Université de Versailles Saint-Quentin-En-Yvelines
Institut National des Télécommunications

Rapport de Stage
DEA Méthodes Informatiques des Systèmes Industriels
(M.I.S.I)

GESTION DE DÉCONNEXION ET TOLÉRANCE AUX FAUTES

Lynda TEMAL
Responsable de DEA : Dominique BARTH
Responsable de stage : Denis CONAN

Septembre 2003

Ce stage de DEA a été réalisé au sein du laboratoire **Systèmes Répartis** du département **Informatique** de l'**Institut National des Télécommunications**

Remerciements

Je tiens à exprimer ma profonde gratitude à tous ceux qui ont apporté leur soutien et leur aide, de près ou de loin pour réaliser ce travail.
Je remercie :

- ***Guy Bernard*** pour m'avoir accueilli dans l'équipe « Systèmes Répartis » de l'Institut National des Télécommunications.
- ***Denis Conan*** qui n'a nullement ménagé ses efforts en vue de m'apporter aide et assistance durant les six mois de stage.

Mes vifs remerciements vont à l'ensemble des enseignants du DEA **MISI** qui m'ont encadré durant ma formation.
Je remercie également l'ensemble de ma famille pour leurs encouragements.
Je remercie infiniment mes amis en particulier : ***Nabiha*** et ***Fazou*** pour leur soutien continu

Merci à tous ceux que j'aime et que je n'ai pas cité.

Table des matières

Remerciements **i**

Introduction **1**

1 Gestion de déconnexion **3**
1.1 Quelques caractéristiques des environnements mobiles 3
1.2 Plate-forme *Domint* 5
1.2.1 Architecture 6
1.2.2 Entités 7
1.3 Hystérésis pour la détection de connectivité 8
1.4 Conclusion 9

2 Tolérance aux fautes **11**
2.1 Modèle 13
2.1.1 Modèles de défaillances 13
2.1.2 Principe des détecteurs de défaillances non fiables 13
2.1.3 Propriétés de la détection de défaillances non fiables 14
2.1.4 Classes de détecteurs de défaillances non fiables 15
2.1.5 Réductibilité 15
2.2 Paradigme du consensus 17
2.2.1 Résolution du consensus avec $\diamond\mathcal{S}$ 18
2.3 Travaux connexes 19
2.3.1 Aspects théoriques 19
2.3.2 Aspects pratiques 21
2.4 Détecteur de défaillances dit battements de cœur 22
2.4.1 Modèle et propriétés 22
2.4.2 Implémentation de $\mathcal{HB}$ pour des réseaux simples 23
2.5 Conclusion 24

3 Détecteurs de connectivité, de déconnexions et de défaillances **25**
3.1 Approches proposées 25
3.2 Détecteurs de défaillances pour la gestion de déconnexion 26
3.2.1 Motivations 26

3.2.2 Choix du détecteur de défaillances $\mathcal{HB}$ 28
3.2.3 Architecture 29
3.3 Détecteurs de connectivité pour la tolérance aux fautes 30
3.3.1 Motivations 30
3.3.2 Consensus et détecteurs de défaillances et de déconnexions 31
3.3.3 Du détecteur de connectivité au détecteur de déconnexions réparti 33
3.3.3.1 Modèle de déconnexions 33
3.3.3.2 Détecteur de déconnexions 33
3.3.3.3 Propriétés 34
3.3.3.4 Complément sur les propriétés de l'hystérésis 35
3.3.3.5 Algorithme du détecteur de déconnexions 35
3.3.3.6 Preuve de correction 38
3.3.4 Nouvel algorithme du consensus 39
3.3.4.1 Propriétés du consensus 39
3.3.4.2 Résolution du consensus avec les détecteurs de défaillances et de déconnexions 39
3.3.4.3 Preuve de correction 40
3.4 Détection de déconnexions dans le détecteur de défaillances 44
3.4.1 Architectures 44
3.4.2 Modèle et propriétés 45
3.4.3 Algorithme de détection de défaillances et de déconnexions 45
3.4.4 Preuve de correction 46
3.5 Conclusion 48

4 Conclusion et perspectives **49**

A Déconnexion, reconnexion et réconciliation dans un consensus **51**
A.1 Motivations 51
A.2 Architecture et scénario 51
A.3 Modèle et propriétés 52
A.4 Algorithmes 53

Table des figures

1.1 Caractéristiques des environnements mobiles . . . 5
1.2 Comportement de *Domint* en mode fortement connecté . . . 6
1.3 Hystérésis du gestionnaire de connectivité . . . 8
1.4 Le diagramme de transitions d'états UML de l'hystérésis. . . . 9

2.1 Relations entre les classes de détecteurs de défaillances . . . 16
2.2 $T_{\mathcal{D} \longrightarrow \mathcal{D}'}$: de la complétude faible à la complétude forte . . . 17
2.3 Algorithme du consensus avec $\diamond\mathcal{S}$ . . . 20
2.4 Implémentation de $\mathcal{HB}$ pour un réseau simple . . . 24

3.1 Détecteur de connectivité utilisant le détecteur de défaillances . . . 29
3.2 Fréquence des battements de cœur projetée sur un hystérésis . . . 30
3.3 Arrêt d'envoi de message lors d'une déconnexion . . . 31
3.4 Utilisation des détecteurs de défaillances et de connectivité par le consensus . . . 32
3.5 Algorithme du détecteur de déconnexions réparti . . . 36
3.6 De la complétude de déconnexion faible à la complétude de déconnexion forte . . 37
3.7 Nouvel algorithme du consensus $\mathcal{CD}$ (« *Consensus with Disconnections* ») . . . 41
3.8 Collaboration des détecteurs de connectivité et de défaillances . . . 44
3.9 Algorithme $\mathcal{HBD}$ . . . 46

A.1 Interaction du consensus, du détecteur de défaillances, du détecteur de connectivité et du mandataire . . . 52
A.2 Algorithme du mandataire . . . 53
A.3 Algorithme du détecteur de défaillances fournissant les ensembles des processus suspectés, vus déconnectés et des mandataires. . . . 54
A.4 Algorithme du consensus . . . 56
A.5 Réconciliation du processus p . . . 57

Liste des tableaux

2.1 Quatre formes de tolérance aux fautes 12
2.2 Classes des détecteurs de défaillances 16

Introduction

La large utilisation des ordinateurs portables et les progrès technologiques dans les réseaux sans fil ont fait des applications mobiles une réalité. De nombreuses applications et interfaces fournissent déjà divers services dédiés aux dispositifs mobiles. La communication sans fil, le traitement d'information personnelles et les services d'information réparties auront une importance stratégique dans l'avenir proche.

Malheureusement, le terminal mobile est plus souvent exposé à des adversités dans l'environnement. En particulier, le terminal mobile est sujet à des déconnexions fréquentes. Il existe deux types de déconnexions : les déconnexions volontaires et les déconnexions involontaires. Les premières sont décidées par l'utilisateur. Les secondes sont le résultat de coupures physiques du réseau. Les déconnexions étant très fréquentes, une machine mobile doit être capable de continuer ses exécutions même avec une faible connexion ou sans connexion du tout. Pour cela un mécanisme de détection du niveau de connectivité est nécessaire.

[CCVB03] présente un algorithme de détection de connectivité permettant de décider si la requête de l'utilisateur peut être transmise ou non sur le réseau sans fil. Un mécanisme de chargement de certaines entités sur le terminal mobile permet de continuer à travailler localement lors de la déconnexion. Un autre mécanisme intervient lors de la reconnexion pour effectuer une réconciliation.

Dans une application répartie, la défaillance d'un des processus fait défaillir l'ensemble de l'application. La tolérance aux fautes est obligatoire pour la fiabilité de ces applications, elle se base généralement sur le paradigme du consensus. Ce paradigme peut être utilisé comme brique de base pour résoudre de nombreux problèmes d'accord tels que la diffusion atomique, la validation atomique *etc*. Dans un système réparti asynchrone, il est impossible d'obtenir un consensus entre processus répartis dès qu'un seul d'entre eux est défaillant [FLP85]. Cette impossibilité résulte de la difficulté de dire si le processus distant est juste lent ou s'il est défaillant. Pourtant, beaucoup de mécanismes de tolérance aux fautes nécessitent de telles décisions, ne serait-ce que pour déterminer de façon cohérente l'ensemble des processus corrects de ceux qui sont défaillants à un instant donné. Pour pallier au résultat d'impossibilité, des solutions de tolérance aux fautes utilisent des détecteurs de défaillances non fiables [CT96].

Les deux propriétés extra-fonctionnelles, continuité de service et tolérance aux fautes, reposent chacune sur un mécanisme de détection. Au cours de ce stage, nous axons nos travaux, d'une part, sur l'étude de la détection de connectivité, disponible dans *Domint*, d'autre part, sur l'étude de différents algorithmes de détection de défaillances. L'objectif final de ce stage est d'étudier les types de détections en terme de propriétés et d'algorithmes, puis de suggérer de nouvelles architectures qui associent les détecteurs de connectivité et les détecteurs de défaillances. Ils peuvent être associés de différentes manières, soit pour améliorer la gestion de déconnexion en utilisant les fonctionnalités des détecteurs de défaillances, soit pour distinguer une déconnexion d'une défaillance dans la tolérance aux fautes en utilisant les détecteurs de connectivité.

Pour notre part, nous proposons en premier lieu une architecture pouvant être incorporée dans *Domint*. Le détecteur de connectivité utilise les fonctionnalités d'un détecteur de défaillances, afin d'élargir la visibilité de la machine mobile, en passant d'une visibilité locale des ressources vers une visibilité globale des ressources distantes. La machine mobile est non seulement en

mesure de connaître son niveau de connectivité mais devient aussi en mesure de connaître le niveau de connectivité des machines distantes. Dans un second temps, nous nous intéressons à l'utilisation des informations de connectivité par le consensus qui utilise déjà un détecteur de défaillances, afin de distinguer le traitement d'une déconnexion de celui d'une défaillance. Pour cela, nous avons ajouté à l'algorithme du consensus présenté dans [CT96] un module de détection de déconnexions qui n'est pas tout à fait un détecteur de connectivité, mais plutôt un détecteur de déconnexions qui échange avec les processus de l'application les informations de déconnexions et de reconnexions obtenues grâce au détecteur de connectivité. En d'autres termes, le détecteur de déconnexions n'est qu'un détecteur de connectivité réparti qui envoie et recueille des informations de déconnexions et de reconnexions. Dans cette architecture, le consensus est en mesure de distinguer les processus déconnectés de ceux qui sont défaillants ou corrects. Nous avons remarqué par la suite que l'information de déconnexion est toute aussi intéressante si elle est également exploitée au niveau du module de détection de défaillances. Pour cela, nous ajoutons la détection de déconnexions à un algorithme de détection de défaillances. En final, nous présentons une application où un hôte mobile qui se déconnecte charge un mandataire sur le réseau fixe pour journaliser les décisions du consensus lors de sa déconnexion, préparant ainsi sa reconnexion.

Le rapport est organisé comme suit. Le premier chapitre aborde quelques caractéristiques des environnements mobiles, l'architecture de *Domint* et en particulier le mécanisme de détection de connectivité. Le second chapitre est consacré à la tolérance aux fautes dans les systèmes répartis asynchrones. Il présente le consensus comme paradigme indispensable de la tolérance aux fautes, il énonce le résultat d'impossibilité de résolution de consensus dans un système asynchrone, puis introduit les détecteurs de défaillances pour contourner le résultat d'impossibilité. Le troisième chapitre détaille les approches proposées pour associer les détecteurs de défaillances et les détecteurs de connectivité.

Chapitre 1

Gestion de déconnexion

Durant la dernière décennie, le monde de l'environnement mobile a été témoin d'une énorme effervescence. Les progrès technologiques, surtout dans le domaine des communications, ont vu l'émergence d'un nouveau défit dans le monde informatique : *la communication sans fil*, grâce à laquelle nous sommes passés des réseaux fixes vers les réseaux sans fil inter-connectant des machines mobiles. Plusieurs dispositifs sans fil tels que le téléphone cellulaire et les assistants personnels numériques (PDA), inondent déjà le marché et font partie de la vie quotidienne. Cependant, plusieurs problèmes limitent l'utilisation de ces dispositifs : la faible capacité des ressources et la déconnexion fréquente. Pour palier à ces désagréments, des travaux de recherches se focalisent sur la création de plates-formes ou d'applications qui permettent à l'utilisateur de travailler même en présence de déconnexions ou d'une connectivité très faible. *Domint* est une architecture permettant la continuité de service lors des déconnexions s'appuyant sur un mécanisme de détection de connectivité.

La section 1.1 aborde d'abord quelques caractéristiques des environnements mobiles. La section 1.2 présente ensuite l'architecture CORBA de *Domint*, un système qui s'appuie sur un mécanisme d'*hystérésis* pour la détection de connectivité, détaillé dans la section 1.3. Enfin, la conclusion (*Cf.* section 1.4) positionne nos travaux dans la continuité des travaux sur la gestion de la connectivité.

1.1 Quelques caractéristiques des environnements mobiles

L'émergence de la technologie sans fil permet à l'utilisateur de se libérer des contraintes temporelles et spatiales. En effet, l'utilisateur peut continuer à accéder aux données fournies par une infrastructure répartie, quelque soit son emplacement. Malheureusement, comme signalé précédemment, l'environnement mobile se distingue de l'environnement statique par quelques problèmes majeurs :

- La connexion sans fil dispose d'une bande passante très variable en terme de performance et de fiabilité.

- Contrairement aux environnements distribués statiques, l'environnement mobile est sujet à des déconnexions fréquentes qui deviennent la règle et non une exception.

- Les dispositifs mobiles sont plus pauvres en ressources telles que la capacité de la mémoire et la vitesse d'exécution.

- Les dispositifs mobiles dépendent de la charge de la batterie, qui est une source d'énergie à capacité limitée.

La figure 1.1 résume les caractéristiques principales des machines et des moyens de communication sans fil utilisés en informatique mobile.

En dépit de tous les problèmes mentionnés ci-dessus, l'utilisation des machines mobiles devient de plus en plus courante. Il est donc souhaitable de fournir les éléments nécessaires pour que leur utilisation devienne aussi naturelle que possible. Un utilisateur mobile souhaite se déplacer librement et continuer à travailler le plus normalement possible avec son terminal mobile sans se préoccuper de sa position. Il convient alors de trouver des solutions pour régler ces désagréments ou du moins les minimiser en adaptant les applications aux environnements mobiles.

Dans ce rapport, nous nous intéressons au problème de la déconnexion. On peut distinguer deux types de déconnexions. Les déconnexions *volontaires* sont décidées par l'utilisateur depuis son terminal mobile. Elles sont justifiées par les bénéfices attendus sur le coût financier des communications, le niveau d'énergie de la batterie, la disponibilité du service applicatif et la minimisation des désagréments induits par des déconnexions inopinées. Par contre, les déconnexions *involontaires* sont le résultat de coupures intempestives des connexions physiques dues par exemple à un passage de l'utilisateur dans une zone d'ombre radio. Dans un système réparti statique, une machine ne peut travailler que dans deux modes différents, soit connectée au réseau, soit totalement déconnectée. Par contre en environnement mobile, il existe des degrés variés de déconnexion ; un mobile peut opérer dans trois modes :

- Le mode connecté : dans ce cas, le mobile dispose d'une connexion normale au réseau comme une station classique. La connexion peut alors être réalisée soit par une liaison filaire, soit par une interface de communication sans fil, qui fournit en général des débits plus faibles qu'une liaison câblée.

- Le mode partiellement connecté : dans ce cas, le mobile ne dispose plus pour communiquer avec le réseau que d'un lien à faible largeur de bande (connexion faible ou déconnexion partielle). Cette perte de capacité de la bande passante peut être due soit à des perturbations ou à des surcharges de la station de base qui gère les communications des mobiles se trouvant dans sa cellule, soit à l'éloignement du mobile de la station de base.

- Le mode déconnecté : dans ce cas, le mobile est totalement déconnecté du réseau.

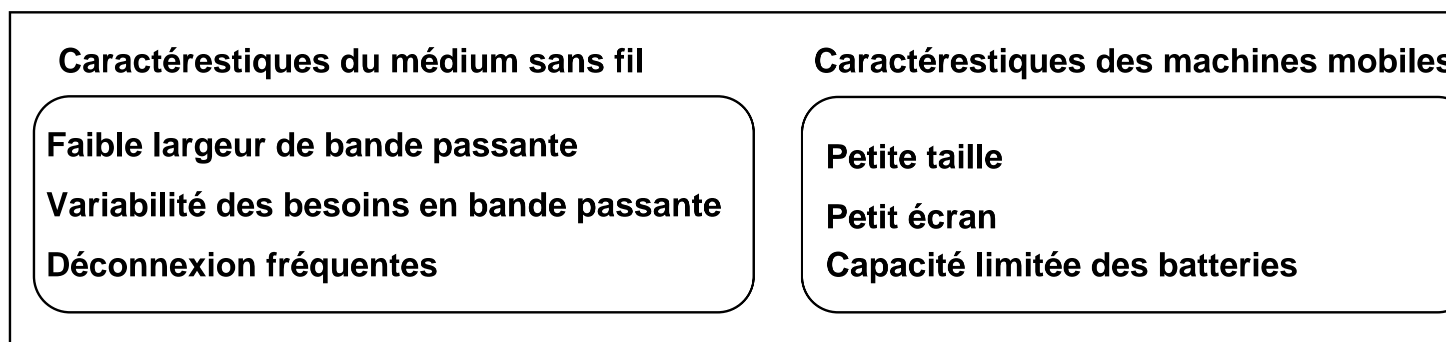

FIG. 1.1 – Caractéristiques des environnements mobiles

Les déconnexions totales ou partielles étant très fréquentes pour un mobile connecté via une liaison sans fil, une machine mobile doit être capable de continuer ses exécutions même avec une faible connexion au réseau ou plus de connexion du tout. Tandis que dans un environnement réparti non mobile la plupart des déconnexions ne peuvent pas être prévues, en environnement mobile, elles peuvent certaines fois être détectées et un protocole spécifique peut donc être mis en oeuvre pour les prendre en compte. Le protocole de déconnexion doit être exécuté avant que la machine mobile ne soit physiquement détachée du réseau ; ce protocole est tenu de s'assurer que suffisamment d'informations sont présentes localement pour garantir au mobile son autonomie durant la déconnexion. Le protocole de déconnexion partielle a pour rôle de préparer le mobile à exécuter les opérations dans un mode où toutes les communications avec le réseau fixe seront aussi réduites que possibles. Le protocole de réconciliation permet de rétablir les connexions avec le réseau fixe et de restaurer un mode d'opération normal (totalement connecté).

Les travaux de recherche précédents [CCVB03] se sont penchés sur cet aspect de déconnexion involontaire qui provoque maints désagréments lors des traitements distribués sur des machines distantes. Un algorithme de détection de connectivité permet de connaître le niveau de connectivité des ressources pour décider si la requête de l'utilisateur peut être transmise ou non sur le réseau sans fil. La fonctionnalité de détection de connectivité est assurée par un *hystérésis* qui stabilise l'application en lissant les variations de la disponibilité des ressources. Il est incorporé dans *Domint*, une plate-forme CORBA qui a pour mission de charger certaines entités sur le terminal mobile afin de permettre de continuer à travailler localement lors de la déconnexion.

1.2 Plate-forme *Domint*

Domint est donc une plate-forme qui fournit une continuité de service. Il dispose de deux mécanismes principaux. D'une part, il supporte la réplication des objets, et d'autre part, il supporte des déconnexions volontaires et involontaires. Dans *Domint*, CORBA est choisi pour sa capacité à être utilisé dans des domaines multiples et parce qu'il fournit des mécanismes pour établir une adaptation transparente à l'application. Afin de continuer à travailler même lors de déconnexions, l'idée principale de *Domint* est de créer sur le terminal mobile des mandataires appelés

les objets déconnectés (DO). Un DO est un objet CORBA spécifiquement construit pour faire face aux déconnexions et à la faible connectivité. Dans les prochaines sections, nous décrivons l'architecture de *Domint* et le fonctionnement de tous les objets de *Domint*.

1.2.1 Architecture

L'architecture de *Domint* est représentée dans la figure 1.2. Elle présente un diagramme de collaborations UML du client envoyant la première requête à un objet distant quand la connectivité est forte. Tous les rectangles sur la figure 1.2 représentent des objets CORBA. L'intercepteur portable PI est également un objet CORBA, mais local, c'est-à-dire qu'il ne peut pas être appelé en dehors de son entité d'exécution (une entité d'exécution est attachée à un seul ORB). Toutes les requêtes du ou vers le client sont interceptées par PI. Sur les requêtes sortantes, PI agit en tant que commutateur entre le DO et l'objet distant. Sur la réception de la réponse, PI détecte les erreurs de communication entre l'envoi de la demande et la réception de la réponse.

