Tolérance aux Défaillances dans les Systèmes Répartis en Environnement Mobile

Haroun Benkaouha

Université des Sciences et Technologies
Houari Boumediène

Faculté d'Electronique et Informatique
Département d'Informatique

Thèse de Doctorat

Tolérance aux Défaillances dans les Systèmes Répartis en Environnement Mobile

Haroun BENKAOUHA

Soumis en vue de l'obtention du Diplôme de Docteur en Informatique

07 / 04 / 2016

Soutenue devant le Jury :

Prof. Abdelkader BELKHIR	Président,
Prof. Makhlouf ALIOUAT	Examinateur,
Prof. Wakid-Khaled HIDOUCI	Examinateur,
Prof. Abdellah BOUKERRAM	Examinateur,
Dr. Djamel TANDJAOUI	Examinateur.

Directeur de Thèse: Prof. Nadjib BADACHE.

Résumé

Les avancées des nouvelles technologiques dans le domaine des systèmes et communications sans fils ont donné naissance aux systèmes distribués modernes. Ceci a engendré une augmentation dans le développement d'applications réparties mobiles. En outre, les systèmes distribués modernes sont sujets à de nouvelles contraintes qu'il faut prendre en considération et parmi ces contraintes la panne qui est devenue plus fréquente avec la vulnérabilité des unités de calcul. De ce fait, la tolérance aux pannes devient une propriété cruciale dans ce contexte et particulièrement pour les réseaux mobiles ad hoc (MANETs).

Notre travail consiste à développer des protocoles et mécanismes de tolérance aux pannes dans cet environnement très contraignant.

Nous nous intéressons particulièrement au contexte du bon déroulement d'une application distribuée. En effet, en cas de panne, il faut arrêter le calcul et un protocole de recouvrement doit être lancé. Il s'agit de trouver un moyen pour remplacer le nœud défaillant et reprendre par la suite le calcul à partir d'un point de reprise. Le point de reprise représente un état global cohérent par lequel est passé ou aurait pu passer le calcul. Pour définir de tels points, il faut enregistrer l'état global du système (pendant le déroulement du calcul distribué calcul) qu'on appelle point de contrôle. Cet enregistrement nécessite la disponibilité d'une mémoire stable pour chaque nœud. La tâche de calcul des points de contrôle (checkpointing) doit se dérouler en arrière plan.

Pour réaliser un tel mécanisme, nous proposons dans ce document un ensemble de protocoles : détection de pannes, checkpointing, mémoires stables virtuelles et un protocole de recouvrement arrière. Une analyse des performances de ces protocoles est donnée.

Mots clés : *Tolérance aux pannes, Système réparti, Application distribuée, Calcul distribué, MANETs, mobilité, ad hoc, défaillance, panne franche, détection de pannes, battement de cœur, point de contrôle, point de reprise, recouvrement arrière, protocole.*

Remerciements

Je tiens à remercier vivement Monsieur ***Nadjib BADACHE****, Professeur à l'U.S.T.H.B., et lui exprimer toute ma reconnaissance pour avoir dirigé les travaux de ma thèse.*

Je remercie particulièrement Monsieur ***Abdelkader BELKHIR****, Professeur à l'U.S.T.H.B., pour m'avoir fait l'honneur de présider le jury de ma soutenance.*

Je remercie également les Professeurs ***Abdellah BOUKERRAM*** *de l'Université de Béjaïa et* ***Walid-Khaled HIDOUCI*** *de L'E.S.I. pour l'intérêt qu'ils portent à mes travaux, en acceptant d'expertiser mon dossier de soutenance puis d'être examinateurs dans le jury de ma soutenance.*

Je remercie aussi Professeur ***Makhlouf ALIOUAT*** *de l'Université de Sétif et Docteur* ***Djamel TANDJAOUI*** *du C.E.R.I.S.T. qui ont accepté d'évaluer ce travail comme étant membres examinateurs de mon jury de soutenance.*

J'exprime toute ma gratitude à Monsieur ***Abdelkrim ABDELLI****, Professeur à l'U.S.T.H.B. pour sa précieuse aide et ses encouragements qui ont beaucoup contribué dans la finalisation de ma thèse.*

Ce travail n'aurait pas pu se faire sans les moyens matériels qui m'ont été offerts par le ***Département Informatique*** *et le laboratoire* ***L.S.I.*** *et à travers l'équipe de recherche* ***M.O.V.E.S.*** *au sein de laquelle j'ai pu participer et contribuer dans différents projets (CNEPRU, PNR et CMEP) qui m'ont beaucoup aidé dans mes travaux de recherche et dans ma thèse. Je tiens à remercier tous les membres de l'équipe, avec qui, il y a eu beaucoup de discussions et d'échanges, particulièrement avec Docteur* ***Youcef HAMMAL****.*

Je n'oublierai pas de remercier nos partenaires dans le cadre du projet de coopération CMEP, les Professeurs ***Lynda MOKDAD*** *et* ***Jalel BEN-OTHMAN****.*

Je remercie également tous mes collègues enseignants du Département Informatique avec qui j'ai le plaisir de travailler ainsi que les étudiants que j'ai encadrés.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Contexte et Problématique

Les systèmes distribués (ou répartis) sont un ensemble fini de sites (nœuds) inter-connectés par un réseau de communication, où chaque composant du réseau communique et coordonne son action uniquement par échange de messages [1]. Les systèmes distribués qui sont apparus dans les années 1960 n'ont pas arrêté de se développer. Avec le développement des réseaux et par la suite de l'Internet, les applications distribuées ont pris de plus en plus de place dans la vie quotidienne.

