEstimate(∅, ∅, ∅, ∅)
3
4   procedure SendEstimate(P, P_f, P_d, P_p)
5   estimate.comp ← estimate.comp \ P
6   estimate.fail ← estimate.fail ∪ P_f
7   estimate.disc ← estimate.disc ∪ P_d
8   estimate.part ← estimate.part ∪ P_p
9   if ∃r ∈ estimate.fail : r ∈ estimate.disc then
10      estimate.fail ← estimate.fail \ {r}
11  if ∃r ∈ estimate.fail : r ∈ estimate.fail then
12      estimate.fail ← estimate.fail \ {r}
13  msend(⟨ESTIMATE, agreed, estimate⟩, reachable \ {p})
14  msend(⟨PROPOSE, (cview, agreed, estimate)⟩, Min(estimate.comp)) % send a proposal to
    the coordinator
15
16  function CheckAgreement(C)
17  return (∀q ∈ C[p].estimate : C[p].estimate = C[q].estimate) and (∀q, r ∈ C[p].estimate :
    C[p].agreed[r] = C[q].agreed[r])
18
19  procedure InstallView(w, C)
20  msend(⟨VIEW, w, C⟩, C[p].estimate.comp \ {p})
21  if ∃q, r ∈ C[p].estimate.comp : q ∈ C[r].cview.comp ∧ C[q].cview.id ≠ C[r].cview.id) then
22      view ← ((w, view.id), {r|r ∈ C[p].estimate ∧ C[r].cview.id = cview.id})
23  else
24      view ← ((w, ⊥), C[p].estimate)
25  generate vchg(view)
26  cview ← (w, C[p].estimate)
27  stable ← (view.comp = reachable) and (∀q, r ∈ C[p].estimate : C[p].agreed[r] = agreed[r])
```

5.3 Implantation

L'implantation est réalisée en se basant sur le code source en Java de Jgroup version 2.0 [17]. Les tâches à réaliser sont les suivantes :

– **T1**. Insérer une couche intermédiare entre MSL et VML :

  – Intercepter les messages *msuspect* de MSL ;

  – Créer une simulation de PDG et transmettre les nouveaux messages *msuspect* à VML ;

– **T2**. Modifier l'implantation de l'algorithme de gestion de groupe VML.

Il y a plusieurs paquetages dans le code source de Jgroup mais nous nous intéressons au deux paquetages suivants :

– *jgroup.relacs.mss* dans lequel sont implémentés le détecteur de défaillances et la couche

MSL ;

– *jgroup.relacs.daemon* dans lequel est implémentée la couche VML.

Nous ajoutons un paquetage *jgroup.relacs.pdg* qui correspond à la couche intermédiaire $\mathcal{PDG}$.

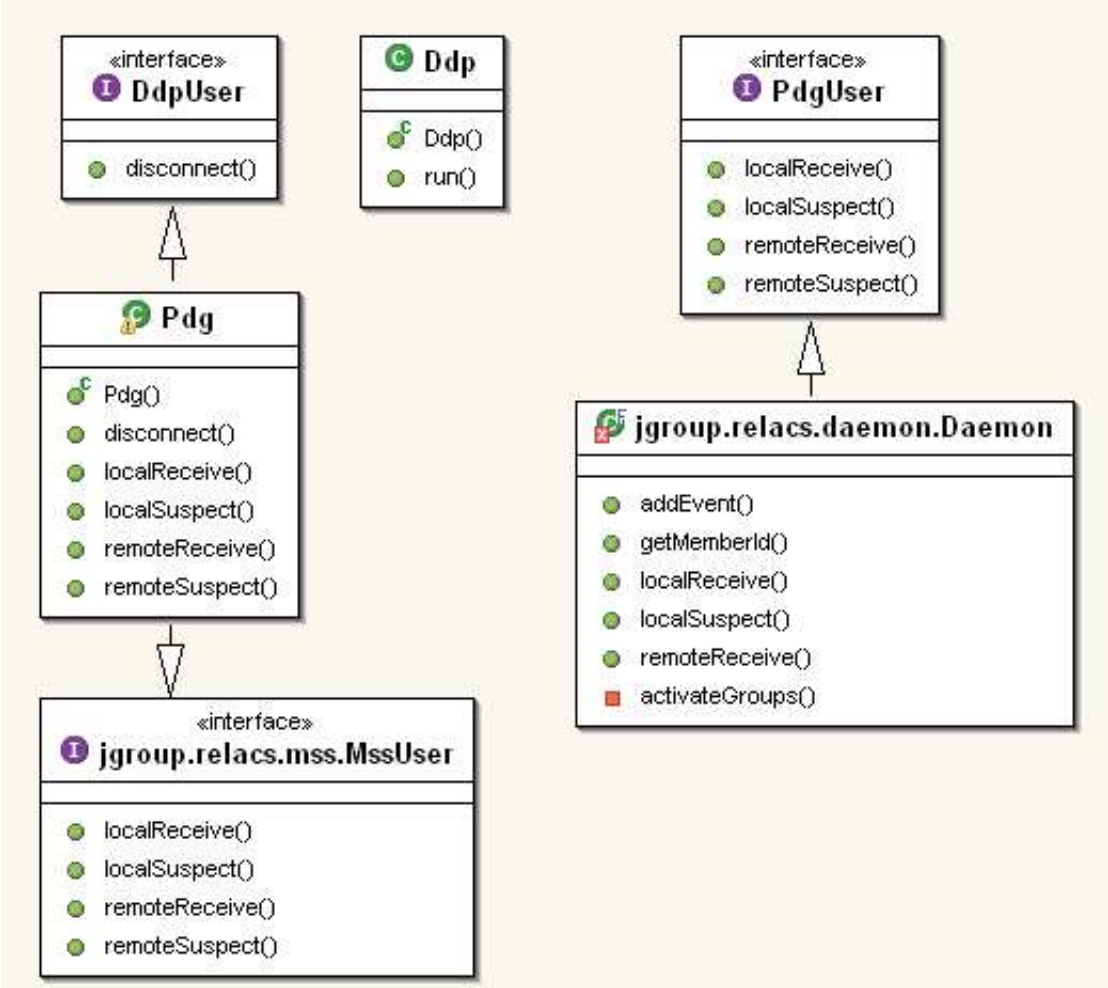

FIG. 5.3 – Diagramme des classes de la couche $\mathcal{PDG}$

5.4 État d'avancement du développement

Pour le développement du prototype, il nous faut étudier le code existant de Jgroup pour comprendre et puis réaliser des modifications. À l'issue du stage de cinq mois, nous avons terminé la première tâche (T1) et nous sommes en train de compléter la deuxième (T2). Ainsi, nous avons réussi à construire l'architecture de notre service de gestion de groupe en utilisant celle

de Jgroup. Dans cet architecture en couche, la couche $\mathcal{PDG}$ intercepte les messages *msuspect* et réalise la simulation des détecteurs de déconnexions et de partitions.

Les fichiers suivants sont créés dans le paquetage *jgroup.relacs.pdg* :