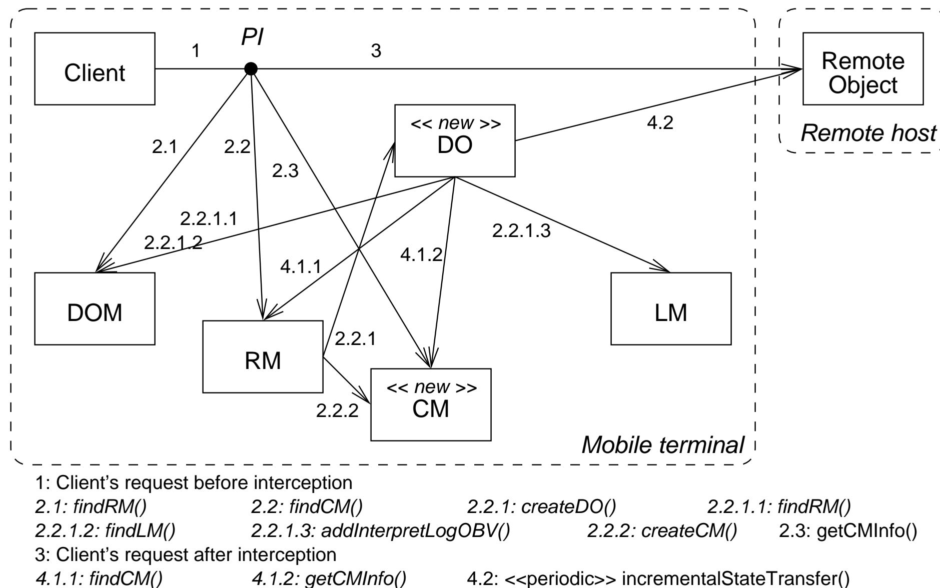

FIG. 1.2 – Comportement de *Domint* en mode fortement connecté

Au premier appel de l'objet distant, RM (Resource Manager) crée un DO (Disconnected Object) et le CM (Connectivity Manager) correspondant sur la demande de PI. La figure 1.2 montre les interactions entre les différents objets de *Domint* lors du premier appel à un objet distant, en cas de forte connectivité. Dans le diagramme, chaque flèche est une requête CORBA. et 3 sont des cas particuliers : la requête 1 est interceptée par PI qui ne l'interprète pas mais laisse l'ORB la transmettre en tant que requête 3 à l'objet distant. Entre ces deux demandes, PI recherche la référence de CM associé à cet objet dans sa table interne et conclut que c'est le premier appel pour cet objet distant. Dans ce cas, PI demande à DOM (2.1) la référence de RM pour que ce dernier crée DO (2.2.1) et le CM associé (2.2.2). Pendant sa construction, DO

obtient les références de RM et de LM en appelant DOM (2.2.1.1, 2.2.1.2). Ensuite, PI décide où la requête du client va être envoyée (soit vers l'objet distant soit vers l'objet déconnecté). Dans le scénario où la connectivité est forte, les requêtes du client sont exécutées sur les objets distants (3). Enfin, l'objet déconnecté appelle périodiquement l'objet distant pour un transfert incrémental d'état (4.2). Cependant, il doit d'abord contrôler la connectivité en appelant le CM (4.1.2).

En générale, à chaque requête à un nouvel objet distant, PI demande à DOM (Disconnected Object Manager) (2.1) la référence de RM pour que ce dernier crée DO (2.2.1) et le CM associé (2.2.2). DO obtient les références de RM et de LM (Log Manager) en appelant DOM (2.2.1.1, 2.2.1.2). À chaque nouvelle requête, PI interroge CM pour décider la destination de la requête (soit vers l'objet distant soit vers l'objet déconnecté).

[CCVB03] présente un autre diagramme similaire du comportement de *Domint* pour le traitement de requêtes des utilisateurs lorsque la connectivité est faible ou nulle. Dans ce diagramme, le client est forcé de rentrer respectivement en mode partiellement connecté ou déconnecté pour travailler localement grâce à l'objet déconnecté, tout en maintenant une journalisation des opérations par LM. Cette commutation entre les modes se fait d'une manière transparente aux applications.

1.2.2 Entités

Après avoir vu la collaboration entre les différentes entités composant *Domint*. Cette section donne le rôle de chacune de ces entités.

Gestionnaire d'objets déconnectés : DOM est le point d'entrée du service de gestion des objets déconnectés. Il possède la référence du gestionnaire de ressources RM et du gestionnaire du journal d'opérations LM.

Gestionnaire de ressources : RM centralise la surveillance des ressources dans le terminal mobile. *Domint* se focalise sur les ressources liées à la connectivité : l'activité du réseau, la largeur de bande disponible, le coût de transmission, la charge de la batterie... L'intercepteur PI, l'objet déconnecté DO et le gestionnaire du journal LM obtiennent la référence d'un gestionnaire de connectivité via RM.

Gestionnaire de connectivité : CM manipule une connexion logique entre un client sur le terminal mobile et un objet distant sur le terminal fixe. *Domint* définit un mécanisme d'hystérésis (*Cf.* section 1.3) qui permet de gérer la connectivité entre l'objet distant et DO.

Gestionnaire du journal d'opérations : LM est responsable du contrôle des opérations. LM ne sait pas interpréter les données enregistrées, mais les objets déconnectés fournissent le code (via un objet par valeur CORBA) qui permet par exemple de transmettre les requêtes à l'objet distant.

FIG. 1.3 – Hystérésis du gestionnaire de connectivité

1.3 Hystérésis pour la détection de connectivité

Comme souligné dans la section précédente, le gestionnaire de connectivité s'appuie sur un mécanisme d'*hystérésis*. Ce dernier est relié directement aux ressources locales telles que le niveau de la batterie, la largeur de la bande passante et l'activité du réseau. Il détecte le taux d'utilisabilité des ressources locales. Le mécanisme d'hystérésis évite de trop fréquents transferts d'état et des commutations entre l'objet déconnecté et l'objet serveur distant. De cette manière, il stabilise les applications et lisse les variations de la disponibilité des ressources locales. Pour cela, est introduit le mode partiellement connecté, ajouté au mode connecté et au mode déconnecté.

1. **Mode connecté** : le mobile dispose d'un accès normal à l'objet distant. Les requêtes du client sont directement envoyées à l'objet distant. Les traitements se font uniquement sur l'hôte distant.
2. **Mode partiellement connecté** : le mobile ne dispose plus pour communiquer que d'un accès faible liaison. Dans ce cas de figure, les requêtes sont traitées au niveau local, puis si le niveau de connectivité le permet, sur l'hôte distant.
3. **Mode déconnecté** : le mobile est déconnecté soit parce qu'il n'y a plus d'accès physique soit parce que l'utilisateur ne veut plus de communication sans fil. Dans ce cas de figure, la requête du client est uniquement exécutée localement. Les opérations sont exécutées par l'objet déconnecté et journalisé par LM.

Sur la figure 1.3-1, quand le niveau de ressource augmente et est inférieur au seuil *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). Sans la figure 1.3-2, il peut y avoir un effet « *ping-pong* » qui se produit lors de petites variations autour de la valeur *highDown* et autour de la valeur *lowUp* impliquant le transfert d'état et du journal d'opérations. Par exemple, quand la connectivité arrive dans l'état *F* en provenance de l'état *E* et quand le niveau de ressource dépasse la valeur *highDown*, le mode demeure « partiellement connecté » jusqu'à la valeur *highUp*.

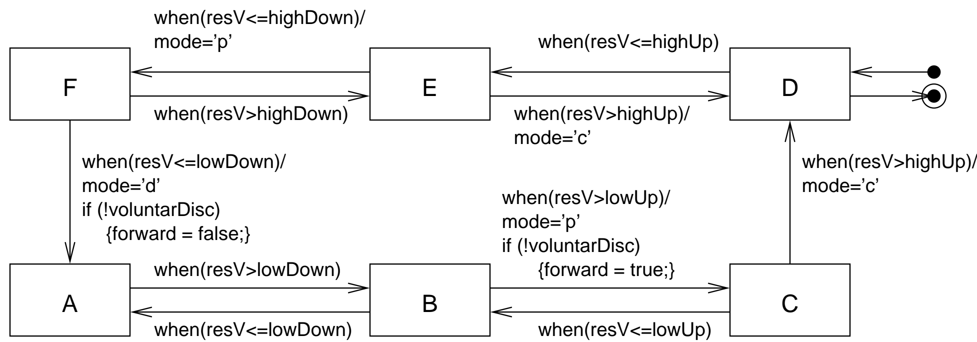

FIG. 1.4 – Le diagramme de transitions d'états UML de l'hystérésis.

Le diagramme de transitions d'état UML de l'hystérésis est représenté dans la figure 1.4. Il montre l'affectation du mode si le client demande ou non une déconnexion volontaire et l'affectation du booléan `forward`, variable qui indique si l'objet déconnecté et le gestionnaire du journal d'opérations peuvent transmettre les requêtes DO à l'objet distant. L'état *D* suit immédiatement l'état initial et précède immédiatement l'état final, de sorte que chaque première requête d'un client vers un objet distant et la fin de l'application se produisent uniquement lorsque le terminal mobile est connecté. L'hystérésis continue à évoluer même si le client demande une déconnexion volontaire.

Domint offre une interface pour configurer et personnaliser les quatre seuils suivant les besoins de l'utilisateur. Il est également reconfigurable dynamiquement par les applications.

1.4 Conclusion

Dans ce chapitre, nous avons présenté les deux formes de déconnexions dans un environnement mobile, volontaire et involontaire. Puis, nous avons décrit l'architecture du système CORBA *Domint* qui adapte les applications suivant le niveau de connectivité, et enfin, nous avons défini le mécanisme d'hystérésis de *Domint* qui détecte la connectivité selon le niveau de disponibilité des ressources locales : largeur de la bande passante, niveau d'énergie de la batterie....

Le mécanisme d'hystérésis nous permet de prédire le futur proche, ce qui peut permettre de disposer de suffisamment de temps pour prendre une décision ou effectuer quelques opérations avant une déconnexion. Dans le chapitre suivant, nous présentons une autre partie complètement différente et sans relation apparente, la détection de défaillances dans un environnement asynchrone.

Chapitre 2

Tolérance aux fautes

La tolérance aux fautes dans les systèmes répartis est un domaine très vaste qui a fait l'objet d'une littérature abondante. La tolérance aux fautes est la capacité du système à suivre le comportement défini, et cela même en présence de fautes[Gär99]. L'exigence de tolérance aux fautes est incontournable avec les systèmes répartis. D'une part, la tolérance aux fautes est un besoin induit par la multiplicité des ressources et d'autre part de nombreux travaux sur les systèmes répartis ont pour but de garantir que la sûreté de fonctionnement de ces systèmes n'est pas dégradée par la répartition. Par ailleurs, la tolérance aux fautes peut être en elle-même un facteur motivant la répartition. En effet, la tolérance aux fautes ne peut être assurée sans redondance et la répartition des traitements et des données sur des processeurs différents permet de structurer et gérer cette redondance. La tolérance aux fautes dans de tels systèmes est mise en œuvre principalement par logiciel, ce qui permet d'espérer une certaine pérennité des moyens vis-à-vis des évolutions du matériel.

Un système tolérant aux fautes satisfait deux propriétés : la *sûreté* et la *vivacité*. Un algorithme réparti est correct s'il satisfait les propriétés de sûreté et de vivacité. La sûreté est une propriété continue, elle est garantie grâce à la détection des fautes. La vivacité est une propriété éventuelle, elle est assurée grâce à la correction [Gär99]. Il existe quatre formes de tolérance aux fautes classées dans le tableau 2.1. Si un algorithme satisfait continuellement les deux propriétés de sûreté et de vivacité alors il utilise le « *masquage* », c'est la forme de tolérance aux fautes la plus coûteuse et la plus souhaitable car le programme est capable de tolérer les fautes d'une manière transparente. Si au contraire il ne satisfait aucune des deux propriétés alors il n'offre aucune forme de tolérance aux fautes. Dans les formes intermédiaires, l'une satisfait la sûreté et pas la vivacité, et inversement, l'autre satisfait la vivacité et pas la sûreté. La première forme nommée « *fail safe* » est préférable à la seconde nommée « *non masquage* » dans le sens où la sûreté est plus importante que la vivacité [Gär99].

La majorité des travaux de recherche en tolérance aux fautes sont inscrits sous la première forme de tolérance aux fautes. Durant ces dernières années, plusieurs paradigmes ont été conçus et mis en œuvre pour simplifier cette tâche. Le plus répandu, le *consensus*, est une forme générale d'accord. Il permet aux processus de prendre une décision commune, dépendant de leurs valeurs initiales, malgré la présence de défaillances. L'algorithme du consensus peut être utilisé pour résoudre plusieurs problèmes pratiques tels que l'élection d'un leader de groupe, et l'accord sur

	Vivace	Non Vivace
Sûre	masquage	silence sur défaillances (« *fail-safe* »)
Non Sûre	non masquage	aucun

TAB. 2.1 – Quatre formes de tolérance aux fautes

une variable répartie.

Garantir la tolérance aux fautes dans un système reparti *asynchrone* qui communique par envoi de messages sans aucune supposition sur les délais de transfert et la vitesse d'exécution garantit aussi la tolérance aux fautes dans des systèmes synchrones ou partiellement synchrones : le système asynchrone est le plus faible en terme de contraintes temporelles. Bien que ces propriétés rendent ce modèle très attractif, il devient aussi impossible de résoudre le consensus d'une manière déterministe dans un système asynchrone sujet à un seul *crash* de processus [FLP85]. Cette impossibilité est due à la difficulté de déterminer si un processus est actuellement défaillant ou s'il est simplement très lent.

Dans cette étude, nous nous intéressons uniquement aux solutions du problème du consensus dans les systèmes répartis asynchrones. Pour palier à ce résultat d'impossibilité, plusieurs approches ont vu le jour : l'utilisation de techniques aléatoires, l'étude de différents modèles de systèmes répartis partiellement synchrones, et l'utilisation de détecteurs de défaillances non fiables.

Pour notre part, nous nous sommes intéressés aux détecteurs de défaillances non fiables proposés dans [CT96]. Dans les systèmes répartis asynchrones, le délai de transfert d'un message ou le temps d'exécution d'une unité de traitement ne sont pas bornés. Ainsi, l'absence de réponse de la part d'un processus ne signifie pas nécessairement qu'il est défaillant. Constatant que la détection des défaillances dans un système asynchrone est obligatoirement *approximative*, Chandra et Toueg ont introduit la notion de détecteurs de défaillances non fiables. Chaque processus a accès à un module détection de défaillances local qui maintient une liste de processus suspectés d'être défaillants. Le module de détection de défaillances peut faire de fausses suspicions en ajoutant par erreur un processus non défaillant à sa liste de suspects. Autrement dit, il peut suspecter un processus p d'être défaillant alors que p s'exécute toujours correctement. Si ce module découvre plus tard qu'il a fait une erreur, il enlève le processus p de la liste des suspects. Ainsi, il peut ajouter et enlever les processus de sa liste des suspects plusieurs fois pendant son exécution. En outre, à un instant donné, deux modules dans deux hôtes différents peuvent avoir des listes de suspects différentes. Chandra et Toueg ont montré qu'il est possible de résoudre le problème du consensus dans des systèmes répartis asynchrones munis de détecteurs de défaillances non fiables satisfaisant certaines propriétés de *précision* (relative à la *vivacité*), limitant les fausses suspicions, et de *complétude* (relative à la *sûrté*), spécifiant la détection effective des processus défaillants.

La section 2.1 introduit le modèle du système réparti, les différentes propriétés caractérisant les détecteurs de défaillances non fiables, les huit classes de détecteurs de défaillances et le concept de réductibilité. Ensuite, la section 2.2 présente les propriétés du consensus et sa résolu-

tion en utilisant un détecteur de défaillances de la classe $\diamond\mathcal{S}$. La section 2.3 cite quelques travaux sur les détecteurs de défaillances. La section 2.4 examine un des détecteurs de défaillances utilisé dans la suite de nos travaux. Enfin, la section 2.5 conclue ce chapitre.

2.1 Modèle

Le modèle considéré se compose d'un système réparti *asynchrone* ou il n'existe aucune borne ni sur les délais de transfert d'un message ni sur le temps nécessaire pour l'exécution d'une étape. Le système consiste en un ensemble de n processus $\Pi = \{p_1, p_2 ..., p_n\}$ où chaque paire de processus est connectée par un canal de communication supposé fiable. En guise de clarté pour la présentation, l'existence d'une horloge globale virtuelle $\mathcal{T}$ est supposée. $\mathcal{T}$ est inaccessible par les processus, elle prend ses valeurs dans l'ensemble des entiers naturels.

2.1.1 Modèles de défaillances

Un processus peut être défaillant à cause d'un *crash* (c-à-d un arrêt prématuré). Le modèle de défaillances F est une fonction de $\mathcal{T}$ vers 2^{Π}, où $F(t)$ est l'ensemble des processus défaillants jusqu'à l'instant t. $crashed(F) = \bigcup_{t \in \mathcal{T}} F(t)$ représente l'ensemble des processus défaillants dans F, et $corrrect(F) = \Pi \setminus crashed(F)$ représente l'ensemble des processus corrects dans F. Seul le cas où il existe au moins un processus correct dans F est considéré, c'est-à-dire $correct(F) \neq \emptyset$. Si un processus subit une défaillance alors il le reste, c-à-d $\forall t \in \mathcal{T} : F(t) \subseteq F(t+1)$. Si $p \in crashed(F)$, p est dit défaillant dans F, et si $p \in correct(F)$, p est dit correct dans F.

2.1.2 Principe des détecteurs de défaillances non fiables

Chaque module de détection de défaillances fournit l'ensemble des processus qu'il suspecte d'être défaillants. L'historique H d'un détecteur de défaillances est une fonction de l'ensemble $\Pi \times \mathcal{T}$ vers l'ensemble 2^{Π} où $H(p, t)$ est la valeur du détecteur de défaillances du processus p à l'instant t dans H. Si $q \in H(p, t)$ alors p *suspecte* q à l'instant t dans H. Notons que deux processus distincts peuvent avoir deux listes de processus suspectés différentes, c'est-à-dire si $p \neq q$ alors $H(p, t) \neq H(q, t)$ est possible.

Un détecteur de défaillances $\mathcal{D}$ fournit des informations (peut-être erronées) sur le modèle de défaillances F pendant l'exécution du processus correspondant. Formellement, un détecteur de défaillances $\mathcal{D}$ est une fonction qui associe à chaque modèle de défaillances F un ensemble d'historiques de détecteurs de défaillances $\mathcal{D}(F)$. $\mathcal{D}(F)$ est l'ensemble de tous les historiques pouvant exister durant les exécutions avec l'ensemble des défaillances F et le détecteur de défaillances $\mathcal{D}$.

2.1.3 Propriétés de la détection de défaillances non fiables

Un algorithme est dit correct s'il satisfait les propriétés de sûreté et de vivacité [Gär99]. Chandra et Toueg définissent les détecteurs de défaillances selon deux propriétés. La *complétude* est la capacité de détection des processus défaillants dans F satisfaisant la sûreté. La *précision* limite les fausses suspicions et les corrige, satisfaisant à son tour la vivacité.

1. **Complétude** :
   – *Complétude forte* : il existe un instant après lequel un processus défaillant est suspecté d'une manière permanente par tout processus correct. D'une manière formelle, $\mathcal{D}$ satisfait la complétude forte si :

   $$\forall F,\, \forall H \in D(F),\, \exists t \in \mathcal{T},\, \forall p \in crashed(F),\, \forall q \in correct(F),\, \forall t' \geq t : p \in H(q,\, t')$$

   – *Complétude faible* : il existe un instant après lequel tout processus défaillant est suspecté d'une manière permanente par au moins un processus correct. D'une manière formelle, $\mathcal{D}$ satisfait la complétude faible si :

   $$\forall F,\, \forall H \in D(F),\, \exists t \in \mathcal{T},\, \forall p \in crashed(F),\, \exists q \in correct(F),\, \forall t' \geq t : p \in H(q,\, t')$$

   L'utilisation de la complétude seule n'est pas très utile. Considérons le détecteur de défaillances suivant : chaque processus suspecte d'une manière permanente tous les autres processus du système. Il est clair qu'un tel détecteur satisfait la propriété de complétude forte mais n'est pas utile car il ne fournit pas d'information sur les défaillances. Pour être utile, un détecteur de défaillances doit aussi satisfaire des propriétés de précision qui limitent les suspicions erronées.