Les systèmes répartis sont caractérisés par une classe très large de problèmes et ses composants sont sujets à plusieurs types de fautes (pannes). Les pannes sont classifiées dans la littérature comme *transitoire, intermittente ou permanente*. Dans notre étude, nous nous intéressons aux pannes permanentes. Ce type de fautes continue à persister jusqu'à ce que le nœud défaillant soit remplacé. Nous traitons plus particulièrement les *pannes franches* appelées aussi *crash*. Cette dernière survient quand le processus d'un nœud donné s'arrête de façon définitive.

Le système comme entité unique doit continuer à fonctionner du mieux possible pour permettre le déroulement continu du calcul sous-jacent. Pour ces raisons, le système doit être tolérant aux pannes. De plus, les pannes ont un caractère imprévisible ainsi que la possibilité de la survenue de fautes multiples font de la conception de systèmes répartis tolérants aux pannes un problème difficile et souvent sans solution optimale.

Par ailleurs, un système est dit *tolérant aux pannes*, s'il peut continuer à fonctionner, même en *mode dégradé,* malgré la présence des pannes [2]. En d'autres

termes, les fonctionnalités du système doivent être implémentées de telle manière qu'il puisse automatiquement recouvrir à partir de pannes partielles sans affecter de façon sérieuse ses performances globales [2]. Cette propriété est un aspect important voire primordial et qui doit être pris en considération lors de la conception de protocoles distribués, plus particulièrement dans le contexte de l'environnement mobile.

Dans les réseaux mobiles et particulièrement ad hoc qu'on note MANETs (Mobile Ad hoc NETworks), les utilisateurs peuvent accéder aux données et aux applications n'importe où et à n'importe quel moment [3]. En effet, les MANETs apportent quelques avantages : mobilité, déploiement rapide, flexibilité... Par conséquent, on peut les utiliser dans divers domaines, dans des situations extrêmes et dans des conditions hostiles comme par exemple les désastres naturels, les guerres, ...

Toutefois, avec les nouvelles contraintes introduites par la mobilité aux systèmes distribués conventionnels (filaires), les protocoles conçus pour les réseaux filaires ou même pour les réseaux mobiles cellulaires ne sont plus appropriés aux MANETs. En effet, la mobilité des nœuds, l'absence d'infrastructures (comme par exemple les stations de base), la vulnérabilité des nœuds, les déconnexions volontaires et non-volontaires des nœuds, le problème d'autonomie en énergie et la limitation des ressources (mémoire, vitesse du processeur, bande passante) sont des contraintes additionnelles qui doivent être prises en considération lors de la conception et le développement de protocoles dans le contexte ad hoc mobile.

Nous situons nos travaux plus particulièrement dans le contexte de l'exécution d'une application distribuée sur un réseau mobile ad hoc. Cette dernière doit fonctionner normalement et surtout atteindre ses objectifs malgré la présence éventuelle des pannes. Ceci a orienté les recherches dans les systèmes et applications distribuées vers ces environnements de calcul. Les nouvelles contraintes introduites par ses environnements qui sont sujets à différents types de pannes, a mené à revoir la conception des systèmes distribués afin qu ils intègrent la tolérance aux pannes.

Le *calcul de points de contrôle basé recouvrement* (en anglais *checkpointing based recovery*) est l'une des approches de tolérance aux pannes qui a été définie dans la perspective de garantir le bon déroulement d'une application distribuée en présence de fautes. Cette technique est utilisée dans divers situations comme par exemple les bases de données distribuées et les applications critiques. Elle

est aussi utilisée dans le but de déboguer les programme distribués, dans le suivi et le contrôle et dans la détection de certaines propriétés du système tel que la terminaison ou le blocage [2]. Une telle approche nécessite l'implémentation de trois protocoles complémentaires :