`Ddp.java, DdpUser.java, Pdg.java, PdgUser.java`

De plus, nous réalisons des modifications dans les fichiers existants selon les tâches définies dans la section précédente.

Pour tester le fonctionnement du prototype, nous utilisons une application dans le paquetage *jgroup.test.hello*. Il faut lancer les commandes suivantes :

```
# dregistry
# helloserver
```

La première commande a pour but de créer un service de nom utilisé dans Jgroup. La deuxième est lancée plusieurs fois pour avoir plusieurs processus dans le système. Après l'initialisation, ce processus affichera sa vue actuelle du système. Les autres processus mettent à jour aussi leur vue avec l'apparition du nouveau processus. Si un processus est défaillant (p. ex. par `Ctr-C`) ou déconnecté ou partitionné (en utilisant la simulation), les autres processus afficheront donc le changement correspondant dans leur vue.

Chapitre 6

Conclusions et perspectives

L'apparition du nouveau paradigme appelé « informatique mobile » offre aux utilisateurs la capacité de pouvoir se déplacer tout en restant connecté aux applications répartis et d'être indépendant de la localisation géographique. Il suscite aussi beaucoup de travaux de recherche qui cherchent à résoudre les problèmes existants dans les systèmes répartis avec de nouvelles caractéristiques de l'environnement mobile.

Les impossibilités de consensus, de gestion de groupe sont déjà montrées dans le modèle des systèmes répartis asynchrones. Le modèle partiellement synchrone est proposé pour résoudre ces problèmes, p. ex. en utilisant le détecteur de défaillances non fiable. Pour traiter le problème de gestion de déconnexions en environnement mobile, les autres détecteurs sont proposés : détecteurs de connectivité, de déconnexions et de partitions. Ils sont tous définis de façon abstraite comme le détecteur de défaillances.

Dans ce travail, nous avons réalisé une étude bibliographique sur les détecteurs existants en environnement mobile. Nous avons discuté les limitations à lever et l'idée d'utiliser ces détecteurs pour un service de gestion de groupe ayant de nouvelles propriétés (p. ex. accord de partition). Nous avons aussi choisi une implantation existante des systèmes de communication de groupe pour réutiliser dans notre recherche et argumenté notre choix grâce à une étude sur les travaux existants. Ensuite, en se basant sur ce système, nous avons proposé une spécification d'un nouveau service de gestion de groupe et réaliser un prototype de ce service. Nous avons réussi à insérer une simulation simple des détecteurs dans l'architecture de ce système et adapté l'algorithme de gestion de groupe pour satisfaire de nouvelles propriétés de notre service. Ce travail a montré la faisabilité de notre approche et peut être considéré comme une première

expérience pour les autres recherches à venir.