2. **Précision** :
   – *Précision forte* : aucun processus n'est suspecté avant qu'il ne devienne défaillant. Formellement, $\mathcal{D}$ satisfait la précision forte si :

   $$\forall F,\, \forall H \in D(F),\, \forall t \in \mathcal{T},\, \forall p,\, q \in \Pi - F(t) : p \notin H(q,\, t)$$

   Puisque cette propriété est difficile à assurer (si ce n'est impossible) dans la pratique, considérons le cas où un processus correct p s'exécute très lentement. Celui-ci est toujours susceptible d'être suspecté par un autre processus correct q. Si le délai de garde de p expire dans le module de détection de défaillances d'un autre processus q sans qu'il ait reçu le message de présence de p, q insère p dans sa liste de suspects. Puisque la précision forte est souvent violée, une autre propriété est alors définie.

   – *Précision faible* : certains processus corrects ne sont jamais suspectés. Formellement,

$\mathcal{D}$ satisfait la précision faible si :

$$\forall F,\ \forall H \in D(F),\ \exists p \in correct(F),\ \forall t \in \mathcal{T},\ \forall q \in \Pi - F(t) : p \notin H(q,\ t)$$

De même, puisque cette propriété peut être difficile à assurer dans la pratique, deux autres propriétés sont introduites qui exigent l'existence d'un délai t à partir duquel les deux propriétés de précision précitées sont assurées.

- *Précision forte inévitable* : il existe un instant après lequel tout processus correct n'est pas suspecté par un autre processus correct. Formellement, $\mathcal{D}$ satisfait la précision forte inévitable si :

$$\forall F,\ \forall H \in D(F),\ \exists t \in \mathcal{T},\ \forall t' \geq t,\ \forall p,\ q \in correct(F) : p \notin H(q,\ t')$$

- *Précision faible inévitable* : il existe un instant après lequel il existe des processus qui ne sont pas suspectés par un processus correct. Formellement, $\mathcal{D}$ satisfait la précision faible inévitable si :

$$\forall F, \forall H \in D(F), \exists t \in \mathcal{T}, \exists p \in correct(F), \forall t' \geq t, \forall q \in correct(F) : p \notin H(q, t')$$

La précision forte inévitable et la précision faible inévitable sont appelées « *précision inévitable* », et la précision forte et la précision faible sont appelées « *précision perpétuelles* ».

2.1.4 Classes de détecteurs de défaillances non fiables

Il existe huit paires de classes de détecteurs de défaillances, obtenues en combinant une des deux propriétés de complétude avec une des quatre propriétés de précision. Leurs définitions et les notations correspondantes sont données dans le tableau 2.2. Un détecteur de défaillances est dit « *parfait* » s'il satisfait la complétude forte et la précision forte. L'ensemble de tels détecteurs de défaillances, nommé la « *classe des détecteurs de défaillances parfaits* » est noté par $\mathcal{P}$. Similairement, chaque paire de propriétés de complétude et de précision composent une classe de détecteurs de défaillances non fiables.

2.1.5 Réductibilité

La réductibilité[CT96] est un concept permettant de se focaliser sur quatre classes de détecteurs de défaillances plutôt que sur les huit classes de départ.

Un algorithme $T_{\mathcal{D} \longrightarrow \mathcal{D}'}$ nommé « *algorithme de réduction* », transforme un détecteur de défaillances $\mathcal{D}$ en un autre détecteur de défaillances $\mathcal{D}'$. Étant donné un algorithme de réduction $T_{\mathcal{D} \longrightarrow \mathcal{D}'}$, tout problème résolu avec le détecteur de défaillances $\mathcal{D}$ peut être résolu avec $\mathcal{D}'$. S'il

	Précision			
Complétude	Forte	Faible	Forte inévitable	Faible inévitable
Forte	*Parfait* $\mathcal{P}$	*Fort* $\mathcal{S}$	*Parfait inévitable* $\diamond\mathcal{P}$	*Fort inévitable* $\diamond\mathcal{S}$
Faible	$\mathcal{Q}$	*Faible* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Faible inévitable* $\diamond\mathcal{W}$

TAB. 2.2 – Classes des détecteurs de défaillances

existe un algorithme $T_{\mathcal{D}\longrightarrow\mathcal{D}'}$ qui transforme $\mathcal{D}$ en $\mathcal{D}'$, on écrit $\mathcal{D} \succeq \mathcal{D}'$ et on dit que $\mathcal{D}'$ est *réductible* à $\mathcal{D}$, on dit aussi que $\mathcal{D}'$ *est plus faible* que $\mathcal{D}$. Si $\mathcal{D} \succeq \mathcal{D}'$ et $\mathcal{D}' \succeq \mathcal{D}$, on écrit $\mathcal{D} \cong \mathcal{D}'$ et on dit que $\mathcal{D}$ et $\mathcal{D}'$ sont *équivalents*. Similairement, étant données deux classes de détecteurs de défaillances $\mathcal{C}$ et $\mathcal{C}'$, si pour chaque détecteur de défaillances $\mathcal{D} \in \mathcal{C}$, il existe un détecteur de défaillances $\mathcal{D}' \in \mathcal{C}'$ tel que $\mathcal{D} \succeq \mathcal{D}'$, on écrit $\mathcal{C} \succeq \mathcal{C}'$ et on dit que $\mathcal{C}'$ *est plus faible que* $\mathcal{C}$. En outre, si $\mathcal{C} \succeq \mathcal{C}'$, alors si un problème est solvable en utilisant $\mathcal{C}'$ il est aussi solvable en utilisant $\mathcal{C}$. Si $\mathcal{C} \succeq C'$ et $C' \succeq C$, on écrit $\mathcal{C} \cong \mathcal{C}'$ et on dit que $\mathcal{C}$ et $\mathcal{C}'$ sont *équivalentes*.

La figure 2.1 schématise les relations existant entre les classes de détecteurs de défaillances. Toutes les relations d'équivalence ou de réduction sont représentées respectivement par un trait plein ou une flèche pointillée. Les classes qui ne sont pas reliées sont incomparables. D'après le schéma, toutes les classes de complétude faible sont équivalentes aux classes de complétude forte. L'algorithme de réduction est donné dans la figure 2.2. Il suffit maintenant de résoudre le consensus pour les quatre classes de détecteurs de défaillances qui satisfont la complétude forte :

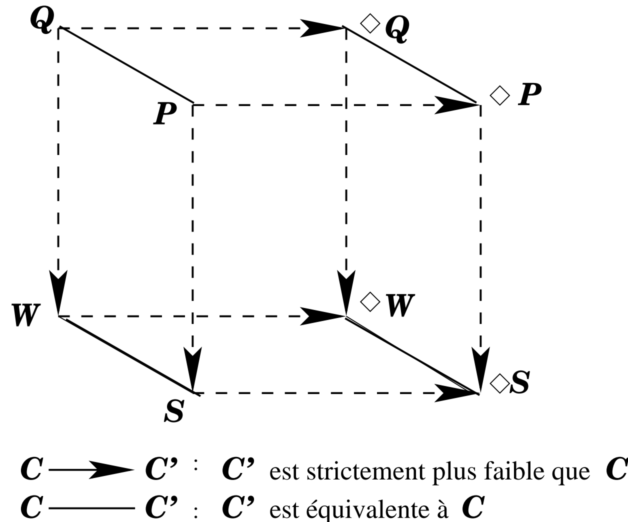

FIG. 2.1 – Relations entre les classes de détecteurs de défaillances

1. Every process p executes the following :
2. **Initialization :**
3. $output_p \leftarrow \emptyset$
4. **cobegin**
5. $\|$*Task 1 :* **repeat forever**
6. $suspects_p \leftarrow \mathcal{D}_p$ $\{p$ interroge son module de détection de défaillances $\mathcal{D}_p\}$
7. **send** $(p, suspects_p)$ **to all**
8. $\|$*Task 2 :* **when receive** $(q, suspects_q)$ from some q
9. $output_p \leftarrow (output_p \cup suspect_q) \setminus \{q\}$
10. **coend**

FIG. 2.2 – $T_{\mathcal{D}\longrightarrow\mathcal{D}'}$: de la complétude faible à la complétude forte

$\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$ et $\diamond\mathcal{S}$. [CT96] présente une résolution du consensus utilisant $\mathcal{S}$. Puisque $\mathcal{P} \succeq \mathcal{S}$, l'algorithme résout aussi le consensus en utilisant $\mathcal{P}$. Ils donnent aussi la résolution du consensus utilisant $\diamond\mathcal{S}$. Puisque $\diamond\mathcal{P} \succeq \diamond\mathcal{S}$, l'algorithme résout aussi le consensus en utilisant $\diamond\mathcal{P}$. Dans ce rapport, nous ne présentons que le consensus résolu en utilisant la classe $\diamond\mathcal{S}$, le plus faible pour résoudre le consensus [CT96].

2.2 Paradigme du consensus

Dans le paradigme du consensus, tous les processus corrects proposent une valeur et doivent prendre une décision irrévocable sur une valeur appartenant à l'ensemble des valeurs proposées [FLP85]. Le consensus est composé des deux primitives $proposer(v)$ et $décider(v)$, où v est une valeur extraite de l'ensemble des valeurs proposables par les processus. Quand un processus p exécute la primitive $proposer(v)$, on dit que le processus p *propose* la valeur v. De même, quand il exécute $décider(v)$, on dit qu'il *décide* la valeur v. Le paradigme du consensus est caractérisé par les quatre propriétés suivantes :

- *Terminaison* : chaque processus correct décide éventuellement une valeur.
- *Validité uniforme* : si un processus décide une valeur v alors v a été proposée par un processus.
- *Intégrité uniforme* : chaque processus décide au plus une fois.
- *Accord* : deux processus correct ne décident pas deux valeurs différentes.

La propriété de terminaison indique que tout processus correct doit décider une valeur au bout d'un temps fini. La propriété d'intégrité uniforme indique que cette valeur décidée est unique. La propriété de validité uniforme définit le domaine de définition de la valeur décidée. La propriété d'accord définit la sémantique du consensus.

2.2.1 Résolution du consensus avec $\diamond\mathcal{S}$

Nous présentons dans cette section une solution au problème du consensus en utilisant la classe de détecteur de défaillances non fiables $\diamond\mathcal{S}$. Cet algorithme présenté dans la figure 2.3 est repris dans le chapitre suivant. Le consensus ne peut être résolu en utilisant un détecteur de défaillances qui appartient à la classe $\diamond\mathcal{S}$ que si le nombre maximum de processus défaillants est strictement inférieur à la moitié du nombre de processus participant au consensus [CHT96]. Soit f le nombre maximum de processus qui peuvent être défaillants. Le système réparti asynchrone considéré satisfait la contrainte $f < \lceil \frac{n}{2} \rceil$, autrement dit, au moins $\lceil \frac{(n+1)}{2} \rceil$ processus sont corrects ($n = |\Pi|$ le nombre de processus).

Un détecteur de défaillances de classe $\diamond\mathcal{S}$ peut rajouter éventuellement tous les processus corrects à sa liste de suspects. Cependant, il existe un processus correct et un instant après lequel ce dernier n'est plus suspecté. L'algorithme présenté dans la figure 2.3 est basé sur le paradigme du *coordinateur tournant* ; il procède par tours successifs asynchrones. Durant le tour r, tous les processus savent que le processus coordinateur est le processus $c = (r \bmod n) + 1$. Durant ce tour, les processus communiquent uniquement avec c. Le nombre de tours qu'effectue un processus dépend des défaillances et du comportement des détecteurs de défaillances. À chaque fois qu'un processus devient le coordinateur, il tente de déterminer une valeur cohérente. S'il est correct et n'est suspecté par aucun processus, il décide une valeur et la diffuse aux autres processus.

Chaque tour de cet algorithme de consensus est divisé en quatre phases asynchrones.

1. Chaque processus p maintient une variable $estimate_p$ (initialisée à sa valeur initiale v_i) dans laquelle il mémorise sa vision présente de la valeur décidée. Durant cette première phase, chaque processus p envoie au coordinateur courant la valeur de sa variable $estimate_p$ estampillée du numéro du tour auquel elle appartient.
2. Durant la deuxième phase, c collecte $\lceil \frac{(n+1)}{2} \rceil$ estimations proposées par les processus. Il choisit une des estimations reçues dont l'estampille est la plus grande et met à jour $estimate_c$ qu'il envoie à tous les processus.
3. Durant la troisième phase, chaque processus p attend l'estimation du coordinateur. Deux cas peuvent se présenter : le processus suspecte le coordinateur courant d'être défaillant (en consultant son détecteur de défaillances) ou bien il reçoit l'estimation du coordinateur. Dans le premier cas, il envoie un acquittement négatif au coordinateur courant et passe au tour suivant, c'est à dire au tour $r+1$. Dans le second cas, il envoie un acquittement positif au coordinateur et met à jour sa variable $estimate_c$ à la valeur de l'estimation reçue du coordinateur.
4. La dernière phase est effectuée par le coordinateur. Il attend une majorité d'acquittements. Deux cas peuvent se présenter : tous les acquittements reçus sont positifs ou bien au moins un message d'acquittement est négatif. Dans le premier cas, la majorité des processus a adopté l'estimation du coordinateur. Par conséquent, il verrouille la valeur de sa variable $estimate_c$ et diffuse d'une manière fiable la valeur de décision. Dans le deuxième cas, au

moins un des processus a suspecté le coordinateur. Ce dernier passe au tour suivant sans décider.

Enfin, lorsqu'un processus reçoit un message de décision avec une valeur v, il décide cette valeur et termine.

2.3 Travaux connexes

Le concept de détecteur de défaillances non fiable introduit par Chandra et Toueg a suscité une littérature abondante. Il y a d'une part les travaux d'aspect théorique qui portent leur intérêt sur l'étude des classes de détecteurs de défaillances, leur possibilité d'implémentation dans des environnements complètement asynchrones ou partiellement synchrones, et la détermination des classes de détecteurs de défaillances les plus faibles pour résoudre certains paradigmes. D'autre part, les travaux d'aspect plus pratique portent sur la qualité de service des détecteurs de défaillances et certaines architectures ou protocoles.

La sous-section 2.3.1 présente quelques travaux d'aspect théorique, tandis que la sous-section 2.3.2 présente quelques travaux d'aspect pratique.

2.3.1 Aspects théoriques

[CHT96] détermine l'information nécessaire et suffisante sur les défaillances pour résoudre le problème du consensus dans un système réparti asynchrone sujet aux *crashs*. [CT96] définissent $\diamond\mathcal{W}$ comme étant le détecteur de défaillances le plus faible pour résoudre le consensus dans un système réparti asynchrone avec une majorité de processus corrects. [CHT96] démontre que si un détecteur de défaillances fournit au moins autant d'information que $\diamond\mathcal{W}$, il peut résoudre le consensus. Donc, tout algorithme de détection de défaillances réductible à un algorithme de détection de défaillances qui appartient à la classe $\diamond\mathcal{W}$ est suffisant pour résoudre le consensus.

Entre le système asynchrone et le système synchrone, [DLP$^+$86] définit 32 modèles partiellement synchrones. [DLS88] généralise les systèmes partiellement synchrones en deux modèles. Dans le premier modèle $\mathcal{M}_1$, il existe des bornes dans le délai de transmission des messages et la vitesse d'exécution des processus, mais ces bornes ne sont pas connues. Dans le modèle $\mathcal{M}_2$, les bornes existent et sont connues, seulement elles ne sont valables qu'après une durée de stabilisation globale GST inconnue (mais bornée). Il est possible d'implémenter un détecteur de défaillances parfait inévitable $\mathcal{D} \in \diamond\mathcal{P}$ dans l'un des deux modèles [CT96, LFA99]. [CT96] définit un troisième modèle partiellement synchrone plus faible $\mathcal{M}_3$. Dans ce modèle les bornes existent mais sont inconnues et ne sont valables qu'après un temps GST inconnu (mais borné). Une résolution du consensus pour $\mathcal{M}_3$ est aussi une solution pour $\mathcal{M}_1$ et $\mathcal{M}_2$.

[LFA02] montre la possibilité d'implémenter les différentes classes de détecteurs de défaillances dans les systèmes partiellement synchrones. Se basant sur les huit classes de détecteurs de défaillances (*Cf.* tableau 2.2), les algorithmes qui implémentent les classes $\diamond\mathcal{P}$, $\diamond Q$, $\diamond\mathcal{S}$ et $\diamond\mathcal{W}$ dans un système partiellement asynchrone existent déjà et sont présentés dans [CT96, LFA99]. Par contre, [LFA02] démontre qu'aucune des quatre classes restantes ($\mathcal{P}$, $\mathcal{Q}$, $\mathcal{S}$

1. For every process p :
2. To execute propose(v_p) :
3. $estimate_p \leftarrow v_p$
4. $state_p \leftarrow undecided$
5. $r_p \leftarrow 0$
6. $ts_p \leftarrow 0$
7. **while** $state_p = undecided$
8. $r_p \leftarrow r_p + 1$
9. $c_p \leftarrow (r_p \bmod n)+1$
10. **Phase 1 :**
11. **send**$(p, r_p, estimate_p, ts_p)$ to c_p
12. **Phase 2 :**
13. **if** $p = c_p$ **then**
14. **wait until** [**for** $\lceil \frac{(n+1)}{2} \rceil$ processes q : received $(q, r_p, estimate_q, ts_q)$ from q]
15. **if** [for $\lceil \frac{(n+1)}{2} \rceil$ processes q : received$(q, r_p, estimate_q, ts_q)$] **then**
16. $msgs[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) | p$ received$(q, r_p, estimate_q, ts_q)$ from $q\}$
17. $t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
18. $estimate_p \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
19. **send** $(p, r_p, estimate_p)$ to all
20. **Phase 3 :**
21. **wait until** [received $(c_p, r_p, estimate_{c_p})$ from c_p **or** $\mathcal{D}_p$ $suspect$ c_p]
22. **if** [received $(c_p, r_p, estimate_{c_p})$ from c_p] **then**
23. $estimate_p \leftarrow estimate_{c_p}$
24. $ts_p \leftarrow r_p$
25. **send** (p, r_p, ack)
26. **else send** $(p, r_p, nack)$to c_p
27. **Phase 4 :**
28. **if** $p = c_p$ **then**
29. **wait until** [for $\lceil \frac{(n+1)}{2} \rceil$ processes q : received (q, r_p, ack) **or** $(q, r_p, nack)$]
30. **if** [for $\lceil \frac{(n+1)}{2} \rceil$ processes q : received (q, r_p, ack)] **then broadcast** $(q, r_p, estimate_p, decide)$
31.
32. **wait until** [**upon** R-deliver $(q, r_p, estimate_q, decide)$]
33. **if** R-deliver $(q, r_p, estimate_q, decide)$ **and** $state_p = undecided$ **then**
34. **decide(**$estimate_q$**)**
35. $state_p \leftarrow decided$

FIG. 2.3 – Algorithme du consensus avec $\Diamond\mathcal{S}$

et $\mathcal{W}$) ne peut être implémentée dans aucun des trois modèles de systèmes partiellement synchrones sujets aux *crashs*.

2.3.2 Aspects pratiques

[CTA01] s'intéresse à l'aspect pratique des détecteurs de défaillances et plus spécialement à la mention *inévitable*. Pour les applications ayant des contraintes temporelles, les détecteurs de défaillances non fiables ne sont pas suffisants. Prenons comme exemple un détecteur de défaillances qui commence à suspecter un processus défaillant après une heure d'exécution. Il peut être utilisé pour résoudre le consensus mais est inutilisable pour une application qui a besoin de résoudre plusieurs instances du consensus par minute. Une application qui a des contraintes temporelles requiert un détecteur de défaillances qui fournit une qualité de service. [CTA01] étudie la QoS des détecteurs de défaillances dans un système où le temps de transmission et les pertes de messages suivent une probabilité distribuée. La QoS englobe deux métriques : la première est la vitesse de détection d'une défaillance effective, la deuxième est la capacité du détecteur de défaillances à éviter les fausses suspicions. Les auteurs montrent les paramètres utilisés dans leurs algorithmes et la manière de les calculer, et cela même si les propriétés probabilistes du système sont inconnues. D'autres références reprennent les travaux de [CTA01] pour leurs algorithmes de détection de défaillances. [Bal01] met en œuvre un service de détection de défaillances dans un système réparti asynchrone sur une plate-forme CORBA. [BMS02] montre comment implémenter et adapter dynamiquement le détecteur de défaillances selon le temps de détection, le temps entre deux erreurs consécutives et le temps de correction des erreurs. L'algorithme proposé est réductible à la classe $\diamond\mathcal{S}$ dans un modèle partiellement synchrone selon le modèle $\mathcal{M}_3$. Le détecteur de défaillances est structuré en deux couches. La première couche fournit une estimation des temps d'arrivée des messages pour optimiser le temps de détection. La deuxième couche adapte le service de détection fourni par la première couche aux besoins de l'application.