Un protocole de *détection de pannes* : Chaque nœud doit exécuter le protocole, qui a pour but de détecter les nœuds défaillants. Pour cet effet, chaque nœud maintient et gère une liste contenant les identités des nœuds suspectés d'être défaillants. En effet, lors de l'occurrence d'une panne dans le système, le calcul distribué doit s'arrêter jusqu'à ce que le nœud défaillant soit réparé ou remplacé. Le processus de recouvrement est par la suite déclenché. La difficulté dans la conception de protocoles de détection de pannes consiste à garantir au mieux l'asynchronisme du système. Un système distribué *asynchrone* est décrit comme étant une collection de processus qui communiquent uniquement par échange de messages à travers le réseau de communication. Ce genre de systèmes est aussi appelé *systèmes par transmission de messages* (en anglais *message-passing systems*), où aucune hypothèse n'est émise sur le temps et le moyen de synchronisation. En d'autres termes, il n'y a ni horloge globale partagée, ni mémoire partagée. Ce genre de système rend la détection des nœuds défaillants une tâche complexe à atteindre. En effet, il devient difficile de distinguer entre un nœud lent et un nœud défaillant ; d'où le concept de détecteurs de pannes non fiables [4]. En plus des contraintes de mobilité sus-cités, un protocole de détection de panne dans les MANETs a tendance à suspecter les nœuds déconnectés volontairement ou involontairement comme étant défaillants, ce qui augmente les *fausses suspicions*.

Un protocole de *calcul de points de contrôle* : (appelé en anglais *checkpointing* qui capture et enregistre régulièrement l'état du système (un par nœud) et le sauvegarde dans une *mémoire stable* (en anglais *Stable storage*). Ces états sont appelés *points de contrôle* (en anglais *checkpoints*). L'ensemble de tous les états locaux doit former un état global cohérent. Il s'agit d'éviter d'enregistrer dans cet état des messages orphelins. Un message est dit orphelin, si l'événement de réception du message a été enregistré mais l'événement d'émission ne l'a pas été[5][6][2]. Dans ce qui suit, nous utilisons les termes point de contrôle ou checkpoint et les termes calcul de points de contrôle ou checkpointing pour indiquer le même sens.

Un protocole de *recouvrement arrière* : (appelé en anglais *rollback recovery*) est lancé après la détection de la panne. Il a comme fonction de permettre à l'application distribuée de récupérer après l'occurrence d'une panne en effectuant un retour arrière vers un état correct et surtout cohérent qui doit être le plus récent possible. Il consiste à faire revenir chaque nœud vers un état antérieur dans son calcul. Cet état a été enregistré par le protocole de calcul des points de contrôle.

1.2 Objectif et méthodologie

L'objectif de cette thèse est de concevoir puis analyser les performances d'un outils de tolérance aux pannes destiné aux applications distribuées s'exécutant sur un environnement mobile ad hoc. Cet outil consiste en un ensemble de protocoles : Des protocoles de détection de pannes, un protocole de calcul de points de contrôle, un protocole de reprise après défaillance. Notre conception offre la synchronisation entre ces différents protocoles. L'étude des performances de ses protocoles permet de valider nos propositions. Cette dernière étape est réalisée à travers des simulations.

Nos contributions peuvent être résumées à travers les points suivants :

La détection de panne : Nous proposons le protocole FDAN [7] qui a été amélioré en deux temps pour obtenir AFDAN [8] puis FDRAM [9]. Par ailleurs, les résultats d'un protocole de détection de pannes sont utilisés par un protocole complémentaire. Le plus souvent, il s'agit des protocoles de consensus. En effet, plusieurs protocoles ont été définis dans la littérature. La majorité de ces protocoles visent à résoudre les problèmes du consensus et/ou l'accord. Par contre, nos protocoles sont conçus afin qu'ils soient un complément aux protocoles de checkpointing et de recouvrement. Ainsi, nous tentons de réduire au maximum le nombre de fausses suspicions pour éviter de stopper le calcul distribué et de déclencher des opérations de retour arrière inutiles.

Le calcul de points de contrôle et le recouvrement : Nous proposons le protocole 2PACA (*Two Phases Algorithm of checkpointing for Adhoc mobile networks*) [10][11]. A notre connaissance, peu de travaux ont abordé le checkpointing dans le contexte des MANETs. La majorité des protocoles ont été définis pour les réseaux mobiles ad hoc hiérarchisés (clusterisés).

Notre protocole 2PACA est un protocole hybride qui s'exécute en deux (2) phases et en combinant deux techniques de checkpointing. Nous l'avons conçu pour les MANETs plats, en prenant en considération les contraintes de mobilité et de telle façon qu'il puisse fonctionner indépendamment de toute structure ou topologie du réseau et de tout protocole de routage. Ainsi, il peut être utilisé pour une large classe d'applications distribuées dans les MANETs.

1.3 Organisation du document

Le présent document est structuré en deux parties et est organisé en 8 chapitres, comme suit :

La première partie du document consistant en les quatre premiers chapitres, présente l'état de l'art :

Dans le chapitre 2, nous présentons un bref aperçu sur le domaine de la tolérance aux pannes. Dans le chapitre 3, des notions de base sur l'environnement mobile sont reprises ; nous abordons en particulier les réseaux mobiles ad hoc. Dans les chapitres 4 et 5, nous abordons respectivement les domaines de détection de pannes et le calcul de points de contrôle et recouvrement.

La seconde partie consistant en trois chapitres présente et discute nos différentes contributions :

Dans le chapitre 6, nous présentons un bref aperçu sur la conception de nos différents protocoles qui constituent notre outil de tolérance aux pannes. Les chapitre 7 et 8, présentent en détail les différents protocoles que nous avons proposés et établissent une étude comparative et une analyse de performances de nos protocoles afin de les valider.