En guise de perspectives, nous devons compléter le développement du prototype et exécuter des tests de performance. D'autre part, nous étudions également la possibilité de réaliser une implantation de ce service de gestion de groupe sur les terminaux mobiles (p. ex. avec J2ME). Pour ce faire, il faut améliorer les algorithmes des détecteurs et de la gestion de groupe 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) (cf. section 2.6).

Bibliographie

[1] 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.

[2] M. Aguilera, W. Chen, and S. Toueg. On quiescent reliable communication. *SIAM Journal of Computing*, 29(6) :2040–2073, 2000.

[3] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report TR 95-1534, Department of Computer Science, Cornell University, Ithaca, New-York (USA), Aug. 1995.

[4] Appia. Appia home page. http://appia.di.fc.ul.pt.

[5] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group communication in partitionable systems : specification and algorithms. *IEEE Transactions on Software Engineering*, 27(4) :308–336, Apr. 2001.

[6] M. U. Bhatti. Partitions in wireless group communication system. Master's thesis, Université de Paris-Sud, Sept. 2004.

[7] M. U. Bhatti and D. Conan. Détections de partition pour la gestion de groupes en environnement mobile. In *Actes de la 2ème Conférénce Francophone Mobilité & Ubiquité 2005*, Grenoble, France, May 2005. ACM International Conference Proceedings Series.

[8] K. P. Birman. *Building Secure and Reliable Network Applications.* Manning, USA, 1995.

[9] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proc. 15th ACM Symposium on Principles of Distributing Computing*, Philadeplhia, USA, May 2005.

[10] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the Association for Computing Machinery*, 43(2) :225–267, Mar. 1996.

[11] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications : A comprehensive study. *ACM Computing Surveys*, 33(4) :427–469, Dec. 2001.

[12] C. Delporte-Gallet. Sur l'algorithmique distribué tolérant aux pannes, Dec. 2001. Thèse HDR, Université Paris VII.

[13] R. Ekwall, A. Schiper, and P. Urbán. Token-based atomic broadcast using unreliable failure detectors. In *Proc. of the 23rd IEEE International Symposium on Reliable Distributed Systems*, Washington, DC, USA, Oct. 2004. IEEE Computer Society.

[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2) :374–382, Apr. 1985.

[15] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1) :1–26, Mar. 1999.

[16] Javagroup. Javagroup home page. http://www.jgroups.org.

[17] Jgroup. Jgroup home page. http://jgroup.sourceforge.net.

[18] J. Jing, A. Helal, and A. Elmagarmid. Client-server computing in mobile environments. *ACM Computing Surveys*, 31(2) :117–157, June 1999.

[19] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe : A group membership service for wans. *ACM Transactions on Computer Systems*, 20(3) :191–238, Aug. 2002.

[20] S. Mena, A. Schiper, and P. Wojciechowski. A step towards a new generation of group communication systems. In *Proc. of International Middleware Conference 2003*. Springer Verlag, June 2003.

[21] A. Montresor. *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems*. PhD thesis, Dept. of Computer Science, University of Bologna, Bologna, Italy, Feb. 2000.

[22] R. Prakash and R. Baldoni. Architecture for group communication in mobile systems. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, pages 235–242, Washington, DC, USA, 1998. IEEE Computer Society.

[23] M. Raynal. A short introduction to failure detectors for asynchronous distributed systems. *ACM SIGACT News*, 36(1) :53–70, Mar. 2005.

[24] M. Satyanarayanan. Fundamental challenges in mobile computing. In *PODC '96 : Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 1–7, New York, NY, USA, 1996. ACM Press.

[25] M. Satyanarayanan. Pervasive computing : Vison and challenges. *IEEE Personal Communications*, 8(4) :10–17, Aug. 2001.

[26] L. Temal and D. Conan. Détections de défaillances, de connectivité et de déconnexions. In *Actes de la 1ère Conférénce Francophone Mobilité & Ubiquité*, pages 90–97, Nice, France, June 2004. ACM International Conference Proceedings Series.

Annexe A

Preuves

Dans cette section, nous présentons la preuve de l'algorithme de gestion de groupe, donné en section 5.2. Nous devons montrer que l'algorithme se termine toujours et qu'il satisfait les propriétés AGM1–AGM5 présentées dans la spécification du nouveau service de gestion de groupe (cf. section 4.6). Puisque cet algorithme se base sur celui de Jgroup [5], les preuves sont donc présentées en utilisant la même organisation.

Lemme 1 *Le nombre de messages* PROPOSE *envoyé par un processus est borné.*

PREUVE. Les messages PROPOSE sont envoyés dans la procédure *SendEstimate*. Parce que nous utilisons la même condition d'appel de cette procédure que l'original, [5] a montré que le nombre de messages PROPOSE envoyé par un processus est borné.