[ACT97] utilise le concept des détecteurs de défaillances pour résoudre le problème de la communication fiable avec des algorithmes « *silencieux* » (en anglais « *quiescent* », c-à-d, des algorithmes qui arrêtent inévitablement d'envoyer des messages) dans un système réparti asynchrone sujet à la fois aux *crashs* de processus et à la rupture des liens de communications. Il prouve d'abord qu'il est impossible de résoudre le problème sans utiliser les détecteurs de défaillances. Il montre ensuite comment le résoudre en utilisant un nouveau détecteur de défaillances nommé « *battements de cœur* » $\mathcal{HB}$. Contrairement à d'autres détecteurs de défaillances, celui-ci est implémentable et n'utilise pas de délai de garde (en anglais « *timeout* »). Ces résultats ont une large applicabilité. Ils peuvent être utilisés pour transformer des algorithmes existants qui tolèrent uniquement le *crash* de processus en des algorithmes «*silencieux*» qui tolèrent aussi la rupture des liens. Ces algorithmes peuvent être appliqués au consensus, à la diffusion atomique et à la validation atomique. [ACT99] montre comment résoudre le consensus « *silencieux* » dans un réseau partitionnable avec $\diamond\mathcal{S}$ et un détecteur de défaillances $\mathcal{HB}$. Ils utilisent pour cela les primitives de communication introduites dans [ACT97] qui utilisent $\mathcal{HB}$ pour réaliser une communication fiable « *silencieux* ». Il modifient aussi la définition de la classe $\diamond\mathcal{S}$ en $\diamond\mathcal{S}_{LP}$ pour une résolution du consensus dans la partition primaire contenant la majorité des processus corrects. La simplicité de cet algorithme de détection de défaillances a suscité notre

intérêt, et nous l'utilisons dans notre travail. Il est présenté plus en détail dans la section suivante.

2.4 Détecteur de défaillances dit battements de cœur

[ACT97] présente un détecteur de défaillances implémenté sans délai de garde dans un système sujet aux défaillances des processus et des liens. Ce détecteur de défaillances nommé « battements de cœur » et noté $\mathcal{HB}$. En règle générale, le module de détection de défaillances $\mathcal{HB}$ retourne un vecteur de compteurs, un compteur par voisin. Si le voisin q n'est pas défaillant, sa valeur du compteur croît indéfiniment. Si q *crashe*, son compteur cesse de croître. Chaque processus envoie périodiquement un message « *je-suis-vivant* » (« *heartbeat* ») et tout processus recevant les battements de cœur incrémente le compteur correspondant.

Avant de présenter $\mathcal{HB}$ dans la sous-section 2.4.2, la sous-section 2.4.1 introduit d'abord le modèle utilisé et énonce les propriétés du détecteur de défaillances.

2.4.1 Modèle et propriétés

Toutes les hypothèses considérées dans la section 2.1 sont valables dans ce modèle, excepté qu'il n'y a aucune supposition que le réseau est complètement connecté ou que les liens sont bidirectionnels. De plus, le système est sujet à deux types de défaillances : *crash* de processus et *crash* de liens.

Le détecteur de défaillances « *battements de cœur* » $\mathcal{HB}$ est diffèrent des détecteurs de défaillances définis dans [CT96, CTA01, BMS02]. Il possède les propriétés suivantes. La sortie de $\mathcal{D}$ dans chaque processus p est un tableau de compteurs $((p_1, n_1), (p_2, n_2), ..., (p_k, n_k))$ où $p_1, p_2, ... p_k$ sont les voisins de p et chaque n_j est un entier positif. D'une manière intuitive, n_j augmente tant que p_j n'est pas défaillant, et cesse d'augmenter dans le cas contraire. On dit que n_j est *la valeur de battements de cœur du processus* p_j *dans* p. L'historique de $\mathcal{D}$ dans p à l'instant t $H(p, t)$ est un vecteur d'index $\{p_1, p_2, ..., p_k\}$. Ainsi, $H(p, t)[p_j]$ est la valeur de n_j du compteur de p_j dans p. La *séquence de battements de cœur de* p_j *dans* p est la séquence de valeurs des battements de coeur de p_j dans p en fonction du temps. $\mathcal{D}$ satisfait les propriétés suivantes :

- $\mathcal{HB}-$*Complétude* : dans chaque processus correct, la séquence de *battements de cœur* de tout processus défaillant est borné. Formellement :

$$\forall F,\ \forall H \in \mathcal{D}(F), \forall p \in correct(F),\ \forall q \in crashed(F) \cap neighbor(p),$$
$$\exists K \in N,\ \forall t \in \mathcal{T}\ : H(p,\ t)[q] \leq K$$

- $\mathcal{HB}-$*Précision* :
  - dans chaque processus, la séquence de *battements de cœur* de chaque processus voisin ne diminue jamais. Formellement :

$$\forall F,\ \forall H \in \mathcal{D}(F), \forall p \in \Pi,\ \forall q \in neighbor(p),\ \forall t \in \mathcal{T} : H(p,\ t)[q] \leq H(p,\ t+1)[q]$$

– dans chaque processus correct, la séquence de *battements de cœur* de chaque processus voisin correct est non borné. Formellement :

$$\forall F,\ \forall H \in \mathcal{D}(F), \forall p \in correct(F),\ \forall q \in\ correct(p) \cap neighbor(p),$$
$$\forall K \in N,\ \exists t \in \mathcal{T}\ : H(p,\ t)[q] > K$$

2.4.2 Implémentation de $\mathcal{HB}$ pour des réseaux simples

Il existe trois implémentations de $\mathcal{HB}$. [ACT97] présente deux algorithmes : un algorithme destiné aux réseaux simples, où tous les liens du réseau sont bidirectionnels et non défaillants, et un algorithme destiné aux réseaux généraux, où les liens peuvent être unidirectionnels et/ou défaillants. [ACT99] présente aussi une implémentation de $\mathcal{HB}$ pour un réseau partitionnable, où les liens sont unidirectionnels et leur défaillance peut créer des partitions permanentes dans le réseau. Il donne aussi la résolution d'un consensus si la partition primaire contient une majorité des processus corrects.

Le détecteur de défaillances présenté dans la figure 2.4 implémente l'algorithme de $\mathcal{HB}$ pour les réseaux simples. Le choix de cet algorithme repose sur le fait qu'il est le plus proche du modèle utilisé pour implémenter le consensus dans [CT96]. En effet, il est facile de généraliser la définition de $\mathcal{HB}$ pour correspondre au modèle présenté dans [CT96], en retournant dans chaque processus p les battements de cœur pour tous les processus dans le système. Tout processus p exécute deux tâches concurrentes. Dans la première tâche, p envoie périodiquement un message HEARTBEAT à tous ses voisins. La seconde tâche manipule la réception des messages HEARTBEAT. Quand un tel message est reçu de q, p incrémente la valeur du battement de cœur de q.

$\mathcal{HB}$ n'utilise pas de délai de garde dans pour déterminer si un processus est défaillant. $\mathcal{HB}$ comptabilise juste le nombre de *battements de cœur* reçu de chaque processus et retourne le tableau de compteurs sans aucune autre interprétation.

```
1.  For every process p :
2.  Initialization :
3.      for all q ∈ neighbor(p) do D_p[q] ← 0        {D_p[q] nombre de battements de cœur de q reçus par p}
4.  cobegin
5.      ||Task 1 :
6.          repeat periodically
7.              for all q ∈ neighbor(p) do send (HEARTBEAT) to q
8.      || Task 2 :
9.          upon receive (HEARTBEAT) from q do
10.             D_p[q] ← D_p[q] + 1
11. coend
```

FIG. 2.4 – Implémentation de $\mathcal{HB}$ pour un réseau simple

2.5 Conclusion

Dans ce chapitre, nous avons présenté le concept de tolérance au fautes dans les systèmes répartis, le paradigme du consensus qui représente la base des solutions de tolérance aux fautes, le résultat d'impossibilité de résolution du consensus dans un système réparti asynchrone sujet aux *crashs* de processus, puis la solution qui s'appuie sur un module de détection de défaillances non fiable. Nous avons ensuite cité quelques résultats et propriétés sur les différentes classes de détecteurs de défaillances non fiables. Nous nous sommes intéressés enfin à l'un des détecteurs de défaillances nommé $\mathcal{HB}$. Dans le chapitre suivant, nous rapprochons les détecteurs de connectivité vus dans le chapitre 1 et les détecteurs de défaillances non fiables, en l'occurrence $\mathcal{HB}$.

Chapitre 3

Détecteurs de connectivité, de déconnexions et de défaillances

Après avoir introduit les deux mécanismes principaux autour desquels s'articule le sujet du stage de DEA, nous avons entrepris de savoir si les deux types de détecteurs sont comparables ou complémentaires en répondant à quelques interrogations. Pouvons-nous compléter les fonctionnalités de l'un des types de détecteurs par celles de l'autre ? L'un peut-il être utilisé au profit de l'autre ? Existe-il des possibilités d'avoir une utilisation qui associe les détecteurs dans une seule architecture ou une seule application ? Ou sont-ils complètement différents ?

Nous présentons dans ce chapitre les approches proposées dans lesquelles nous essayons de donner des utilisations possibles pour répondre aux questions posées ci-dessus. Nous présentons donc principalement trois approches. Dans la première approche nous utilisons le détecteur de défaillances au profit de la gestion de déconnexions. Dans la deuxième approche, nous présentons l'utilisation de la gestion de déconnexion, précisément la capacité de préventions des déconnexions au profit de la tolérance aux fautes. Enfin, dans la troisième approche, nous ajoutons la détection de déconnexions dans le détecteur de défaillances.

La première section de ce chapitre énonce les trois approches proposées. Les trois sections qui suivent les détaillent. Enfin, la dernière section conclue ce chapitre.

3.1 Approches proposées

Les deux mécanismes de détections étudiés dans les deux précédents chapitres expliquent ce qu'est une déconnexion et ce qu'est une défaillance, mettant ainsi en évidence qu'une déconnexion n'est pas une défaillance. Par conséquent, il faut différencier le traitement d'une déconnexion de celui d'une défaillance.

Prenant comme exemple un système réparti asynchrone composé à la fois de machines fixes et mobiles qui participent à une application répartie. Une déconnexion d'une des machines mobiles est considérée comme une défaillance, car après une certaine durée les autres machines ne reçoivent plus de messages de la part de la machine déconnectée. Cependant, grâce à la gestion de déconnexions, la déconnexion d'un hôte mobile, qu'elle soit volontaire ou involontaire, n'est

plus considérée comme une défaillance mais reconnue comme une déconnexion.

Nous suggérons trois approches pour associer et relier les deux types de détecteurs. Dans la première approche, nous mettons le détecteur de défaillances au profit de la gestion de déconnexions. Pour cela, nous proposons une architecture où le détecteur de connectivité de *Domint* utilise un détecteur de défaillances. Cette approche a pour but de compléter la vue locale des détecteurs de connectivité d'une vue globale fournie par les détecteurs de défaillances.

La deuxième approche est l'utilisation des détecteurs de connectivité dans le contexte de la tolérance aux fautes, particulièrement dans le consensus. Pour cela, nous concevons d'abord un détecteur de déconnexions à partir d'un détecteur de connectivité pour obtenir une vue globale des déconnexions. Nous intégrons ensuite le détecteur de déconnexions dans le consensus. Le but de cette approche est d'exclure les processus déconnectés du consensus.

La troisième approche est une optimisation de la deuxième approche. Dans cette approche, le détecteur de défaillances utilise un détecteur de connectivité pour gérer des informations de déconnexions pour exclure à son tour les processus déconnectés de sa liste de diffusion.

3.2 Détecteurs de défaillances pour la gestion de déconnexion

En premier, la sous-section 3.2.1 développe les motivations de cette première approche. Ensuite, la sous-section 3.2.2 présente les raisons du choix de l'algorithme de détection de défaillances, et enfin la sous-section 3.2.3 montre l'architecture proposée.

3.2.1 Motivations

Le gestionnaire de connectivité CM présenté dans le chapitre 1 manipule une connexion logique entre un client sur le terminal mobile et un objet distant sur le terminal fixe. Il peut aussi manipuler une connexion logique où le client et l'objet distant se trouvent sur des terminaux mobiles. Les informations que fournit le mécanisme d'hystérésis ne concernent que les ressources locales. Elles n'offrent donc qu'une visibilité locale à la machine. En d'autres termes, la machine locale ne connaît que ses propres informations de connectivité et n'a aucune connaissance de l'état de connectivité des hôtes distants. Cette lacune peut ne pas être très gênante si l'objet distant se trouve sur un terminal fixe car ce dernier est correctement connecté en permanence. Cependant, si l'objet distant est sur un terminal mobile, il est important de savoir s'il dispose à son tour d'une bonne connectivité. Par ailleurs, si des hôtes mobiles communiquent, il est intéressant qu'une machine mobile ne dispose pas que des informations liées à sa propre connectivité mais aussi des informations sur la connectivité des autres machines mobiles qui communiquent avec elle. De cette manière, la visibilité d'un hôte mobile englobe une connectivité de bout-en-bout avec tous les hôtes qui lui sont connectés. Cette visibilité globale peut être fournie en utilisant un module de détection de défaillances. L'ajout d'un module de détection de défaillances à un hôte mobile permet d'avoir une visibilité globale à travers l'échange de battements de cœur entre les hôtes connectés. Ces informations combinées aux informations obtenues grâce aux hystérésis liés à la charge de la batterie et à la largeur de la bande passante peuvent refléter plusieurs situations.

La périodicité des envois de battements de cœur peut aussi être configurée suivant la disponibilité des ressources locales, reflétant ainsi l'utilisabilité des ressources locales d'un hôte à un autre hôte distant. Prenons l'exemple où un hôte h_1 ouvre une communication avec un autre hôte mobile distant h_2. L'hôte h_1 commence par insérer l'hôte h_2 dans la liste de son détecteur de défaillances. Périodiquement, h_1 envoie des messages de présence à h_2 . Si h_1 dispose d'une bonne largeur de bande passante et d'une charge de batterie suffisante, il configure la périodicité de ses battements de cœur en des envois relativement fréquents. À la réception des battements de cœur de h_1, h_2 l'insert à son tour dans son détecteur de défaillances et commence à envoyer des battements de cœur à h_1. Si h_2 ne dispose pas d'une bonne connectivité, il envoie des battements de cœur espacés dans le temps à h_1 . h_1 interprète l'état de connectivité de h_2 en comparant la périodicité des envois de battements de cœur et la fréquence de réception. Si h_1 ne reçoit aucun battement de cœur de h_2 après une certaine durée, il conclue que h_2 est déconnecté, il arrête donc de lui envoyer des battements de cœur et le supprime de la liste du détecteur de défaillances. De son côté, h_2 vide sa liste de détecteur de défaillances dès qu'il rentre dans le mode déconnecté. Si par contre la connectivité est favorable des deux cotés, à la fin des traitements de h_1, il enlève h_2 de son détecteur de défaillances, après une certaine durée, h_2 constate que h_1 n'envoie plus de battements de cœur, h_2 enlève h_1 à son tour de son détecteur de défaillances.

Si le délai d'attente de h_1 dans h_2 expire sans que h_1 soit déconnecté ou ait fini son traitement, h_2 croit par erreur que h_1 est déconnecté ou a terminé son traitement. h_2 arrête les envois de battements de cœur et enlève h_1 de sa liste de diffusion du détecteur de défaillances. Cette fausse détection n'altère aucunement la correction de cet architecture, puisqu'elle est corrigée dès que les messages de h_1 arrivent à nouveau à h_2, celui-ci remet h_1 dans sa liste de diffusion et recommence à lui envoyer à son tour des battements de cœur, tout en augmentant les délais de réception des messages.

Le détecteur de connectivité configure le détecteur de défaillances suivant son mode de connectivité : le mode connecté est reflété par un envoi de battements de cœur à la fréquence f_c, le mode partiellement connecté est reflété par un envoi de battements de cœur à la fréquence minimale f_p, et dans le mode déconnecté, aucun battement de cœur n'est envoyé : $f_d = 0$. Si la fréquence de réception f_r au niveau de l'hôte distant est proche ou égale à f_c, cela signifie que l'hôte qui les envoie est en mode connecté et que les transmissions au niveau du réseau sont stables. Si $f_p < f_r \leq f_c$ mais f_r est plus proche de f_p que de f_c alors l'hôte qui les envoie est en mode connecté mais les transmissions du réseau sont plus perturbées. $f_r \leq f_p$ signifie que l'hôte distant est en mode partiellement connecté. Enfin, f_r pratiquement nulle signifie que l'hôte distant est en mode connecté ou partiellement connecté mais le réseau est fortement chargé. Si l'hôte h_1 sait que h_2 est en mode partiellement connecté, il peut décider de préparer la déconnexion de h_2 en téléchargeant les données nécessaires à son travail en mode déconnecté grâce à la fonctionnalité fournie par *Domint*.

En résumé, une déconnexion d'un hôte distant est traduite par une fréquence f_r nulle positionnées en dessous de la valeur *lowDown*. Une fréquence significative des battements de cœur supérieur ou égale à f_c positionnés au dessus de la valeur *highUp* indique le mode connecté de l'hôte distant. Une variation moins fréquente positionnée entre la valeur *highDown* et *lowUp* indique le mode partiellement connecté de l'hôte distant.

Les valeurs des fréquences d'envois de battements de cœur f_c, f_p et f_d sont configurables.

Elles sont négociées connexion logique par connexion logique au niveau de chaque hôte mobile, et sont adaptées à la disponibilité des ressources locales.

Remarque : la connaissance du modèle de système : synchrone, asynchrone ou partiellement synchrone, permet d'utiliser des informations supplémentaires sur les délais de transmission, pour obtenir plus de précision concernant la détermination des ressources locales d'un hôte distant. Dans un système synchrone où la vitesse d'exécution et les délais de transmission des messages sont connus, la fréquence de réception f_r correspond exactement à la fréquence d'émission. Les hôtes peuvent mutuellement déterminer leur mode de connectivité avec précision. La fin d'un traitement ou d'une déconnexion est facilement détectées. Dans un système partiellement synchrone, nous pouvons utiliser les caractéristiques du modèle pour interpréter la fréquence de réception f_r. Cette fréquence de réception peut ne pas correspondre exactement aux fréquences d'émission, mais l'étude des caractéristiques du système peuvent déterminer avec plus de précision le mode de connexion d'un hôte distant. La fin d'un traitement ou une déconnexion sont détectées après un certain délai calculé suivant les caractéristiques du système (Il existe 32 modèles de systèmes partiellement synchrones [DLP+86]). Enfin, dans un système asynchrone où la vitesse d'exécution et les délais de transmission ne son pas bornés, la fréquence de réception peut ne pas correspondre à la fréquence d'émission. Par exemple, comme signalé ci-avant, une fréquence très faible peut correspondre à un mode connecté ou partiellement connecté. La fin d'un traitement ou une déconnexion sont détectées après une certaine durée du dernier battement de cœur reçu. Cette durée est adaptée à la fréquence de réception des messages. Si la détection s'avère erronée, elle est augmentée pour éviter les fausses détections.

3.2.2 Choix du détecteur de défaillances $\mathcal{HB}$

Afin de mettre en œuvre l'utilisation des détecteurs de défaillances dans la gestion de déconnexion, nous avons choisi, parmi les détecteurs de défaillances étudiés au cours de ce stage, le détecteur de défaillances présenté dans la section 2.4 nommé « *heartbeat* » et noté $\mathcal{HB}$ [ACT97]. Le choix de cet algorithme repose sur deux raisons principales. La première raison est que $\mathcal{HB}$ n'utilise pas de délai de garde. La seconde et la plus importante pour cette approche est qu'il manipule un tableau de compteurs au lieu d'une liste de suspects. Il compte juste le nombre total de battements de cœur reçus des autres processus et retourne ce tableau de compteurs sans aucun autre traitement ou interprétation.

Le gestionnaire de connectivité configure la périodicité des envois de battements de cœur suivant les changements de modes. Si l'hôte est en mode connecté, CM met la périodicité d'envois de battements de cœur à f_c. Si l'hôte est en mode partiellement connecté, CM affecte la périodicité à f_p. Enfin, si l'hôte est dans le mode déconnecté, aucun battement de cœur n'est envoyé, la fréquence f_d est nulle. Périodiquement, le gestionnaire de connectivité interroge le détecteur de défaillances qui retourne un vecteur de compteurs associé aux hôtes distants qui lui sont connectés. Le gestionnaire de connectivité calcule alors la fréquence de réception des battements de cœur et les interprète en les positionnant sur un hystérésis qui cette fois-ci ne reflète pas les ressources locales mais des ressources distantes. L'algorithme $\mathcal{HB}$ présenté dans la figure 2.4 manipule pour chaque processus p un ensemble de voisins. Dans notre configuration un processus est assimilé à un hôte mobile.