Nous terminons cette thèse par une conclusion générale résumant le bilan des travaux réalisés et émettant des perspectives à ceux là.

Chapitre 2

La Tolérance aux Pannes

2.1 Introduction

Un système distribué (réparti) est une collection de machines indépendantes inter-connectées via un réseau de communication apparaissant à ses utilisateurs comme un seul et unique système cohérent [1][2]. Ce qui distingue ce genre de systèmes des machines mono-processeurs est la notion de la panne (défaillance) partielle [2]. La panne partielle intervient quand un composant du système réparti échoue. Cette panne peut avoir un impact sur les opérations de quelques composants mais en même temps, elle n'aura aucune incidence sur d'autres composants.[12]

L'un des principaux défis à relever lors de la conception des systèmes distribués est de construire un système capable de récupérer (se rétablir automatiquement afin de reprendre son fonctionnement normal) quand la panne se produit, sans incidence sérieuse sur le système global. Cette discipline est appelée la tolérance aux défaillances. Ce domaine est très large à tous points de vue, que ce soit sur le plan de la terminologie ou de la méthodologie [13].

Dans ce chapitre, nous présentons une synthèse de quelques techniques de tolérance aux pannes. Après un bref aperçu sur les systèmes répartis, nous introduisons quelques notions sur les pannes, et abordons quelques techniques utilisées dans les protocoles de tolérance aux défaillances.

2.2 Les systèmes répartis

Dans cette section nous donnons un aperçu sur les systèmes distribués asynchrones, le calcul distribué et l'état du système. Nous abordons aussi, la sûreté de fonctionnement des systèmes répartis.

2.2.1 Généralités sur les Systèmes Répartis Asynchrones

Un système distribué est un ensemble de sites inter-connectés par un réseau de communication (les différents canaux de communication). Il est modélisé par un ensemble fini de processus communiquants par envoi de messages. Le système est dit asynchrone, s'il ne comporte aucune hypothèse particulière sur le temps physique. Toute idée d'un processus qui détient une horloge locale synchronisée ou un raisonnement basé sur le temps réel global est à écarter [1][3]. L'absence d'hypothèses permet une représentation plus fidèle de la réalité [14] [5]. Toute synchronisation se fait uniquement à travers l'échange de messages. D'où la notion de système à passage (ou par transmission) de message [15].

A un système réparti, on associe un calcul distribué qui est réalisé à travers la collaboration de tous les processus du système. De façon non formelle, nous pouvons définir un calcul distribué comme une description de l'exécution d'un programme réparti sur un ensemble de processus [6][12].

Le programme distribué est vu comme étant l'exécution par chaque processus d'un algorithme local. L'activité des différents processus (exécution du programme) définit l'état global du système. On appelle état global du système l'ensemble des état locaux des processus (un par processus) en plus de l'état du sous système de communication (état des canaux). L'état local est l'ensemble des variables d'un processus. Le déroulement de l'algorithme donne une séquence de transitions atomiques entre les états locaux. Chaque transition correspond à un événement qui peut être interne ou externe (envoi ou réception d'un message).[1]

2.2.2 Sûreté de fonctionnement

Les utilisateurs ne peuvent placer leur confiance dans un service que s'il émane d'un système informatique assurant la sûreté de fonctionnement (dependability). Cette dernière est définie dans les systèmes répartis à travers quatre attributs : la disponibilité (availability), la fiabilité (reliability), la sécurité (sa-

TABLE 2.1 – La sureté de fonctionnement.

Sureté de Fonctionnement		
Attributs	**Entraves**	**Moyens**
Disponibilité Fiabilité Maintenabilité Sécurité, Confidentialité, Integritée	Fautes Erreurs Défaillance (panne)	Prévention des fautes Tolérance aux pannes Élimination des fautes Prévision des fautes

fety) et la maintenabilité (maintenability). [16][2]

La disponibilité est définie, dans le temps, pour un instant donné. Elle correspond à la probabilité pour laquelle le système fonctionne correctement à tout moment. Elle permet de vérifier si le système est prêt à être utilisé dans l'immédiat.

La fiabilité est définie en termes d'intervalle de temps. Elle garantit la continuité du fonctionnement du système pendant une période relativement longue sans la moindre interruption.

La sécurité correspond à ce qu'il n'y ait rien de catastrophique qui se produise lors de l'occurrence d'une panne temporelle ; par exemple, lors de l'envoi d'êtres humains dans l'espace. La sécurité consiste à assurer aussi l'intégrité et la confidentialité. L'intégrité correspond à l'évitement d'altérations non désirées (inattendues) de l'information. La confidentialité correspond à l'évitement de divulgations non autorisées de l'information.

La maintenabilité définit le degré d'aptitude à réparer un système qui a échoué. Un système hautement maintenable a automatiquement un très fort degré de disponibilité.