Lemme 2 *Si un processus correct p entre dans la procédure $EstimateExchangePhase$ avec une vue v, alors p installe ultimement une nouvelle vue après v.*

PREUVE. L'algorithme se comporte comme la version originale donc la preuve est identique à [5].

Lemme 3 *Si un processus correct p entre dans la procédure $SynchronizationPhase$ avec une vue v, alors p entre ultimement dans la procédure $EstimateExchangePhase$ de v.*

PREUVE. L'algorithme se comporte comme la version originale donc la preuve est identique à [5].

Corollaire 1 *Si un processus correct p entre la procédure $AgreementPhase$ avec une vue v, alors il installe ultimement une nouvelle vue après v.*

PREUVE. Des lemmes 2 et 3.

Théorème 1 (Précision des vues) *L'algorithme satisfait la propriété AGM1.*

PREUVE. Dans notre algorithme, le concept de vue dans [5] correspond bien à la vue de groupe *view.comp*. C'est la raison pour laquelle la preuve de la sous propriété AGM1.1 est identique à [5]. Nous devons donc prouver ici la sous propriété AGM1.2, c'est-à-dire il existe un instant après lequel aucun processus n'appraît dans les vues des défaillances, des déconnexions ou des partitions avant qu'il ne soit défaillant, déconnecté ou partitionné. Chaque processus utilise la sortie du détecteur de partition comme l'entrée de l'algorithme de gestion de groupe. Ce sont les trois ensembles des processus défaillants f_p, vus déconnectés d_p et vus partitionnés p_p. L'ensemble des processus corrects connectés est calculé à partir de ces trois ensembles ($reachable = \Pi \setminus (f_p \cup d_p \cup p_p)$). Au cours de l'algorithme de gestion de groupe, ces ensembles de chaque processus s'évoluent (R1–R2) et se convergent (R3–R4) en échangeant leur propre estimation. À la terminaison de cet algorithme, tous les processus membres du groupe se mettent d'accord sur ces ensembles. Cette propriété est donc assurée grâce à la propriété de précision de partition forte finale du détecteur de partition.

Théorème 2 (Complétude des vues) *L'algorithme satisfait la propriété AGM2.*

PREUVE. Dans notre algorithme, le concept de vue dans [5] correspond bien à la vue de groupe *view.comp*. C'est la raison pour laquelle la preuve de la sous propriété AGM2.1 est identique à [5]. Nous devons donc prouver ici la sous propriété AGM2.2, c'est-à-dire 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. Chaque processus utilise la sortie du détecteur de partition comme l'entrée de l'algorithme de gestion de groupe. Ce sont les trois ensembles des processus défaillants f_p, vus déconnectés d_p et vus partitionnés p_p. L'ensemble des processus corrects connectés est calculé à partir de ces trois ensembles ($reachable = \Pi \setminus (f_p \cup d_p \cup p_p)$). Au cours de l'algorithme de gestion de groupe, ces ensembles de chaque processus s'évoluent (R1–R2) et se convergent (R3–R4) en échangeant leur propre estimation. À la terminaison de cet algorithme, tous les processus membres du groupe se mettent d'accord sur ces ensembles. Cette propriété est donc assurée grâce à la propriété de complétude de partition forte du détecteur de partition.

Théorème 3 (Cohérence des vues) *L'algorithme satisfait la propriété AGM3.*

PREUVE. L'algorithme se comporte comme la version originale donc la preuve est identique à [5].

Théorème 4 (Ordre des vues) *L'algorithme satisfait la propriété AGM4.*

PREUVE. L'algorithme se comporte comme la version originale donc la preuve est identique à [5].

Théorème 5 (Intégrité des vues) *L'algorithme satisfait la propriété AGM5.*

PREUVE. Dans notre algorithme, le concept de vue dans [5] correspond bien à la vue de groupe *view.comp*. C'est la raison pour laquelle la preuve de la sous propriété AGM5.1 est identique à [5]. Nous devons donc prouver ici la sous propriété AGM5.2, c'est-à-dire l'intersection des vues de groupe, des défaillances, des déconnexions et des partitions est l'ensemble vide. Dans notre algorithme, les estimations de chaque processus sont utilisés comme candidats pour les nouvelles vues. L'ensemble de processus corrects accessibles est calculé à partir des trois ensembles de processus défaillants, vus déconnectés et vus partitionnés ($reachable = \Pi \setminus (f_p \cup d_p \cup p_p)$). En outre, en appliquant les règles pour passer un processus d'un ensemble à l'autre, nous assurons la convergence des trois ensembles de processus défaillants, vus déconnectés et vus partitionnés et résolvons le problème de l'incohérence entre différents processus.