3.2.3 Architecture

Dans cet architecture, nous ajoutons un module de détection de défaillances dans l'architecture déjà existante *Domint* présentée dans la section 1.2. Le détecteur de défaillances est implémenté comme un service CORBA. Il est démarré et configuré par *Domint*. Ce service possède une interface qui implémente principalement trois méthodes : une méthode d'abonnement des objets distants qui ajoute l'hôte distant à l'ensemble des hôtes du détecteur de défaillances, une méthode de désabonnement de ces objets distants qui supprime un hôte de l'ensemble des hôtes du détecteur de défaillances et une méthode d'interrogation qui retourne le vecteur des compteurs au gestionnaire de connectivité.

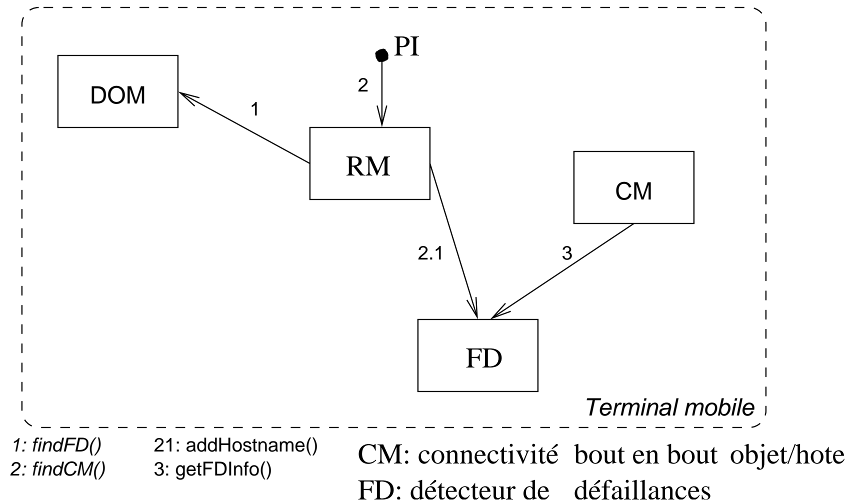

FIG. 3.1 – Détecteur de connectivité utilisant le détecteur de défaillances

La figure 3.1 ne montre que les éléments qui interagissent directement avec le détecteur de défaillances FD. Le fonctionnement des autres éléments reste inchangé. Les collaborations nouvelles sont les suivantes. Au démarrage de l'application, le gestionnaire de ressources RM interroge DOM une unique fois pour obtenir la référence du détecteur de défaillances (1). Lors de la première requête vers un objet distant, en plus des opérations effectuées dans l'architecture d'origine, RM ajoute l'hôte distant à la liste des hôtes maintenus par le détecteur de défaillances via son interface et sa méthode d'abonnement d'objets (2.1). Dès que l'hôte distant est ajouté à la liste du détecteur de défaillances, il est inséré dans la liste de diffusion de celui-ci. Il commence alors à émettre les messages de battements de cœur. À son tour, l'hôte distant l'ajoute à la liste de son détecteur de défaillances dès réception de battements de cœur d'un nouvel hôte qui ne figure pas dans sa liste. Périodiquement, le gestionnaire de connectivité CM interroge le détecteur de défaillances qui retourne la valeur du compteur du battements de cœur de l'hôte distant (3). CM calcule la variation de réception des battements de cœur $\Delta\mathcal{D}_p$ et le positionne alors dans un hystérésis (voir figure 3.2). Une méthode de désabonnement disponible dans l'interface du service du détecteur de défaillances est utilisée pour retirer l'hôte distant de sa liste des hôtes.

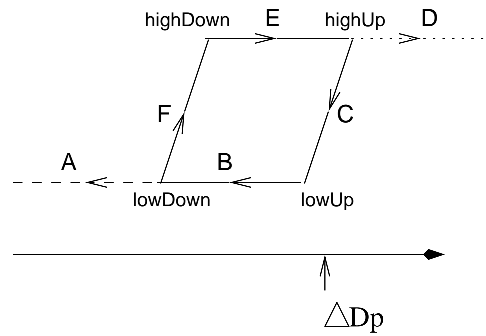

FIG. 3.2 – Fréquence des battements de cœur projetée sur un hystérésis

3.3 Détecteurs de connectivité pour la tolérance aux fautes

En premier, la sous-section **??** développe les motivations de cette seconde approche. La sous-section 3.3.2 illustre les différentes architectures pouvant associées l'algorithme du consensus aux deux types de détecteurs. La sous-section 3.3.3 énonce un détecteur de déconnexions réparti, module nécessaire à la réalisation de l'une des architectures. La sous-section 3.3.4 présente un algorithme du consensus utilisant les deux détecteurs.sous-section 3.4 présente un algorithme de détecteur de défaillances qui optimise l'utilisation séparée des deux détecteurs et prend en compte les déconnexions des processus. Enfin, la sous-section **??** présente un exemple d'application qui montre une manière d'utilisation de ces nouveaux algorithmes.

3.3.1 Motivations

Ordinairement, dans une application répartie localisée sur plusieurs hôtes mobiles et fixes, un hôte est considéré défaillant à cause soit d'un *crash* soit d'une déconnexion. Quand un hôte mobile est défaillant, son état volatil est perdu. Jusque là, il n'existe que la distinction entre une défaillance par *crash* et la déconnexion volontaire. Cette dernière peut être traitée comme une interruption de service planifiée, anticipée et préparée. La déconnexion involontaire continue à être traitée comme une défaillance. Cependant, l'existence de détecteurs de connectivité offre la possibilité de prévoir les déconnexions involontaires, donc de les anticiper et de les préparer.

L'ajout du détecteur de connectivité permet d'avoir l'information liée au contexte d'un nœud mobile tel que le niveau de batterie, le pourcentage de la bande passante et sa capacité à maintenir une connexion vers une autre machine. La machine mobile peut prévoir une déconnexion proche et ainsi disposer de suffisamment de temps pour effectuer des traitements. Suivant les besoins de l'application, il est possible de transférer les données nécessaires pour travailler en mode déconnecté, de charger un mandataire pour effectuer certaines opérations et aussi d'avertir les autres participants de la prochaine déconnexion. Les autres nœuds avertis d'une déconnexion d'un hôte mobile s'abstiennent de lui envoyer d'autres messages et ce jusqu'au prochain avertissement d'une reconnexion suivie d'une réconciliation. La figure 3.3 présente un exemple d'application qui arrête d'envoyer des messages aux hôtes déconnectés.

Afin d'avoir toutes les possibilités d'utilisation des détecteurs de connectivité et des détecteurs de défaillances au profit du consensus, nous proposons plusieurs architectures. Nous donnons ensuite des algorithmes de base pour les architectures les plus intéressantes.

3.3.2 Consensus et détecteurs de défaillances et de déconnexions

Le principe du paradigme du consensus est d'avoir une décision unanime des processus dépendant de valeurs initiales, même en présence de défaillances [FLP85]. L'algorithme du consensus utilise un détecteur de défaillances non fiable comme un oracle qui l'informe sur les processus suspectés d'être défaillants. Un processus correct n'attend pas les acquittements des processus suspectés de défaillances. Cependant, si un processus arrive à prendre une décision, il la diffuse à tous les processus participants au consensus, aussi bien aux processus corrects qu'aux processus suspectés d'être défaillants. Les diffusions aux processus défaillants continuent malgré les suspicions, car celles-ci peuvent être erronées. Une déconnexion d'un processus mobile est considérée après une certaine durée comme une défaillance par les détecteurs de défaillances. Les processus corrects continuent donc à envoyer les messages aux hôtes déconnectés. Une information sur les déconnexions est souhaitable pour éviter des envois de messages inutiles, afin de préserver l'énergie et la bande passante.

L'algorithme du consensus qui s'appuie jusqu'ici sur les détecteurs de défaillances, peut aussi s'appuyer sur les détecteurs de connectivité pour restreindre l'ensemble de diffusion des messages dans le cas de déconnexions. Un détecteur de défaillances d'un processus p fournit une information binaire qui détermine si un autre processus q est correct ou défaillant. Un détecteur de connectivité, quand à lui, fournit à p une information ternaire qui détermine son propre mode (déconnecté, connecté et partiellement connecté). Les informations additionnelles fournies par le détecteur de connectivité impliquent des changements dans la sémantique du consensus, et dans l'algorithme du consensus.

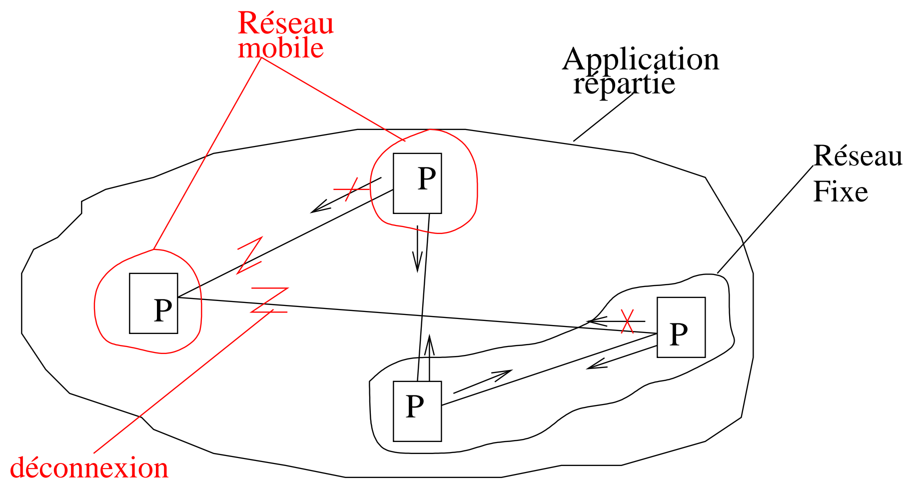

FIG. 3.3 – Arrêt d'envoi de message lors d'une déconnexion

Les architectures possibles rassemblant le consensus et les deux types de détecteurs sont représentées par les diagrammes UML de la figure 3.4. La figure 3.4-*a* présente une architecture où le consensus interroge un détecteur de défaillances qui s'appuie sur un détecteur de connectivité. La figure 3.4-*b* présente le consensus s'appuyant cette fois-ci sur un détecteur de connectivité interroge à son tour un détecteur de défaillances. La dernière figure 3.4-*c* montre une architecture dans laquelle le consensus interroge directement et séparément les deux détecteurs de connectivité et de défaillances.

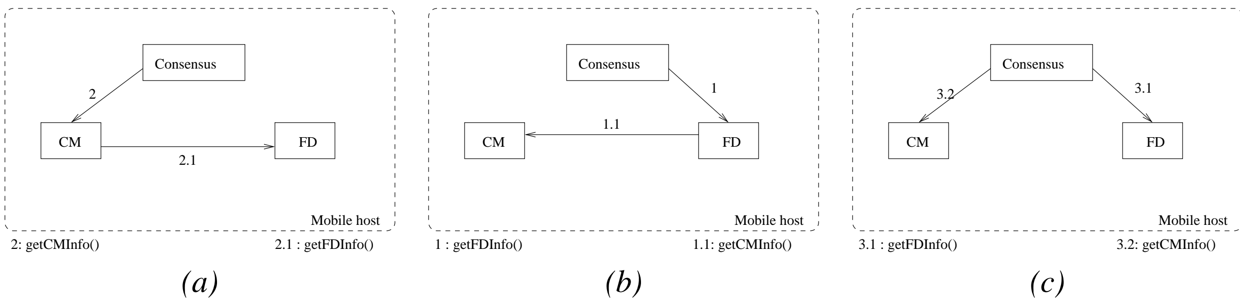

FIG. 3.4 – Utilisation des détecteurs de défaillances et de connectivité par le consensus

Parmi les architectures présentées dans la figure 3.4, la première ne semble pas être une solution qui améliore le problème du consensus. Le consensus a besoin de connaître les processus qui sont défaillants de ceux qui sont corrects. Dans le gestionnaire de connectivité de bout-en-bout présenté dans la section 3.2, une variation nulle de réception de battements de cœur peut être causée soit par une déconnexion, soit par une défaillance. Cette approche ne différencie pas les déconnexions des défaillances, elle n'apporte donc pas d'amélioration de l'algorithme du consensus.

Dans la deuxième approche, en plus de la liste des processus suspectés de défaillances, le détecteur de défaillances fournit aussi une liste de processus déconnectés. Le détecteur de défaillances utilise les informations de déconnexions pour séparer les traitements des hôtes défaillants de ceux qui sont déconnectés, optimisant ainsi les envois inutiles. Cette architecture est intéressante puisqu'elle sépare les traitements des processus déconnectés de ceux qui sont défaillants.

Dans la dernière approche, il est intéressant que le consensus ait des informations sur les déconnexions ajoutées à celles fournies par les détecteurs de défaillances. Le consensus peut alors enlever les hôtes déconnectés des listes de diffusion des messages. Cependant, pour disposer des informations de déconnexions des autres processus, nous devons créer un module de détection de déconnexions à partir des détecteurs de connectivité.

Dans la suite, nous nous intéressons en premier lieu à la conception de cette dernière architecture, ensuite nous revenons à la deuxième architecture, et à la fin nous présentons une architecture qui mixe ces deux dernières.

3.3.3 Du détecteur de connectivité au détecteur de déconnexions réparti

Bien que le détecteur de connectivité fournisse des informations de déconnexions et de reconnexions, celles-ci restent locales. Une information de déconnexion globale est souhaitable, pour cela un détecteur de déconnexions est créé afin de fournir une détection de déconnexion répartie. Un détecteur de déconnexionsest un détecteur de connectivité qui échange des informations de déconnexions avec les autres processus.

Avant de présenter l'algorithme du détecteur de déconnexions réparti, nous commençons par décrire le modèle du système dans la sous-sous-section 3.3.3.1. Ensuite, nous présentons le concept du détecteur de déconnexions dans la sous-sous-section 3.3.3.2, puis ses propriétés dans la sous-sous-section 3.3.3.3. Ensuite, nous complétons quelques propriétés de l'hystérésis dans la sous-sous-section 3.3.3.4. Nous terminons par présenter l'algorithme de détection de déconnexions et la preuve de correction respectivement dans les sous-sous-sections 3.3.3.5 et 3.3.3.6.

3.3.3.1 Modèle de déconnexions

Le modèle considéré se compose d'un système reparti *asynchrone*. Le système est composé de machines mobiles ou fixes possédant un module de détection de déconnexions répartie. Nous utilisons dans la suite le terme processus ou hôte d'une manière interchangeable pour désigner un hôte participant au consensus. Le système réparti consiste en un ensemble de n processus $\Pi = \{p_1, p_2 ..., p_n\}$ où chaque paire de processus est connectée par une interface de communication sans fil supposée fiable sans perte de messages. La communication se fait par envoi de messages via les primitives de diffusion FIFO. Chaque hôte dispose d'un mécanisme d'hystérésis pour la détection de connectivité.

Un processus peut se déconnecté, volontairement ou involontairement. Le modèle de déconnexions DC est une fonction de $\mathcal{T}$ vers 2^{Π}, où $DC(t)$ est l'ensemble des processus déconnectés à l'instant t. $disconnected(DC) = \bigcup_{t\in\mathcal{T}} DC(t)$ représente l'ensemble des processus déconnectés dans DC, et $connected(DC) = \Pi \setminus disconnected(DC)$ représente les processus connectés dans DC. Seul le cas où il existe au moins un processus connecté dans DC est considéré : c-à-d, $connected(DC) \neq \emptyset$. Si un processus se déconnecte alors il le reste pour l'instance du consensus courant : c-à-d, $\forall t\ :\ DC(t) \subseteq DC(t+1)$. Une fois le processus déconnecté, il peut se reconnecter mais il n'est pas enlevé de l'ensemble $DC(t)$ pendant l'exécution du consensus courant et il ne peut donc plus y participer. Il ne sera réintégré qu'au prochain consensus. Si $p \in disconnected(DC)$, on dit que p est vu déconnecté dans DC et si $p \in connected(DC)$, on dit que p est vu connecté dans DC.

3.3.3.2 Détecteur de déconnexions

Un détecteur de déconnexions réparti est un module qui fournit une information sur les hôtes déconnectés. Chaque hôte mobile h possède un module de détection de déconnexions. Le principe est que chaque hôte qui va se déconnecter avertit les autres hôtes connectés de sa déconnexion, et chaque hôte qui se reconnecte avertit aussi les autres hôtes connectés de sa reconnexion. La communication est fiable sauf dans le cas où un hôte ne réussit pas à envoyer

l'avertissement d'une déconnexion. Ce cas est géré par la génération d'une exception adressée à la plate-forme qui se charge d'envoyer le message de déconnexion dès que les ressources le permettent.

Chaque module de détection de déconnexions fournit l'ensemble des processus qu'il voit déconnectés. L'historique H_{DC} d'un détecteur de déconnexions est une fonction de l'ensemble $\Pi \times \mathcal{T}$ vers l'ensemble 2^{Π} où $H_{DC}(p,t)$ est la valeur du détecteur de déconnexions du processus p à l'instant t dans H_{DC} . Si $q \in H_{DC}(p,t)$, alors q *est vu déconnecté par* p à l'instant t dans H_{DC}. Notons que deux processus distincts peuvent avoir deux listes de processus vus déconnectés différentes à un instant donné, c-à-d, si $p \neq q$ alors $H_{DC}(p,t) \neq H_{DC}(q,t)$ est possible. Un détecteur de déconnexions $\mathcal{DD}$ fournit des informations sur le modèle de déconnexions DC pendant l'exécution du processus correspondant. Formellement, un détecteur de déconnexions $\mathcal{DD}$ est une fonction qui associe à chaque modèle de déconnexions DC un ensemble d'historiques de détecteurs de déconnexions $\mathcal{DD}(DC)$. $\mathcal{DD}(DC)$ est l'ensemble de tous les historiques pouvant exister durant des exécutions avec l'ensemble des déconnexions DC et le détecteur de déconnexions $\mathcal{DD}$.

3.3.3.3 Propriétés

Similairement aux détecteurs de défaillances, les détecteurs de déconnexions ne doivent pas être définis pour une implémentation spécifique. Pour cela, nous les définissons par les mêmes propriétés abstraites utilisées pour définir les détecteurs de défaillances. Un détecteur de déconnexions doit satisfaire une propriété de complétude et une propriété de précision. Pour les différencier des propriétés de complétude et de précision des détecteurs de défaillances, nous les référençons respectivement par complétude de déconnexion et précision de déconnexion.

- **Complétude de déconnexion**
  - *forte* : il existe un instant après lequel tout processus qui avertit de sa déconnexion est vu déconnecté par tous les processus connectés. D'une manière formelle, $\mathcal{DD}$ satisfait la complétude faible si :

$$\forall DC,\ \forall H_{DC} \in \mathcal{DD}(DC),\ \exists t \in \mathcal{T},$$
$$\forall p \in disconnected(DC),\ \forall q \in connected(DC),\ \forall t' \geq t : p \in H_{DC}(q,\ t')$$

  - *faible* : tout processus qui avertit de sa déconnexion est vu déconnecté par au moins un processus connecté. D'une manière formelle, $\mathcal{DD}$ satisfait la complétude faible si :

$$\forall DC,\ \forall H_{DC} \in \mathcal{DD}(DC),\ \exists t \in \mathcal{T},$$
$$\forall p \in disconnect(DC),\ \exists q \in connected(DC),\ \forall t' \geq t : p \in H_{DC}(q,\ t')$$

- **Précision de déconnexions**
  - *forte* : aucun processus n'est vu déconnecté avant qu'il ne soit vraiment déconnecté.

Formellement, $\mathcal{DD}$ satisfait la précision forte si :

$$\forall DC,\ \forall H_{DC} \in \mathcal{DD}(DC),\ \forall t \in \mathcal{T},\ \forall p,\ q \in \Pi - DC(t) : p \notin H_{DC}(q,\ t)$$

Contrairement aux propriétés du détecteur de défaillances, les propriétés de complétude et de précision du détecteur de déconnexions sont facilement satisfaisables. À l'inverse d'une défaillance qui est subite, une déconnexion volontaire ou involontaire est prévisible un laps de temps avant son arrivée. Un processus a le temps de prévenir les autres processus de sa déconnexion. Cette faculté de prévision est fournie grâce au mécanisme d'hystérésis encapsulé par le gestionnaire de connectivité. Un processus p est vu déconnecté par q uniquement si q reçoit un message de déconnexion de p. Un processus ne peut être vu déconnecté que s'il a envoyé un message de déconnexion et se déconnecte. Si un processus p émet un message de déconnexion, grâce aux communications fiables, après un certaine durée, les autres processus connectés le reçoivent. Ils mettent alors le processus émetteur dans l'ensemble des processus déconnectés. Lors de la reconnexion, le processus p envoie un message de reconnexion à tous les processus de Π. Après une certaine durée, tous les processus connectés q reçoivent le message de reconnexion de p, ils l'enlèvent alors de la liste des processus déconnectés.