Par ailleurs, si nous parlons de sureté de fonctionnement, il faut parler des entraves (qui sont les *fautes*, les *erreurs* et les *défaillances*) ainsi que les moyens permettant d'assurer la sureté de fonctionnement (*Prévention*, *tolérance*, *élimination* et *prévision*). Les sections suivantes abordent ces deux points. Le tableau 2.1 [17] décrit la sureté de fonctionnement à travers ses attributs, ses entraves et les moyens permettant de l'assurer.

TABLE 2.2 – Sureté de fonctionnement et contrôle des fautes

	Eviter les fautes ou les pannes	**Accepter les fautes ou les pannes**
Fournir la sureté de fonctionnement	*Prévention*	*Tolérance*
Analyser la sureté de fonctionnement	*Élimination*	*Prévision*

2.3 Les pannes (Défaillances)

La construction d'un système sûr nécessite le contrôle des fautes à travers la *prévention*, la *suppression* (*élimination*), la *prévision* ou *la tolérance*. La ***prévention*** se fait en évitant les fautes de conception et d'implémentation ou celles qui surviennent durant le déroulement. La ***suppression*** ou l'*élimination* consiste à réduire le nombre ou la sévérité des défaillances. La ***prévision*** est l'estimation de la présence, de la création et des conséquences des pannes [12]. Nous expliquons plus loin le quatrième point qui est la tolérance aux pannes. Le tableau 2.2 [17] décrit ces quatre mécanismes et leurs liens avec la sureté de fonctionnement d'un coté et la faute d'un autre coté.

En effet, on dit qu'un *système est défaillant* lorsque le service délivré ne remplit plus la ou les fonctions du système. Ceci est du à la présence d'erreurs dans l'état du système. Ainsi, la défaillance est définie par la déviation du système de sa spécification. Un système n'échoue pas toujours de la même façon. Une défaillance peut survenir à la suite d'un comportement arbitraire d'un composant (pannes byzantines), sur le canal de communication ou à cause du partitionnement du système... [2]

On appelle ***faute*** dans un système, la défection de bas niveau qui peut causer une *erreur*. Trouver la cause d'une ***erreur*** dans le système est une tâche importante. Par exemple, un médium de communication de mauvaise qualité peut endommager facilement les paquets des données. L'*erreur* peut mener vers la ***défaillance*** (l'***échec*** ou la ***panne***). Un système échoue lorsqu'il perd sa propriété de correction.

2.3.1 Caractéristiques des fautes

Les fautes sont caractérisées par leurs ***natures***, leurs ***origines*** et leurs ***persistances temporelles***. [16]

Selon la ***nature de la faute***, nous distinguons

— Les *fautes accidentelles* qui surviennent ou sont créées par inadvertance.

— Les *fautes intentionnelles* qui sont produites de façon délibérée.

La faute peut avoir des ***origines*** diverses qu'on peut cerner en deux classes :

— L'origine *phénoménologique* de la faute qui peut être soit des fautes humaines soit des fautes physiques.

— La faute peut venir des *frontières du système*. Dans ce cas, la faute peut être interne (structure fautive du système) ou externe (interactions avec l'environnement). Il s'agit des défauts de conception lors de la phase de création ou des fautes opérationnelles qui surviennent pendant l'exploitation du système.

La ***persistance temporelle*** définit trois types de fautes [2] :

— Les fautes *permanentes* qui sont indépendantes des conditions momentanées (internes ou externes). Elles continuent à exister jusqu'à la réparation du composant défaillant. Par exemple, le bug dans le logiciel ou le crash d'un disque dur.

— Les fautes *intermittentes* sont d'une durée limitée. Elles se produisent puis disparaissent plusieurs fois. Par exemple le mauvais contact d'un connecteur mal placé. Ce genre de fautes sont difficiles à diagnostiquer.

— Les fautes *transitoires* apparaissent une seule fois dans le système puis disparaissent à jamais. Par exemple un oiseau qui vole autour du transmetteur peut parasiter le signal transmis et ainsi causer la perte de bits.

2.3.2 Classification des défaillances

Nous présentons dans cette section les idées et concepts décrits dans [18] et [19] pour classifier les défaillances selon le degré de sévérité.

La moins sévère des défaillances est le ***plantage*** ou le ***crash***. Cette défaillance se produit quand le serveur fonctionne correctement puis s'arrête subitement et de façon prématurée. L'exemple le plus répandu dans les machines mono-processeurs est lors du blocage du système d'exploitation ; la seule solution est de relancer (rebooter) le système.

L'autre de classe de défaillance est l'***omission***. Elle survient lorsque le serveur n'arrive plus à traiter les requêtes qui arrivent de l'extérieur. Il peut s'agir d'un échec de réception des messages entrants (omission de réception), ou un échec d'envoi des messages (omission d'envoi).

Les ***défaillances temporelles*** sont plus sérieuses que l'omission. Il s'agit d'une réponse qui n'arrive pas dans les délais. En d'autres termes, l'arrivée ne respecte pas un intervalle dans le temps prédéfini. Cette défaillance est considérée comme une défaillance temporelle.

Un autre type de défaillances peut survenir lorsque la réponse du serveur est incorrecte. A ce moment, on parle de ***défaillance de réponse***. Elle peut se produire lorsque la valeur de la réponse est fausse. Par exemple, dans le cas du Web où un moteur de recherche retourne des résultats qui n'ont rien à voir avec les termes de recherche [2]. L'autre cas de défaillance de réponse est la défaillance de transition d'état. Il s'agit d'un serveur qui réagit de façon inattendue à une requête. Par exemple, lorsque le serveur reçoit un message qu'il ne reconnaît pas et aucune mesure n'a été prévue pour ce genre de situations [2].

La défaillance la plus sévère est la défaillance ***arbitraire***, appelée aussi ***byzantine*** [20][21]. Ce dernier terme est issu du problème des généraux byzantins [21]. Dans ce cas, le serveur ou un de ses composants se comporte de façon arbitraire à des moments arbitraires. Les pannes byzantines peuvent être "naturelles" ou "malicieuses". Elles sont dites naturelles si par exemple une erreur physique non détectée (sur une transmission de message, en mémoire, sur une instruction ...) ou une erreur logicielle qui mènent vers la non vérification des spécifications. Elles sont dites malicieuses dans le cas d'un comportement visant à faire échouer le système (sabotage, virus, ...).

2.3.3 Quelques défaillances connues

Dans cette section, nous présentons quelques pannes rencontrées souvent dans le domaine des systèmes répartis. Nous citons ci-dessous la *défaillance d'un canal de communication*, la *défaillance arrêt sur panne* et le *partitionnement*.

La défaillance d'un canal est due à un canal de communication non fiable. Ce canal peut perdre, altérer, dé-séquencer ou dupliquer les messages. Pour les deux derniers cas, il suffit seulement de numéroter les messages. En cas de perte, il faut envoyer de nouveau le message si le canal n'est pas suspendu. [14]

La défaillance arrêt sur panne (appelées en anglais ***fail-stop failure***) rapproche les deux extrêmes classes de défaillances (le crash et l'arbitraire). On associe généralement à cette défaillance la notion de proces-

seur ou serveur à arrêt sur panne. Ces serveurs mettent les processeurs à l'arrêt dès qu'une panne est détectée ou qu'un comportement d'un composant est arbitraire. Ce genre de machines n'existe pas physiquement car il est impossible de les réaliser. Par contre, il est possible de mettre en œuvre une abstraction qui possède une probabilité élevée d'avoir un comportement de *fail-stop process* [22][23].

Le partitionnement est la situation dans laquelle le système se retrouve coupé en deux ou plusieurs parties n'ayant plus aucune possibilité de communiquer entre elles. Toute partie, croyant être la seule à fonctionner, peut effectuer des traitements qui devraient se faire par exemple en exclusion mutuelle. Les différentes parties du système vont ainsi diverger et il en résulte un état incohérent après recouvrement du partitionnement. Ceci rend ce problème parmi les plus difficiles à résoudre [14].

2.4 Techniques de tolérance aux défaillances

La tolérance aux pannes est un art et une science qui permet de construire des systèmes qui continuent à fonctionner même en présence de pannes [24]. En effet, il faut fournir au système les moyens de détection et de prise en compte des pannes.

Rendre le système tolérant aux défaillances est une tâche très difficile et compliquée, à cause du caractère imprévisible ainsi que la possibilité de pannes multiples. Par exemple, une défaillance peut survenir pendant l'exécution de la procédure de recouvrement d'une autre panne. [14]

Par contre, la recherche dans le domaine de la tolérance aux pannes couvre une large gamme d'applications telles que : les systèmes à temps réel incorporés, les systèmes de transactions commerciales, les systèmes de transports et les systèmes militaires et spatiaux. [24]

Il existe des mécanismes et techniques de tolérance aux pannes simples et d'autres plus élaborés dont nous citons : les points de reprise, le comptage, le time-out, la duplication (réplication passive ou active), la diffusion atomique, le consensus, les votes, les quorums ou le concept de groupe. D'un autre coté, la mise en place du concept de processus à arrêt sur défaillance (fail-stop process) [22] permet aussi de simplifier la résolution de certains problèmes pour la conception de systèmes de calcul tolérants aux pannes. Nous détaillons ceci ci-après :

2.4.1 Le Comptage

Cette technique est utilisée pour tolérer certaines pannes ou problèmes dus aux canaux de communication. La perte de messages et leur déséquencement peuvent être détectés très facilement en numérotant tous les messages émis sur chaque canal. Une rupture de séquence alerte le récepteur qui demande alors la retransmission du message perdu.

2.4.2 Délai de garde (Timeout)

Dans certains modèles, cette technique est utilisée pour permettre de détecter les pannes sur le système distribué. Appelée aussi délai de garde, elle fixe une limite au temps d'attente d'une réponse ou d'un événement devant être réalisé par un autre site.

Par exemple, si les délais de transfert des messages sont bornés, le time-out est une bonne technique pour détecter les défaillances des sites et des canaux de communication.