La précision forte est donc facile à assurée. La complétude faible est assurée dès qu'un processus p reçoit un message de déconnexion d'un processus q et que $p \in H_{DC}(q,\ t)$. Le passage de la complétude faible à la complétude forte est faisable en ayant un algorithme de réduction $T_{\mathcal{DD} \longrightarrow \mathcal{DD}'}$ transformant un détecteur de déconnexions $\mathcal{DD}$ en un autre détecteur de déconnexions $\mathcal{DD}'$.

3.3.3.4 Complément sur les propriétés de l'hystérésis

Dans l'hystérésis, deux changements de modes sont importants. Le passage du mode partiellement connecté au mode déconnecté autour de la valeur *lowDown*, et le passage du mode partiellement connecté au mode connecté autour de la valeur *highUp*. Il est impératif que ces deux passages soit détectés et signalés par l'hystérésis. L'échantillonnage de l'hystérésis doit signaler tous les changements de modes, pour cela on admet que Δ_{val}, l'intervalle d'échantillonnage de l'hystérésis, doit être strictement inférieur au minimum du temps nécessaire pour un changement de mode.

Le gestionnaire de connectivité notifie le gestionnaire de déconnexions de chaque changement de mode correspondant aux deux changements de mode importants indiqués ci-dessus. Lors de chaque notification, le gestionnaire de déconnexions accède directement aux informations du gestionnaire de connectivité qui lui fournit une variable CONNECTED pour indiquer si le changement se fait pour une déconnexion ou une reconnexion volontaire ou involontaire.

3.3.3.5 Algorithme du détecteur de déconnexions

L'algorithme du détecteur de déconnexions présenté dans la figure 3.5 est composé de quatre tâches parallèles :

1. La première tâche s'exécute sur notification d'un changement de mode émise par l'hystérésis. Les paramètres de l'hystérésis sont accessibles à l'algorithme du *détecteur de déconnexions réparti*. La variable CONNECTED indique l'état de la connectivité du processus. Deux cas intéressants peuvent se présenter :
   (a) CONNECTED $= false$: indique le mode déconnecté. Dans ce cas un message de déconnexion est émis à tous les processus connectés. Si le message n'arrive pas à passer malgré les hypothèses sur les délais d'échantillonnage par l'hystérisis, à cause d'une déconnexion trop brusque, une exception est levée. La plate-forme se chargera d'émettre le message de déconnexion dès que les ressources le permettent.
   (b) CONNECTED $= true$: indique le mode connecté. Ce cas est considéré comme une reconnexion après une déconnexion. Le processus p envoie son message de reconnexion à tous les processus de Π.
2. La deuxième (*resp.* troisième) tâche est en écoute des messages de déconnexion (*resp.* reconnexion) envoyés par des processus qui transmettent leur avis de déconnexion (*resp.* reconnexion). Le processus p ajoute l'émetteur à (*resp.* enlève l'émetteur de) son ensemble des processus déconnectés.

Intuitivement, un processus n'est vu déconnecté que s'il a envoyé un message de déconnexion. Un processus déconnecté n'est vu reconnecté que s'il a envoyé un message de reconnexion. Un processus ne peut pas être vu déconnecté alors qu'il est connecté. Par contre, un

1. For every process p :
2. **Initialization :**
3. $disc(p) \leftarrow \emptyset$;
4. **cobegin**
5. $\|$*Task 1 :* **upon change mode notification**
6. **if** $CONNECTED = false$ **then**
7. **for all** $q \in \Pi \setminus disc(p)$ **F_send**(DISCONNECT) to q
8. **else**
9. **for all** $q \in \Pi$ **F_send**(RECCONNECT) to q
10. $\|$ *Task 2 :* **upon F_receive**(DISCONNECT) from q **do**
11. **if** $q \notin disc(p)$ **then** $disc(p) \leftarrow disc(p) \cup \{q\}$
12. $\|$ *Task 3 :* **upon F_receive**(RECONNECT) from q **do**
13. **if** $q \in disc(p)$ **then** $disc(p) \leftarrow disc(p) \setminus \{q\}$
14. **coend**

FIG. 3.5 – Algorithme du détecteur de déconnexions réparti

processus peut être vu connecté tandis qu'il est déconnecté. Ce dernier cas arrive lorsque le message de déconnexion d'un processus p n'est pas transmis en dépit des propriétés de l'hystérésis et les hypothèses sur les délais d'échantillonnage. Si un processus qui se déconnecte ne réussit pas à envoyer le message de déconnexion, une exception est levée. Ce message reste en attente jusqu'à ce qu'il soit envoyé. La propriété d'envoi FIFO assure que les messages de reconnexion et de déconnexion sont ordonnés. Ainsi, même si un processus p ne réussit pas à envoyer son message de déconnexion, il l'envoie dès qu'il lui est possible d'émettre, et toujours avant le message de reconnexion. que les messages sont reçus dans le même ordre que leur emissions.

Cet algorithme respecte la complétude faible et la précision forte. On peut cependant passé de la complétude faible à la complétude forte par l'algorithme de réduction présenté dans la figure 3.6. Ce dernier algorithme est similaire à l'algorithme de réduction d'un détecteur de défaillances de complétude faible à un détecteur de défaillances de complétude forte [CHT96].

---

1. Every process p executes the following :
2. **Initialization :**
3. $disc_p \leftarrow \emptyset$;
4. **cobegin**
5. $\|$*Task 1 :* **repeat forever**
6. $disc_p \leftarrow \mathcal{DD}_p$ $\{p$ interroge son module de détection de déconnexions $\mathcal{DD}_p\}$
7. **if** $forward = true$ **then F_send**$(p, disc_p)$ **to** $\Pi \setminus disc_p$
8. $\|$*Task 2 :* **upon F_receive**$(q, disc_q)$ from some q
9. $disc_p \leftarrow disc_p \cup disc_q - \{q\}$
10. **coend**

FIG. 3.6 – De la complétude de déconnexion faible à la complétude de déconnexion forte

---

Un algorithme de détection de déconnexions n'a de l'intérêt que s'il fournit une information exacte sur les déconnexions et les reconnexions des processus. Il est évident que si un processus p se reconnecte, il ne trouve pas la même configuration des processus ; d'autres processus peuvent se déconnecter ou se reconnecter pendant sa déconnexion. Son ensemble de processus vus déconnetés ne reflète plus les déconnexions à l'instant de la reconnexion. Pour cela, dans l'algorithme, l'ensemble des processus vus déconnectés par p est mis à zéro dès sa déconnexion. Au moment de la reconnexion, p envoie à tous les processus un message de reconnexion. p reconstruit son ensemble de déconnexions d'une façon générique grâce à l'algorithme de réduction de la figure 3.6 où chaque processus connecté envoie périodiquement son ensemble des processus déconnectés aux autres processus connectés. Il est donc nécessaire que les deux algorithmes de détection de déconnexions répartie et de réduction de la complétude de déconnexion faible à la complétude de déconnexion forte soit exécutés par chaque hôte du système.

3.3.3.6 Preuve de correction

Lemme 1 (Complétude de déconnexion forte :) *tout processus qui avertit de sa déconnexion est vu déconnecté par tous les processus connectés.*

Preuve : Supposons que p veut se déconnecter (*resp.* se reconnecter). Il envoie un message de type (DISCONNECT) à tous les processus q connectés à l'instant t (*resp.* (RECONNECT) à tous les processus de Π). Montrons que les propriétés de la diffusion fiable FIFO et celles de l'hystérésis garantissent que les messages arrivent aux destinataires. Pour la déconnexion, trois cas sont possibles :

- p réussit à envoyer son message de déconnexion à tous les processus connectés.
- p réussit à envoyer au moins un message de déconnexion à un processus connecté.
- p se déconnecte avant de pouvoir envoyer un message de déconnexion.

Dans le premier cas, tous les processus connectés reçoivent le message de déconnexion de p, ils l'insèrent alors dans leur ensemble de processus vus déconnectés. Dans le second cas, le processus q qui reçoit le message de déconnexion, p l'insert dans son ensemble des processus vus déconnectés. Grâce à l'algorithme de réduction qui transforme le détecteur de déconnexions de complétude faible en un détecteur de déconnexions de complétude forte, le processus q envoie périodiquement son ensemble des processus vus déconnectés à tous les autres processus connectés. Le dernier cas est le cas où p ne peut pas transmettre le message de déconnexions. Dans ce cas, p génère une exception gérée par la plate-forme. Celle-ci se charge de transmettre le message de déconnexion dès que les ressources le permettent et avant un message de reconnexion. Par ailleur, il n'existe pas de cas où un processus qui se déconnecte ne se reconnecte jamais, car, par hypothèse, l'hystérésis indique que l'application ne peut se terminer que dans l'état connecté. □

Lemme 2 (Précision de déconnexion forte :) *aucun processus n'est vu déconnecté avant qu'il ne soit vraiment déconnecté.*

Preuve : il est évident d'après l'algorithme qu'un processus q n'est vu déconnecté par p que lorsqu'il reçoit un message de déconnexion de q. Dans la figure 3.5, la ligne 11 est la seule instruction qui insère un processus q dans l'ensemble des processus vus déconnectés par p. Cette instruction n'est exécutée qu'à la réception d'un message de déconnexion de type (DISCONNECT) provenant du processus q (ligne 10). □

Theorem 1 *Les algorithmes de détection de déconnexions répartie de la figure 3.5 et de réduction de la figure 3.6 exécutés en parallèle satisfont la précision de déconnexion forte et la complétude de déconnexion forte.*

Preuve : Déduction des lemmes 1 et 2. □

3.3.4 Nouvel algorithme du consensus

Notre nouvel algorithme de consensus arrête de diffuser aux processus déconnectés car le gestionnaire de déconnexions lui fournit une liste des processus déconnectés. Nous commençons par donner les propriétés du nouveau consensus puis un algorithme et sa preuve de correction.

3.3.4.1 Propriétés du consensus

Les propriétés du consensus présentées dans la section 2.2 subissent une légère modification pour correspondre à un modèle de système réparti où les déconnexions et les reconnexions sont possibles en plus des défaillances. Avant de présenter les modifications et les nouvelles propriétés pour résoudre le consensus rappelons d'abord ses propriétés d'origine.

- *Terminaison* : tout processus correct décide éventuellement une valeur.
- *Validité uniforme* : si un processus décide une valeur v alors v a été proposée par un processus.
- *Intégrité uniforme* : chaque processus décide au plus une fois.
- *Accord* : deux processus corrects ne décident pas deux valeurs différentes.

La propriété de *Terminaison* telle qu'elle est énoncée ci-dessus n'est pas vérifiée par le nouvel algorithme du consensus. En effet, un processus qui se déconnecte avant la fin du consensus n'arrivera jamais à atteindre une décision. Il est tout le temps en attente de recevoir la décision. Afin de résoudre ce problème, une condition est ajoutée à l'attente de réception du message de décision. L'attente ne se fera que si le processus est connecté pendant toute la durée de ce consensus. Si un processus est déconnecté, il n'attend plus le message de décision et termine sans décider. La propriété de Terminaison modifiée ne concerne que les processus corrects connectés :

- *Terminaison connectée* : tout processus connecté (sur un hôte connecté pendant toute la duré du consensus) et correct décide éventuellement une valeur.

Puisqu'une décision ne concerne maintenant que les processus connectés, la propriété de l'accord est modifiée également :

- *Accord connecté* : deux processus *connectés* corrects ne décident pas deux valeurs différentes.

3.3.4.2 Résolution du consensus avec les détecteurs de défaillances et de déconnexions

Nous présentons dans cette section une solution au problème du consensus en utilisant un détecteur de défaillances de classe $\Diamond\mathcal{S}$ et un détecteur de déconnexions. Le consensus ne peut être résolu en utilisant un détecteur de défaillances qui appartient à la classe $\Diamond\mathcal{S}$ que s'il est supposé que le nombre maximum de processus défaillants déconnectés est strictement inférieur à la moitié du nombre de processus participants au consensus [CHT96]. Soit f le nombre maximum de

processus qui peuvent être défaillants ou déconnectés. Le système réparti asynchrone considéré satisfait la contrainte $f(F, DC) < \lceil \frac{n}{2} \rceil$, donc au moins $\lceil \frac{(n+1)}{2} \rceil$ processus sont connectés et corrects.

L'algorithme du consensus $\mathcal{CD}$ (« *Consensus with Disconnections* ») de la figure 3.7 se compose de quatre tâches parallèles. La première et la seconde tâches sont identiques aux tâches de l'algorithme d'origine du consensus, excepté les instructions de contrôles de l'état de connexion dans les lignes 18, 27, 34, 35 et 41 avant chaque envoi, ainsi que dans les lignes 22, 30 et 39 dans les « *wait* » de la tâche 1 et dans la ligne 43 du « *wait* » de la tâche 2. Les deux nouvelles tâches sont liées au détecteur de déconnexions, la tâche 3 est l'attente d'une notification de déconnexions. La tâche 4 est la notification d'un nouvel ensemble des processus vus déconnectés. Elle s'exécute tant que p est connecté. Dès que p devient déconnecté, la tâche 4 se termine.

3.3.4.3 Preuve de correction

La validité de l'algorithme du consensus de la figure 3.7 repose sur la vérification du maintien des propriétés de terminaison connectée, validité uniforme, accord connecté et d'intégrité. Dans le cas général, la preuve de l'algorithme est identique à celle présentée dans [CHT96]. Elle reste inchangée pour les processus qui sont connectés en permanence pendant toute la durée d'une instance d'un consensus. Cependant, la nouveauté dans cette configuration provient du fait qu'un processus a la possibilité de se déconnecter volontairement ou involontairement. Un processus qui est déconnecté ne peut plus émettre de message, il n'est donc plus en mesure de participer au consensus. Avant de prouver les propriétés du consensus, nous commençons par énoncer quelque propriétés concernant les processus déconnectés lors du consensus.

Lemme 3 *Un processus se considère déconnecté si $connect = false$.*

Preuve : Par les propriétés du détecteur de déconnexions, la détection d'une déconnexion est notifiée au consensus dans la tâche 3. La variable *connect* est aussitôt mise à $false$ dans la ligne 49, indiquant à p qu'il est désormais déconnecté. Une fois $connect = false$, elle le reste pendant tout le consensus. □

Lemme 4 *Un processus déconnecté ne participe plus au consensus.*

Preuve : Du Lemme 3, la variable *connect* de tout processus déconnecté est à $false$. De l'algorithme, tous les messages qu'un processus p envoie aux autres processus ne sont émis dans les lignes 18, 27, 34, 35 et 41 que si p est connecté ($connect = true$). Ainsi, lorsque $connect = false$, p n'envoie aucun message aux autres processus. Il arrête donc de participer au consensus. □

Lemme 5 *Un processus déconnecté termine sans décider.*

1. For every process p :
2.
3. To execute propose(v_p) :
4. $estimate_p \leftarrow v_p$ { $estimate_p$ est l'estimation de p de la valeur de décision}
5. $state_p \leftarrow undecided$
6. $r_p \leftarrow 0$ {r_p est le numéro du tour courant de p}
7. $ts_p \leftarrow 0$ {ts_p est le dernier tour dans lequel p modifie $estimate_p$}
8. $disc(p) \leftarrow \emptyset$ {$disc(p)$ ensemble des processus vu déconnecté par p }
9. $connect \leftarrow getConnectState()$ {$connect$ reflète l'état de la connexion}
10.
11. $||$*Task 1 :* **while** $connect = true$ **and** $state_p = undecided$
12. **repeat**
13. $r_p \leftarrow r_p + 1$
14. $c_p \leftarrow (r_p \ mod \ n) + 1$ {c_p est le coordinateur courant}
15. **until** $c_p \notin disc(p)$
16.
17. **Phase 1 :**
18. **if** $connect = true$ **then send**$(p,\ r_p,\ estimate_p,\ ts_p)$ to c_p
19.
20. **Phase 2 :**
21. **if** $p = c_p$ **then** $|disc(p)| < \lceil \frac{(n+1)}{2} \rceil$ **then**
22. **wait until** [**for** $\lceil \frac{(n+1)}{2} \rceil$ processes q : received$(q,\ r_p,\ estimate_q,\ ts_q)$ from q **or** $connect = false$]
23. **if** [for $\lceil \frac{(n+1)}{2} \rceil$ processes q : received$(q,\ r_p,\ estimate_q,\ ts_q)$] **then**
24. $msgs[r_p] \leftarrow \{(q,\ r_p,\ estimate_q,\ ts_q) \mid$ p received $(q,\ r_p,\ estimate_q,\ ts_q)$ from $q\}$
25. $t \leftarrow$ largest ts_q such that $(q,\ r_p,\ estimate_q,\ ts_q) \in msgs_p[r_p]$
26. $estimate_p \leftarrow$select one $estimate_q$ such that $(q,\ r_p,\ estimate_q,\ ts_q) \in msgs_p[r_p]$
27. **if** $connect = true$ **then send**$(p,\ r_p,\ estimate_p)$ to $\Pi \setminus disc(p)$
28.
29. **Phase 3 :**
30. **wait until** [received$(c_p,\ r_p,\ estimate_{c_p})$ from c_p **or** $c_p \in disc(p)$ **or** $D_p\ suspect\ c_p$ **or** $connect = false$]
31. **if** [received$(c_p,\ r_p,\ estimate_{c_p})$ from c_p] **then**
32. $estimate_p \leftarrow estimate_{c_p}$
33. $ts_p \leftarrow r_p$
34. **if** $connect = true$ **then send**$(p,\ r_p,\ ack)$
35. **else if** $c_p \notin disc(p)$ **and** $connect = true$ **then send** $(p,\ r_p,\ nack)$to c_p
36.
37. **Phase 4 :**
38. **if** $p = c_p$ **then**
39. **wait until** [for $\lceil \frac{(n+1)}{2} \rceil$ processes q : received$(q,\ r_p,\ ack)$ **or** $(q,\ r_p,\ nack)$ **or** $connect = false$]
40. **if** [for $\lceil \frac{(n+1)}{2} \rceil$ processes q : received $(q,\ r_p,\ ack)$] **then**
41. **if** $connect = true$ **then R-broadcast** $(q,\ r_p,\ estimate_p,\ decide)$
42.
43. $||$*Task 2 :* **wait until** [**upon** R-deliver $(q,\ r_p,\ estimate_q,\ decide)$ **or** $connect = false$]
44. **if** deliver $(q,\ r_p,\ estimate_q,\ decide)$ **and** $state_p = undecided$ **then**
45. decide($estimate_q$)
46. $state_p \leftarrow decided$
47.
48. $||$*Task 3 :* **wait until** [**upon** disconnect notification from LCM]
49. $connect = false$
50. $||$*Task 4 :* **while** $connect = true$ **and** $state_p = undecided$
51. **wait until** [**upon** received notification from LCM]
52. $disc(p) \leftarrow disc(p) \cup \mathcal{DD}_p$

FIG. 3.7 – Nouvel algorithme du consensus $\mathcal{CD}$ (« *Consensus with Disconnections* »)

Preuve : Du lemme 4, dès qu'un processus se voit déconnecté, il ne participe plus au consensus. Des lignes 22, 30 et 39 de l'algorithme, aucun processus déconnecté ne se bloque dans les « *wait* » des phases 2, 3 et 4. De la ligne 11, aucun processus déconnecté n'entame un nouveau tour, il termine donc l'exécution de la tâche 1. De la ligne 43, aucun processus déconnecté ne se bloque dans l'attente de la décision. Il termine la tâche 2 dès que $connect = false$, son état est toujours indécis. La tâche 4 s'exécute tant que le processus est connecté, dès qu'il devient déconnecté, il termine. □

Lemme 6 *Un processus qui est vu déconnecté demeure vu déconnecté par les autres processus pendant tout le consensus.*

Preuve : L'ensemble $disc(p)$ des processus vus déconnectés par p n'est modifié que dans la ligne 52. Un processus qui est inséré dans l'ensemble $disc(p)$ n'est pas enlevé pendant l'instance de ce consensus. Il est donc clair que si un processus q est vu déconnecté par p, il le reste pendant tous le consensus. □

Lemme 7 (Accord connecté :) *deux processus connectée ne décident pas deux valeurs différentes.*

Preuve : Les processus connectés se comportent exactement de la même manière que dans l'algorithme d'origine de la figure 2.3. Le consensus satisfait la propriété de l'accord prouvé dans [CHT96] pour les processus connectés. Des lemmes 4 et 5, puisqu'un processus déconnecté ne participe plus au consensus et qu'un processus déconnecté termine sans décider, deux décisions différentes ne peuvent exister. □

Lemme 8 (Terminaison connecté) *Tout processus correct connecté décide une valeur.*

Preuve : Deux cas sont possibles :

1. Un processus correct connecté décide. Il doit avoir livré d'une manière fiable un message de type $(-, -, -, decide)$. Par la propriété d'accord de l'émission fiable, tous les processus connectés corrects délivrent d'une manière fiable ce message et décident.
2. Aucun processus correct connecté ne décide. Nous affirmons qu'aucun processus correct connecté ne se bloque dans l'un des « *wait* ». La preuve est par contradiction. Soit r le plus petit numéro de tour où un processus connecté et correct se bloque indéfiniment dans l'un des « *wait* ». Ainsi, tous les processus connectés corrects terminent la phase 1 du premier tour r : ils envoient tous un message de type $(-, r, estimate, -)$ au coordinateur $c = (r \bmod n) + 1$. Puisque la majorité des processus sont corrects et connectés, il y a au moins $\lceil \frac{(n+1)}{2} \rceil$ de ces messages envoyés à c. Il existe trois cas à considérer :
   (a) c reçoit ces messages et répond en envoyant $(c, r, estimate_c)$. Ainsi, il ne se bloque pas indéfiniment dans les blocs « *wait* » de la phase 2.
   (b) c *crash*.

(c) c se déconnecte.

Dans le cas (a), tous les processus corrects connectés reçoivent un message de type $(c, r, estimate_c)$. Dans le cas (b), puisque $\mathcal{D}$ satisfait la complétude forte, pour tout processus correct p, il existe un instant après lequel c est suspecté d'une manière permanente par p, c.-à-d., $c \in \mathcal{D}_p$. Ainsi, il n'existe pas de processus correct connecté qui se bloque dans le second « *wait* » (phase 3). Dans le cas (c), puisque $\mathcal{DD}$ satisfait la complétude forte de déconnexion, pour tout processus correct p, il existe un instant après lequel c est vu déconnecté d'une manière permanente par p, c.-à-d., $c \in disc(p)$. De même, il n'existe pas de processus correct connecté qui se bloque dans le second « *wait* » (phase 3). De cette façon, chaque processus correct et connecté envoie à c, durant la troisième phase, un message de type$(-, r, ack)$ s'il est dans le cas (a), un message de type $(-, r, nack)$ s'il est dans le cas (b), ou enfin n'envoie aucun message s'il est dans le cas (c). Dans ce dernier cas où c est déconnecté, il n'est pas bloqué dans le «*wait*» de la phase 4 puisque $connect = false$ (ligne 39). Dans le cas où c est connecté, puisque il y a au moins $\lceil \frac{(n+1)}{2} \rceil$ processus corrects et connectés, alors c ne peut pas être bloqué dans le bloc « *wait* » de la quatrième phase. Ceci démontre que tous les processus connectés et corrects complètent le tour r — contradiction qui complète la preuve de l'affirmation.

Puisque $\mathcal{D}$ satisfait la précision faible inévitable, il existe un processus connecté correct q, et un instant t tel qu'aucun processus connecté correct ne suspecte q après l'instant t. Soit $t' \geq t$ un instant tel que tous les processus défaillants *crashent*. Notez qu'après l'instant t' aucun processus ne suspecte q. De là et de l'affirmation ci-dessus, il existe obligatoirement un tour r tel que :

(a) Tous les processus connectés corrects atteignent le tour r après l'instant t' (quand aucun processus ne suspecte q),

(b) q est le coordinateur du tour r (c.-à-d., $q = (r \bmod n) + 1$).

Durant la première phase du tour r, tous les processus connectés corrects envoient leur estimation à q. Durant la deuxième phase, q reçoit $\lceil \frac{(n+1)}{2} \rceil$ estimations et envoie $(q, r, estimate_q)$ à tous les processus connectés. Durant la troisième phase, puisque q n'est suspecté par aucun autre processus connecté correct après l'instant t, chaque processus connecté et correct attend l'estimation de q qu'il reçoit. Il répond alors par un message d'acquittement $(-, r, ack)$ à q. De plus aucun processus ne répond avec un message d'acquittement négatif $(-, r, nack)$ à q (ce qui n'arriverait que si un processus suspectait q). Par conséquent, durant la quatrième phase, q reçoit $\lceil \frac{(n+1)}{2} \rceil$ messages de type $(-, r, ack)$ et q diffuse d'une manière fiable $(q, r, estimate_q, decide)$. Par les propriétés de la validité et de l'accord de la diffusion fiable, tous les processus connectés corrects délivrent d'une manière fiable le message de q et décident — contradiction. Il en résulte que le cas 2 est impossible, ce qui conclue la preuve du lemme. □

Theorem 2 *L'algorithme $\mathcal{CD}$ résout le consensus utilisant $\diamond S$ et un détecteur de déconnexions réparti dans un système asynchrone avec $f(F, DC) < \lceil \frac{n}{2} \rceil$.*

Preuve : Les lemmes 7 et 8 montrent que $\mathcal{CD}$ satisfait respectivement les propriétés d'accord connecté et de terminaison connecté. De l'algorithme (ligne 46), il est clair qu'aucun processus ne décide plus d'une fois, par conséquent la propriété de l'intégrité uniforme est vérifiée. À partir de l'algorithme, il est clair aussi que toutes les estimations que le coordinateur reçoit durant la deuxième phase sont des valeurs proposées. Donc, la valeur de décision que le coordinateur choisit parmi ces estimations est une valeur proposée par au moins un processus. Ainsi, la validité uniforme du consensus est également satisfaite. □

3.4 Détection de déconnexions dans le détecteur de défaillances

Bien que l'algorithme du consensus présenté dans la figure 3.7 optimise les envois inutiles aux processus déconnectés, des envos inutiles existent encore dans le détecteur de défaillances puisque le détecteur de défaillances ne peut pas savoir si un processus est déconnecté. Un détecteur de défaillances utilisant un module de détection de connectivité optimise le nombre de messages envoyés en éliminant les processus déconnectés de la liste des diffusions.

3.4.1 Architectures

Pour incorporer la détection de déconnexions dans la détection de défaillances, nous proposons deux architectures. Dans la première architecture, figure 3.8-*a*, le détecteur de défaillances collabore directement avec différents détecteurs de connectivité, chacun attaché à une ressource locale. Dans la figure 3.8-*b*, le détecteur de défaillances collabore avec un seul détecteur de connectivité globale qui fournit une vue unique sur l'utilisabilité de différentes ressources.

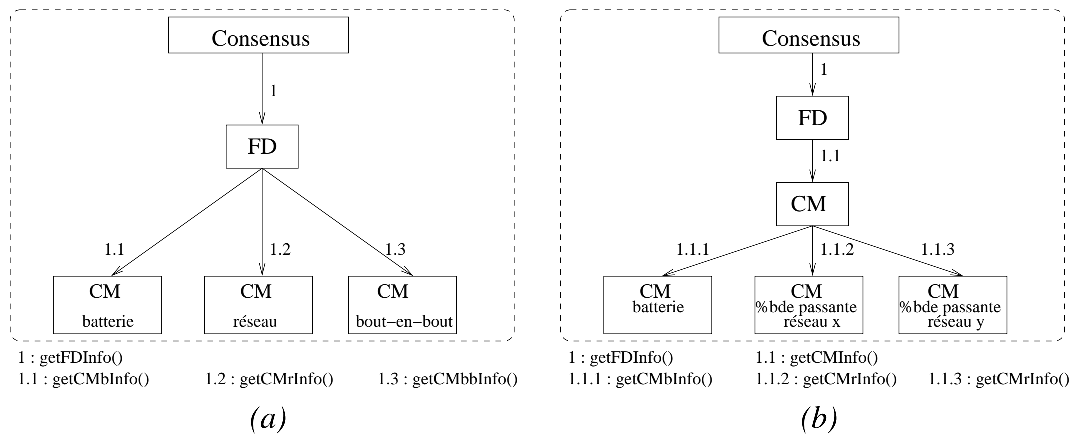

FIG. 3.8 – Collaboration des détecteurs de connectivité et de défaillances

Pour réaliser cette approche où un détecteur de défaillances utilise un détecteur de connectivité, nous avons choisi le détecteur de défaillances $\mathcal{HB}$ [ACT97] vu dans la section 2.4. Les

deux sous-sections suivantes présentent le modèle du système et les propriétés, puis la sous-section 3.4.3 présente l'algorithme de « *battements de cœur* » gérant en plus des défaillances, les déconnexions des processus. Nous nommons ce dernier $\mathcal{HBD}$ (« *HeartBeat with Disconnections* »).

3.4.2 Modèle et propriétés

Le modèle utilisé est le même que celui décrit dans le modèle du détecteur de déconnexions réparti.

L'ensemble $F = (F_P, F_{DC})$ combine l'ensemble des défaillances de processus F_P, où F_P à la même définition que F dans la section 2.1, et l'ensemble de déconnexion F_{DC}, où F_{DC} à la même définition que DC dans la sous-sous-section 3.3.3.1.

On suppose que le nouveau détecteur des défaillances nommé $\mathcal{HBD}$ a la même architecture que celle présentée dans la figure 3.8-*b*. $\mathcal{HBD}$ réunit à la fois les propriétés du détecteur de défaillances originel présenté dans la section 2.4 et les propriétés du détecteur de déconnexions réparti présenté dans la section 3.3.3. Comme dans le détecteur de déconnexions, le détecteur de connectivité fournit aussi une fonction qui détermine si le processus est connecté ou pas.

3.4.3 Algorithme de détection de défaillances et de déconnexions

L'algorithme est constitué de plusieurs tâches parallèles. Chaque processus p exécute cinq tâches pour gérer tous les types de messages : envois et réceptions de battements de cœur et messages de déconnexion et de reconnexion. Au démarrage de l'application, on suppose que tous les processus participants sont connectés. À l'initialisation, l'ensemble des processus déconnectés est vide. À la reconnexion d'un processus, l'ensemble des processus déconnectés est vide, il est reconstitué grâce à l'algorithme de réduction présenté dans la figure 3.6 qui permet de passer de la complétude de déconnexion faible à la complétude de déconnexion forte. Les traitements effectués par chaque tâche sont détaillés ci-dessous :

- La première tâche est déclenchée par une notification d'un changement de mode de connectivité. En plus des trois modes gérés par l'hystérésis et la variable $forward$ qui informe sur la possibilité d'envoi de messages, le gestionnaire de connectivité fournit la variable $\mathsf{CONNECTED}$ qui indique une notification de déconnexion $(false)$ ou de reconnexion $(true)$. Si $\mathsf{CONNECTED} = false$, le processus p commence par envoyer un message de déconnexion à tous les processus connectés puis vide l'ensemble $disc(p)$ des processus déconnectés. Si $\mathsf{CONNECTED} = true$, p envoie un message de reconnexion à tous les processus dans Π.

- Dans la deuxième tâche, périodiquement, p envoie un battement de cœur à tous les processus connectés. L'envoi ne se fait que si $forward = true$ indiquant que la connectivité permet de faire des envois.

- La troisième tâche gère la réception des messages de battements de cœur de q. p incrémente la valeur du compteur de q.

- La quatrième (*resp.* cinquième) tâche gère la réception des messages de la forme (DISCONNECT) (*resp.* (RECONNECT)) de q. À la réception d'un tel message, p ajoute q à (*resp.* enlève q de) l'ensemble des processus vus déconnectés.

```
1.  For every process p :
2.    Initialization :
3.      disc(p) ← ∅;
4.    cobegin
5.     ||Task 1 : upon change mode notification
6.                 if CONNECTED = false then
7.                   for all q ∈ Π \ disc(p) F_send(DISCCONNECT) to q
8.                   disc(p) ← ∅
9.                 else for all q ∈ Π F_send(RECONNECT) to q
10.    ||Task 2 : repeat periodically
11.                 if forward = true then
12.                   for all q ∈ Π \ disc(p) do send(HEARTBEAT) to q
13.    || Task 3 : upon receive(HEARTBEAT) from q do
14.                 D_p[q] ← D_p[q] + 1
15.    || Task 4 : upon F_receive(DISCONNECT) from q do
16.                 if q ∉ disc(p) then disc(p) ← disc(p) ∪ {q}
17.    || Task 5 : upon F_receive(RECONNECT) from q do
18.                 if q ∈ disc(p) then disc(p) ← disc(p) \ {q}
19.   coend
```

FIG. 3.9 – Algorithme $\mathcal{HBD}$

3.4.4 Preuve de correction

L'algorithme $\mathcal{HBD}$ présenté dans la figure 3.9 est un hybride de $\mathcal{HB}$ présenté dans la figure 2.4 et du détecteur de déconnexions réparti de la figure 3.5.

Nous allons prouver que $\mathcal{HBD}$ vérifie les propriétés de complétude et de précision de $\mathcal{HB}$ et les propriétés de complétude et de précision de déconnexion du détecteur de déconnexions réparti.

Puisqu'on utilise le modèle de Chandra et Toueg où chaque paire de processus est connecté (atteignable par une communication sans fil), les voisins de p sont tous les processus de Π.

Lemme 9 ($\mathcal{HBD}$-Complétude) *Dans chaque processus correct connecté, la séquence de battements de cœur de tout processus défaillant ou déconnecté est bornée.*

Preuve : Il est évident que si un processus est défaillant ou déconnecté, il n'envoie pas de battements de cœur. Par conséquent, la valeur de son compteur dans un processus connecté et correct p est bornée. □

Lemme 10 *Dans chaque processus p, la séquence de battements de cœur de tout processus ne diminue jamais.*

Preuve : Il est claire que $\mathcal{D}_p[q]$ ne diminue pas puisqu'il n'est changé que dans la ligne 14. □

Lemme 11 *Dans chaque processus p connecté correct, la séquence de battements de cœur de tout processus connecté correct q est non bornée.*

Preuve : Soit p et q une paire de processus. Si q est correct, sa tâche 1 est exécutée un nombre infini de fois. Donc, si q est connecté, il envoie un nombre infini de messages HEARTBEAT au processus connecté p. Par la propriété des communications fiables, si p est connecté correct, il reçoit un nombre infini de messages HEARTBEAT de q. Lorsqu'un processus p connecté correct reçoit un message HEARTBEAT de q, il incrémente $\mathcal{D}_p[q]$ dans la ligne 14. Ainsi, p incrémente $\mathcal{D}_p[q]$ un nombre infini de fois. De plus, par le lemme 10, $\mathcal{D}_p[q]$ n'est pas décrémenté. Donc, le nombre de battements de cœur de q dans p est non bornée. □

Corollaire 1 ($\mathcal{HBD}$-Précision) *Dans chaque processus, la séquence de battements de cœur de tous les processus de* Π *ne diminue pas. Et dans chaque processus correct connecté, la séquence de battements de cœur de tous les processus corrects connectés est non bornée.*

Preuve : Des lemmes 10 et 11. □

Lemme 12 *L'algorithme $\mathcal{HBD}$ vérifie les propriétés de $\mathcal{HB}$.*

Preuve : Du lemme 9 et du corollaire 1 □

Lemme 13 *Les propriétés de complétude et de précision de déconnexion que vérifie l'algorithme de détection de déconnexions de la figure 3.5, sont aussi garanties par l'algorithme $\mathcal{HBD}$ de la figure 3.9.*

Preuve : L'algorithme de détection de déconnexions de la figure 3.5 est constitué d'une première tâche de notification de changement de mode, d'une seconde tâche de réception des messages de déconnexion et d'une troisième tâche de réception des messages de reconnexion. Ces tâches correspondent respectivement aux tâches 1, 4 et 5 de l'algorithme $\mathcal{HBD}$ de la figure 3.9. Le modèle du système $\mathcal{HBD}$ est identique au système du détecteur de déconnexions réparti. Étant donné que les propriétés de complétude et de précision de déconnexion sont vérifiées par l'algorithme de déconnexions, d'une façon triviale, elles le sont aussi par l'algorithme $\mathcal{HBD}$. □

Theorem 3 *À partir des lemmes 12 et 13, la figure 3.9 implémente $\mathcal{HBD}$ le nouveau détecteur de défaillances et de déconnexions.*

3.5 Conclusion

Au cours de ce chapitre, nous avons montré que les détecteurs de défaillances et les détecteurs de connectivité sont complètement différents. Chacun d'eux possède ses propres propriétés. Néanmoins, leur association dans des applications est fort intéressante pour améliorer les traitements et compléter une vue globale sur le contexte des hôtes mobiles. Par ailleurs, nous avons donné, dans ce chapitre, quelques idées d'utilisation des deux détecteurs. Nous avons d'abord répondu à la question : un détecteur peut-il être utilisé au profit de l'autre détecteur ? En proposant deux approches, nous avons montré que, bien que les deux détecteurs soient complètement dissemblables, il est possible d'utiliser chacun des deux au profit de l'autre. Un détecteur de connectivité peut utiliser un détecteur de défaillances pour avoir une vue globale sur la connectivité bout-en-bout initialement restreinte à une vue de la connectivité des ressources locales. Un détecteur de défaillances peut aussi utiliser un détecteur de connectivité, d'abord pour connaître le niveau de disponibilité de ses ressources locales et avoir ainsi une vue sur le futur. Nous avons ensuite répondu à la question : existe-il des possibilités d'avoir une utilisation qui associe les détecteurs dans une seule architecture ou une seule application ? Plusieurs architectures associent les deux détecteurs. D'abord, un détecteur de défaillances est incorporé dans l'architecture CORBA de *Domint*. Ensuite, un détecteur de connectivité est utilisé par le consensus. Dans cette dernière approche, nous avons optimisé le nombre d'envois de messages en excluant les processus vus déconnectés des listes de diffusion.

Chapitre 4

Conclusion et perspectives

Depuis l'apparition des technologies sans fil dans les années 90, les systèmes mobiles n'ont pas cessé de prendre de l'ampleur, proposant des services de plus en plus nombreux. Ces services donnent naissance à de nouvelles applications réparties mobiles. Cependant, celles-ci sont sujettes aux déconnexions fréquentes. La déconnexion est soit décidée par l'utilisateur pour des raisons pratiques (coût des communications, énergie de la batterie), c'est la déconnexion *volontaire*, soit elle est due à une coupure physique de la communication sans fil, c'est la déconnexion *involontaire*. Un terminal mobile doit être capable de travailler en mode déconnecté. Un protocole de détection de connectivité est donc nécessaire pour prévoir une déconnexion afin de charger les données nécessaires aux traitements de l'application en mode déconnecté. *Domint* est une plate-forme fournissant au terminal mobile un mécanisme de détection de connectivité. Il offre une continuité de service en mode déconnecté. Toutefois, les applications réparties sont aussi sujettes aux défaillances. La résolution du consensus dans un système sujet aux *crashs* de processus fait appel aux modules de détection de défaillances non fiables.

L'objectif de ce stage est d'étudier les deux types de détecteurs en terme de propriétés et de fonctionnalités, et enfin, de proposer des applications qui gèrent à la fois les déconnexions et les défaillances.

Dans une première partie, nous avons présenté le contexte des applications dans un environnement mobile. La plate-forme CORBA de *Domint* utilise l'hystérésis pour la détection de connectivité afin d'adapter le traitement des requêtes aux objets distants en mode partiellement connecté et en mode déconnecté. Cette étude nous a permis de dégager certaines propriétés telles que la détection de connectivité ne concerne que les ressources locales de la machine mobile. Nous avons ensuite étudié la tolérance aux fautes dans les systèmes répartis asynchrones sujet aux *crashs* de processus. Le consensus est un paradigme de la tolérance aux fautes, il est généralement résolu grâce aux algorithmes de détection de défaillances non fiables.

Dans la deuxième partie, nous avons décrit trois approches principales proposées au cours de ce stage. Nous avons essayé pour chaque approche d'utiliser les fonctionnalités de l'un des détecteurs pour compléter l'autre. Dans la première approche, nous avons utilisé les fonctionnalités d'un détecteur de défaillances au service du détecteur de connectivité pour offrir à ce dernier une vue globale avec une connectivité bout-en-bout des ressources distantes. Cette proposition améliore les décisions prises dans l'architecture de *Domint* pour le chargement des données et

le travail en mode déconnecté. Ces décisions ne s'appuient plus uniquement sur le niveau de connectivité locale, mais aussi sur le niveau de connectivité de la machine distante.