Cette technique nécessite parfois de supposer que les horloges des processeurs sont synchronisées. Par exemple, si deux processus ne se mettent pas d'accord sur le fait qu'un troisième est à l'arrêt peut avoir des conséquences désastreuses.

2.4.3 Tolérance aux pannes matérielle

La majorité des conceptions tolérantes aux pannes a été orientée vers la construction de machines qui récupèrent automatiquement des défaillances générées par les fautes aléatoires qui se produisent dans les composants matériels.

Les premières techniques consistaient à dupliquer et fragmenter l'information. La duplication et la fragmentation des données sont utilisées pour pouvoir tolérer des défaillances dans un système. La lecture et l'écriture d'une donnée dupliquée restent possibles tant qu'il y ait suffisamment de copies disponibles, ce qui préserve l'accessibilité ainsi que la cohérence des différentes copies [14].

Ces techniques ont été améliorées et on parle beaucoup plus de redondance au lieu de duplication. Généralement, le système de calcul est fragmenté en modules qui agissent en tant que régions (zones) de limitation d'expansion des pannes. Chaque module est sauvegardé avec une redondance protectrice de telle sorte que, si le module échoue, d'autres puissent assumer sa fonction. Des mécanismes spéciaux sont ajoutés pour détecter les défaillances et implémenter le recouvrement.

Deux approches générales pour le recouvrement à partir des pannes matérielles sont utilisées : masquer les fautes et le recouvrement dynamique. [24][2][13]

On peut masquer les fautes par la redondance. Le terme redondance est utilisé actuellement à la place de duplication. On parle de redondance d'information ou de temps ou de la redondance physique.

La **redondance de l'information** consiste à rajouter des bits supplémentaires à l'information. La **redondance du temps** consiste à exécuter une action puis encore la même action s'il y a nécessité. La **redondance physique** consiste à rajouter des équipements supplémentaires.

2.4.4 La réplication

Une autre approche, qui ressemble à la précédente et qui est plus efficace, est basée sur la réplication de processus. Elle consiste à créer des copies multiples des processus sur des processeurs différents. La réplication peut être active, semi-active ou passive, des détails sur ces techniques peuvent être trouvés dans [16].

Sur le plan application, les bases de données réparties et la réplication des données sont reconnues aujourd'hui comme moyens efficaces pour augmenter la disponibilité et la fiabilité des bases de données. De plus la réplication peut contribuer favorablement à l'amélioration des performances en utilisant les copies locales voire les copies plus proches.

Toutefois, ces avantages sont contraints par un problème majeur de cohérence mutuelle des copies. La gestion des copies en termes de propagation des mises à jour est ainsi nécessaire. La charge induite peut entraîner un impact significatif sur le système. Celle-ci ne doit pas altérer de façon excessive le temps de réponse global. En d'autres termes il s'agit de garantir la cohérence mutuelle dans les délais acceptables.

De plus, on peut noter que le temps nécessaire, pour qu'une mise à jour prenne effet sur toutes les copies, peut varier selon les méthodes de gestion. Les copies peuvent ainsi présenter un décalage les unes par rapport aux autres. Ce retard, appelé temps de latence, définit la période ou une donnée peut être utilisée (lecture) alors qu'elle ne reflète pas toutes les modifications antérieures de la base de données.

Les stratégies de réplication visent à garantir une cohérence forte (strong consistency) entre les copies d'un objet répliqué. De façon informelle, ceci revient à assurer que l'état de chaque copie soit identique. La réplication passive et la

réplication active sont les deux stratégies de références.

La réplication active (active replication ou state machine approach) se définit [25] par la symétrie des comportements des copies d'un composant répliqué. Chaque copie joue un rôle identique à celui des autres. En d'autres termes, les messages d'entrée sont traités par l'ensemble des répliques de façon concomitante afin de garder leurs états internes étroitement synchronisés. Cette technique permet en particulier de mettre en œuvre un vote sur les sorties, afin de se prémunir contre les défaillances arbitraires.

La réplication passive [25] distingue deux comportements d'un composant répliqué : la copie primaire et les copies secondaires. La copie primaire est la seule à effectuer tous les traitements. Les copies secondaires, oisives, surveillent la copie primaire. Les répliques secondaires sont mises à jour à l'aide de points de reprise. En cas de défaillance de la copie primaire, une copie secondaire devient la nouvelle copie primaire.

La réplication semi-active [25] se situe à mi-chemin entre la réplication active et passive. La copie primaire est appelée *leader* et les copies secondaires *suiveurs*. Les copies secondaires ne sont plus oisives. Cette stratégie consiste à recevoir et traiter les messages d'entrée par toutes les copies, mais seul le leader fournira les messages de sortie tant qu'aucune panne n'est détectée. Un copie primaire traite une réplique dès qu'elle la reçoit. Une copie secondaire doit attendre une notification du leader pour pourvoir traiter une requête.