Dans la deuxième approche, nous avons pensé que connaître le niveau de connectivité dans un terminal mobile participant au consensus ainsi que les déconnexions est important pour améliorer l'algorithme du consensus, premièrement en s'abstenant d'envoyer des messages dès que le terminal est en mode déconnecté, deuxièmement en supprimant tout envoi de messages aux processus déconnectés. Nous avons donc utilisé les fonctionnalités du détecteur de connectivité pour concevoir au début un détecteur de déconnexions réparti. Celui-ci retourne à l'application une liste des processus vus déconnectés et une notification sur les changements de modes d'opération. Son rôle consiste à prévenir les autres terminaux de sa déconnexion et de sa reconnexion. Il collecte aussi les messages de déconnexion et de reconnexion provenant des autres terminaux mobiles. Nous avons ensuite décrit un algorithme de consensus qui utilise à la fois un détecteur de défaillances et un détecteur de déconnexions. Cependant, pour une amélioration complète et beaucoup plus intéressante, il faut aussi supprimer les envois des messages de battements de cœur. Nous avons donc proposé une autre approche où le détecteur de connectivité est utilisé dans le détecteur de défaillances. Dans cette architecture, le détecteur de défaillances englobe les mêmes fonctionnalités que le détecteur de déconnexions réparti. Il fournit en plus d'une liste des suspects, une liste des processus vus déconnectés.

À la fin, nous avons pensé qu'il est d'autant plus intéressant d'utiliser la faculté de prévision des déconnexions non seulement pour prévenir les autres processus, mais aussi pour effectuer des traitements propres à l'application. Dans le cas du consensus résolu avec le détecteur de défaillances de la classe $\diamond\mathcal{S}$, avant de se déconnecter un terminal peut charger un mandataire sur le réseau fixe pour proposer son estimation. Ce dernier journalise les décisions prises par le consensus pendant la déconnexion du terminal mobile, préparant ainsi sa réconciliation. N'ayant pas le temps nécessaire pour formuler toute les propriétés, précisément sur le mécanisme de réconciliation, cette dernière suggestion est décrite en annexe. Elle est présentée comme une perspective d'une application complète qui englobe les mécanismes de détection de déconnexions, de détection de défaillances et de réconciliation.

Les approches décrites dans ce stage ne sont qu'un aperçu sur les possibilités d'associations des détecteurs de défaillances et des détecteurs de connectivité. Il est intéressant d'exploiter d'autres architectures qui peuvent se révéler très intéressantes et efficaces aussi bien dans la gestion de déconnexion que dans la tolérance aux fautes. Il est clair que nous n'avons pas pu exploiter toutes les possibilités d'utilisation, un travail que nous présentons comme perspective.

Enfin, dans ce rapport nous avons utilisé le détecteur de défaillances $\mathcal{HB}$ comme algorithme de base. En perspective, il est intéressant de voir l'intégration des fonctionnalités du détecteur de connectivité dans d'autres détecteurs de défaillances qui utilisent par exemple des paramètres de qualité de service. Il est aussi intéressant d'intégrer les fonctionnalités de détection de connectivité dans le détecteur de défaillances $\mathcal{HB}$ pour les réseaux non complètement connectés.

Annexe A

Déconnexion, reconnexion et réconciliation dans un consensus

Remarque : Cette approche n'est pas complète, particulièrement pour la réconciliation. La réintégration d'un processus nécessite une discussion, notamment sur la possibilité de commencer en concomitance d'autres consensus.

La section A.1 présente les motivations de cette approche. La section A.2 décrit l'architecture globale de collaboration du consensus, d'un détecteur de défaillances, d'un détecteur de connectivité et d'un mandataire, ainsi que leur mode d'interaction. La section A.4 détaille les implémentations des différents collaborateurs de cette architecture.

A.1 Motivations

Nous avons vu une optimisation des envois de messages dans le consensus et dans le détecteur de défaillances. Plusieurs utilisations peuvent être présentées dans lesquelles, en plus de l'optimisation des envois de messages, on exploite le travail en mode déconnecté et la réconciliation lors de la reconnexion. Le travail en mode déconnecté dépend des propriétés de l'application. Par exemple, dans un consensus qui utilise un détecteur de défaillances de la classe $\diamond\mathcal{S}$ il y a au moins la majorité des processus corrects. Cependant un processus qui veut se déconnecter peut charger un mandataire sur le réseau fixe pour effectuer des traitements pour participer à ce consensus et préparer sa réconciliation. Dans la suite, nous présentons un scénario dans lequel on utilise à la fois les propriétés du détecteur de connectivité dans le consensus et dans le détecteur de défaillances, plus les services d'un mandataire pour la réconciliation.

A.2 Architecture et scénario

Le scénario de collaboration représenté par la figure A.1 sans les déconnexions se déroule comme un consensus avec un détecteur de défaillances classique. Cependant, lorsqu'un processus p veut se déconnecter, il informe son mandataire mdt_p de sa déconnexion en lui adressant

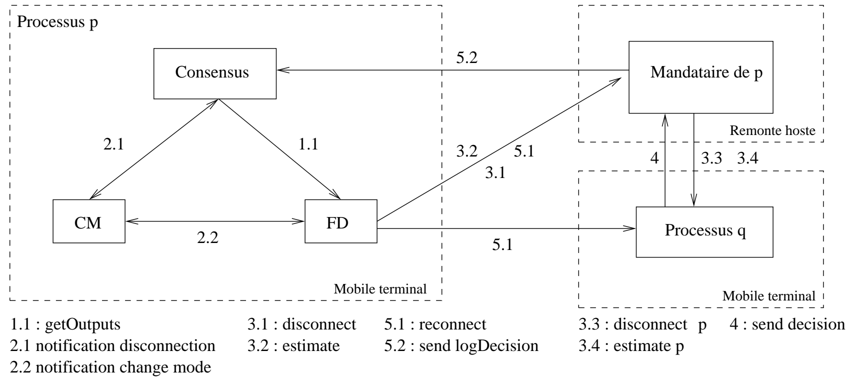

FIG. A.1 – Interaction du consensus, du détecteur de défaillances, du détecteur de connectivité et du mandataire

un message à partir du détecteur de défaillances l'avertissant de sa déconnexion. Si le processus p exécute la procédure $proposer(v)$, il envoie un message à partir du consensus en incluant sa proposition $estimate_p$ et l'ensemble des processus vus connectés ($\Pi \setminus disc(p)$) . Le mandataire informe l'ensemble des détecteurs de défaillances des processus connectés de la déconnexion du processus p, puis envoie la proposition de p au processus coordinateur. À la réception du message de déconnexion de p par le détecteur de défaillances FD, chaque processus connecté q ajoute p à son ensemble de processus déconnectés et ajoute mdt_p à son ensemble des mandataires représentant des processus déconnectés. À chaque décision prise par le consensus, le processus coordinateur diffuse à tous les processus connectés, y compris les mandataires qui journalisent les décisions dans un journal d'opérations. À la reconnexion de p, p envoie des messages de reconnexions à tous les processus dans Π, puis un message de reconnexion à mdt_p. À la réception du message de reconnexion de p, mdt_p transmet le journal des décisions.

A.3 Modèle et propriétés

On considère un système identique à celui présenté dans la sous-section 2.4.1, à l'exception que dans ce modèle, on suppose qu'il existe pour chaque hôte mobile un mandataire sur le réseau fixe qui le représente lors de ses déconnexions. Le détecteur de défaillances est identique à celui présenté dans la sous-section 3.4, sauf que celui-ci fournit en plus des défaillances et des déconnexions un ensemble de mandataires des processus déconnectés pour que le consensus puisse leur transmettre les décisions prises. Les propriétés de complétude et de précision du détecteur de défaillances et les quatre propriétés du consensus sont maintenues. Les deux algorithmes gardent la même structure de base que ceux présentés dans les sous-sections 3.3.4 et 3.4. Leur preuves de corrections sont identiques.

```
1.  For every mandataire mdt_p :
2.    cobegin
3.    ||Task 1 :  upon F_receive (p, r_p, estimate_p, ts_p, c_p) from p do        {À la réception de l'estimation de p}
4.                  F_send (mdt_p, r_p, estimate_p, ts_p, c_p) to c_p           { le mandataire l'envoie au coordinateur c_p }
5.                                                                              {À la réception du message de déconnexion de p, }
6.    ||Task 2 :  upon F_receive (DISCONNECT, connectSet, p) from p              {mdt_p informe les processus}
7.                ∀q ∈ connectSet F_send (DISCONNECT, p)                         {connectés connectSet de la déconnexion de p }
8.
9.    ||Task 3 :  upon F_receive (estimate_{c_p})
10.                 log(estimate_{c_p})                                          {Le mdt_p journalise les décisions prises par le consensus}
11.                                                                             {À la réception du message de reconnexion de p, }
12.   ||Task 4 :  upon F_receive (RECONNECT) from p                             { mdt_p lui transmet le journal des décisions}
13.                 F_send (log) to p                                            { pour la réconciliation}
14.   coend
```

FIG. A.2 – Algorithme du mandataire

A.4 Algorithmes

L'architecture présentée dans la figure A.1 fait appel à deux nouveaux éléments, l'algorithme du mandataire et celui de la réconciliation. Dans cette section, nous présentons en détail les tâches de chaque algorithme.

L'algorithme du mandataire présenté dans la figure A.2 a pour rôle essentiel d'informer les processus connectés des déconnexions du processus p et de garder les décisions prises par les consensus effectués pendant la déconnexion de p. Cet algorithme est constitué de quatre tâches. Dans la première tâche, si le processus p entame l'exécution de la procédure $proposer(v)$ juste avant sa déconnexion, alors il envoie à partir du consensus un message de type $(p,\, r_p,\, estimate_p,\, ts_p,\, c_p)$ au mandataire. À la réception d'un tel message, mdt_p envoie le même message de p au coordinateur c_p. Dans la deuxième tâche, mdt_p gère le message de déconnexion de p envoyés par le détecteur de défaillances. Il reçoit des messages de type $(DISCONNECT,\, connectSet,\, p)$ où $connectSet$ représente l'ensemble des processus vus connectés au moment de la déconnexion de p. mdt_p informe alors les processus connectés de la déconnexion de p. La troisième tâche s'occupe de la journalisation des décisions prises par le consensus pendant la durée de la déconnexion de p. La quatrième tâche transmet le journal des décisions au processus p dès la réception du message de reconnexion de p.

L'algorithme de détection de défaillances présenté dans la figure A.3 gère trois ensembles : l'ensemble des processus suspectés, l'ensemble des processus vus déconnectés et l'ensemble des mandataires. Il est composé de cinq tâches. La première tâche se déclenche par une notification

1. For every process p :
2. **Initialization :**
3. $disc(p) \leftarrow \emptyset$;
4. $mdtSet(p) \leftarrow \emptyset$;
5. $mdt_p \leftarrow MDT_Name$
6. **cobegin**
7. $||$*Task 1 :* **upon change mode notification**
8. **if** CONNECTED $= false$ **then**
9. **F_send** (DISCONNECT, $(\Pi - disc(p))\, p$) to mdt_p
10. $disc(p) \leftarrow \emptyset$
11. **else** **for all** $q \in \Pi \cup mdt_p$ **F_send** (RECONNECT) to q
12. **fork** reconcilisation()
13.
14. $||$*Task 2 :* **repeat periodically**
15. $D_p[p] \leftarrow D_p[p] + 1$
16. **if** $forward = true$ **then**
17. **for all** $q \in \Pi \setminus disc(p)$ **do send**(HEARTBEAT) to q
18.
19. $||$ *Task 3 :* **upon receive**(HEARTBEAT)from q **do**
20. $D_p[q] \leftarrow D_p[q] + 1$
21.
22. $||$ *Task 4 :* **upon F_receive** (DISCCONNECT, q) from mdt_q **do**
23. **if** $q \notin disc(p)$ **then** $disc(p) \leftarrow disc(p) \cup \{q\}$
24. **if** $mdt_q \notin mdtSet(p)$ **then** $mdtSet(p) \leftarrow mdtSet(p) \cup \{mdt_q\}$
25.
26. $||$ *Task 5 :* **upon F_receive** (RECONNECT) from q **do**
27. **if** $q \in disc(p)$ **then** $disc(p) \leftarrow disc(p) \setminus \{q\}$
28. **if** $MDT_q \in mdtSet(p)$ **then** $mdtSet(p) \leftarrow mdtSet(p) \setminus \{mdt_q\}$
29. **coend**

FIG. A.3 – Algorithme du détecteur de défaillances fournissant les ensembles des processus suspectés, vus déconnectés et des mandataires.

de changement de mode. Si le processus se déconnecte, il envoie un message de déconnexion à son mandataire avec la liste des processus connectés. Il vide ensuite son ensemble de processus déconnectés. À la reconnexion, p envoie un message de reconnexion à tous les processus et à son propre mandataire mdt_p. Il lance ensuite la fonction de la réconciliation. La deuxième et la troisième tâches sont les échanges classiques de battements de cœur. La quatrième tâche construit les ensembles des processus vus déconnectés à partir des messages de déconnexions

transmis par les mandataires et l'ensemble des mandataires à partir de la source des messages. La cinquième tâche gère les reconnexions des processus. Elle enlève les processus reconnectés et leur mandataire respectivement des ensembles $disc(p)$ et $mdtSet(p)$.

1. For every process p :
2.
3. To execute propose(v_p) :
4. $estimate_p \leftarrow v_p$ $\{$ $estimate_p$ est l'estimation de p de la valeur de décision$\}$
5. $state_p \leftarrow undecided$
6. $r_p \leftarrow 0$ $\{r_p$ est le numéro du tour courant de $p\}$
7. $ts_p \leftarrow 0$ $\{ts_p$ est le dernier tour dans lequel p modifie $estimate_p\}$
8. $disc(p) \leftarrow \emptyset$ $\{disc(p)$ ensemble des processus vu déconnecté par p $\}$
9. $connect \leftarrow getConnectState()$ $\{connect$ reflète l'état de la connexion$\}$
10.
11. ||*Task 1 :* **while** $connect = true$ **and** $state_p = undecided$
12. **repeat**
13. $r_p \leftarrow r_p + 1$
14. $c_p \leftarrow (r_p\ mod\ n) + 1$ $\{c_p$ est le coordinateur courant$\}$
15. **until** $c_p \notin disc(p)$
16.
17. **Phase 1 :**
18. **if** $connect = true$ **then send**$(p,\ r_p,\ estimate_p,\ ts_p)$ to c_p
19.
20. **Phase 2 :**
21. **if** $p = c_p$ **then**
22. **wait until [for** $\lceil \frac{(n+1)}{2} \rceil$ processes q : received$(q,\ r_p,\ estimate_q,\ ts_q)$ from q **or** $connect = false]$
23. **if** [for $\lceil \frac{(n+1)}{2} \rceil$ processes q : received$(q,\ r_p,\ estimate_q,\ ts_q)$] **then**
24. $msgs[r_p] \leftarrow \{(q,\ r_p,\ estimate_q,\ ts_q)$ | p received $(q,\ r_p,\ estimate_q,\ ts_q)$ from $q\}$
25. $t \leftarrow$ largest ts_q such that $(q,\ r_p,\ estimate_q,\ ts_q) \in msgs_p[r_p]$
26. $estimate_p \leftarrow$select one $estimate_q$ such that $(q,\ r_p,\ estimate_q,\ ts_q) \in msgs_p[r_p]$
27. **if** $connect = true$ **then send**$(p,\ r_p,\ estimate_p)$ to $\Pi \setminus disc(p)$
28.
29. **Phase 3 :**
30. **wait until** [received$(c_p,\ r_p,\ estimate_{c_p})$ from c_p **or** $c_p \in disc(p)$ **or** $D_p\ suspect\ c_p$ **or** $connect = false]$
31. **if** [received$(c_p,\ r_p,\ estimate_{c_p})$ from c_p] **then**
32. $estimate_p \leftarrow estimate_{c_p}$
33. $ts_p \leftarrow r_p$
34. **if** $connect = true$ **then send**$(p,\ r_p,\ ack)$
35. **else if** $c_p \notin disc(p)$ **and** $connect = true$ **then send** $(p,\ r_p,\ nack)$to c_p
36.
37. **Phase 4 :**
38. **if** $p = c_p$ **then**
39. **wait until** [for $\lceil \frac{(n+1)}{2} \rceil$ processes q : received$(q,\ r_p,\ ack)$ **or** $(q,\ r_p,\ nack)$ **or** $connect = false]$
40. **if** [for $\lceil \frac{(n+1)}{2} \rceil$ processes q : received $(q,\ r_p,\ ack)$] **then**
41. **if** $connect = true$ **then R-broadcast** $(q,\ r_p,\ estimate_p,\ decide)$ to $\Pi \setminus disc(p) \cup mdtSet$
42.
43. ||*Task 2 :* **wait until** [**upon** R-deliver $(q,\ r_p,\ estimate_q,\ decide)$ **or** $connect = false$]
44. **if** deliver $(q,\ r_p,\ estimate_q,\ decide)$ **and** $state_p = undecided$ **then**
45. decide($estimate_q$)
46. $state_p \leftarrow decided$
47.
48. ||*Task 3 :* **wait until** [**upon** disconnect notification from LCM]
49. **if** $state_p = undecided$ **then send**$(p,\ r_p,\ estimate_p,\ ts_p)$ to mdt_p
50. $connect = false$
51. ||*Task 4 :* **while** $connect = true$ **and**
52. **wait until** [**upon** received notification from LCM]
53. $disc(p) \leftarrow disc(p) \cup \mathcal{DD}_p$

FIG. A.4 – Algorithme du consensus

L'algorithme du consensus subit quelques changements minimes. Il est composé de quatre tâches. La première tâche est la procédure $proposer(v)$. Elle reste identique à celle présentée précédemment, mise à part la décision. Le processus p diffuse aussi les décisions aux mandataires qui remplacent les processus déconnectés. La deuxième tâche est la procédure $decide(v)$ qui reste inchangée. Dans la troisième tâche, p attend les notifications de déconnexions du gestionnaire de connectivité. À la réception d'une telle notification, p envoie son estimation à son mandataire, et met la variable d'état de déconnexion $connect$ à faux pour arrêter de participer au consensus et terminer sans décider. La quatrième tâche attend les notifications du détecteur de défaillances pour avoir la mise à jour des processus déconnectés.

---

1. for every processus p :
2. **cobegin**
3. to execute reconciliation ()
4. **wait until** [**upon** received (log) from mdt_p]
5. **execute** log
6. **coend**

FIG. A.5 – Réconciliation du processus p

L'algorithme de réconciliation de la figure A.5 est lancé par le détecteur de défaillances lors de la reconnexion du processus p. Il est composé d'une seule tâche qui se bloque sur l'attente de la réception du journal de décisions envoyé par mdt_p de p. À la réception du journal, p exécute les décisions prises pendant sa déconnexion. Ainsi, il est prêt à réintégrer la prochaine instance du consensus.

Bibliographie

[ACT97] M.K. Aguilera, W. Chen, and S. Toueg. HeartBeat : A Timeout-Free Failure Detector for Quiescent Reliable Communication. *Proceeding of the 11th Internationnal Workshop on Distributed Algotithms, Lecture Notes on Computer Science. Springer-Verlag.*, May 1997.

[ACT99] M.K. Aguilera, W. Chen, and S. Toueg. Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks. *Theoretical Computer Science, special issue on distributed algorithms*, July 1999.

[Bal01] R. Baldoni. Designing a Service of Failure Detection in Asynchronous Distributed Systems. *IEEE Computer*, 2001.

[BMS02] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. *IEEE Computer*, June 2002.

[CCVB03] D. Conan, S. Chabridon, O. Villin, and G. Bernard. Domint : A Platform for Weak Connectivity and Disconnected CORBA Objects on Hand-Held Device. *Rapport technique, INT, mai 2003.*, March 2003.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(2), February 1996.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2), March 1996.

[CTA01] W. Chen, S. Toueg, and M.K. Aguilera. On the Quality of Service of Failure Detectors. *IEEE Computer*, June 2001.

[DLP+86] D. Dolev, N.A. Lynch, S.S. Pinter, E.W. Stark, and W.E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3), July 1986.

[DLS88] C. Dwork, N.A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 1988.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2) :374–382, April 1985.

[Gär99] F.C. Gärtner. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM Computing Surveys*, 31(1) :1–26, March 1999.

[LFA99] M. Larrea, A. Fernández, and S. Arévalo. Efficient algorithms to implement unreliable failure detectors in partially synchrounous systems. *IEEE Computer*, September 1999.

[LFA02] M. Larrea, A. Fernández, and S. Arévalo. On the Imossibility of Implementing Perpetual Failure Detectors in Partially Synchronous Systems. *IEEE Computer*, March 2002.