2.4.5 Recouvrement et points de reprise

Consiste à prendre régulièrement une image globale du système. Ces images, appelées aussi *points de contrôle*, *snapshots* ou *checkpoints*, permettent de reprendre l'exécution après l'arrêt causé par la défaillance. La reprise se fait à partir de la dernière image cohérente sauvegardée. La difficulté est d'assurer la synchronisation de tous les sites, i.e, à prendre une image globale en synchrone alors que le système distribué est par essence asynchrone. Au fait, chaque site capture son propre état et le sauvegarder dans un endroit stable et accessible. L'ensemble des états locaux doit constituer un état global représentatif et cohérent du système.

En cas de défaillance, chaque site est amené à suspendre son exécution et réaliser un retour en arrière pour reprendre son exécution à partir du dernier point cohérent global sauvegardé.

Dans le cadre de notre travail, nous nous intéressons à cet aspect là et plus précisément au calcul de ces points et de la façon de les enregistrer pour coordonner entre les différents sites dans le but d'obtenir un état global consistant. Nous montrons comment déterminer ces points de contrôle (checkpoints). Un point de contrôle représente le plus récent état possible du système et, de facto, un état par lequel est passé le système ou aurait pu passer sans influer sur le résultat final. Nous explicitons plus loin la notion de cohérence ou consistance d'un point de contrôle global.

Il existe deux approches dans la conception de protocoles de ***calcul de points de contrôle*** (***checkpointing***) : La classe des protocoles de checkpointing ***non-coordonné*** (*uncoordinated checkpointing*) et la classe des protocoles de ***checkpointing coordonné*** (*coordinated checkpointing*). Une troisième classe appelée ***checkpointing induit communication*** (*communication-induced checkpointing*) qui est plutôt proche des algorithmes non coordonnés.

Les algorithmes adoptant la technique *non-coordonnée* sont simples avec un coût d'exécution très bas. Les protocoles de cette classe (appelés aussi dans la littérature Checkpointing conventionnel indépendant) ne nécessitent pas de processus coordinateur. Chaque processus sauvegarde périodiquement ses états de façon indépendante et aucun message supplémentaire n'est nécessaire.

Cette méthode offre aux processus plus d'autonomie dans la décision de prise de checkpoint. Le principal avantage de cette autonomie est que chaque processus peut prendre un checkpoint au meilleur moment qui lui convient (par exemple quand la quantité d'information sauvegardée est petite pour réduire le surcoût du checkpointing local). Mais cette technique a aussi plusieurs inconvénient. Le premier est la possibilité que l'effet domino [6][15] peut survenir. Le deuxième inconvénient est que certains points de reprise pris ne sont d'aucune utilité. Le troisième est lié à la capacité de stockage car chaque processus est forcé de maintenir plusieurs points de contrôle ce qui nécessite une fonction de ramasse-miettes (garbage collector) pour récupérer les points de contrôle qui ne sont d'aucune utilité. Le dernier inconvénient est cité dans [26] montrant

que ce genre de protocoles ne sont pas adéquats aux systèmes ayant de fréquentes sorties.

La technique *induite communication* est utilisée pour éviter l'effet domino survenu dans la première technique (non coordonnée). Aucun message de contrôle supplémentaire n'est utilisé et donc aucun coût supplémentaire de synchronisation n'est ajouté au calcul distribué. Les algorithmes non-coordonnés utilisent des configurations de communication ou de l'*information embarquée* (*piggybacked information*) pour obtenir la cohérence.

Toutefois, l'indépendance des processus est contrainte ne garantissant pas l'éventuelle progression de la ligne de recouvrement. Par conséquent, les processus pourraient être forcés à prendre des checkpoints supplémentaires.

Pour les protocoles *coordonnés*, les messages de contrôle (qui viennent s'ajouter à ceux de l'application) sont utilisés pour réaliser la coordination. On parle aussi de protocoles à synchronisation explicite [5]. Un processus se charge d'orchestrer l'application de checkpointing afin d'obtenir un état global cohérent de telle façon que l'ensemble des derniers points de contrôle pris par les processus forment cet état. Ainsi, après défaillance, le recouvrement se fait à partir du dernier checkpoint pris (le plus récent). De ce fait, les processus nécessitent de garder un seul point de contrôle permanent dans un support stable, ce qui réduit le surcoût de stockage et élimine le besoin de la fonction de ramasse-miettes (garbage collector). Il existe deux grandes orientations dans la conception des algorithmes de checkpointing coordonnés. La première [27][28][29] consiste à bloquer le calcul distribué pour engager un nombre minimum de processus dans le checkpointing. Pour la seconde, elle consiste à ne jamais bloquer le calcul sous-jacent [30][31][32].

Nous nous intéressons à cette technique dans le cadre des travaux de cette thèse. Plus de détails sont donnés dans le chapitre 5.

2.4.6 Groupes de processus

Cette approche consiste à organiser un certain nombre de processus dans un groupe. Les groupes ainsi constitués peuvent être dynamiques.

On distingue deux types de groupes : les **groupes plats** et les **groupes**

hiérarchiques. Dans un groupe plat, tous les processus sont égaux et la décision est prise de façon collective. Dans un groupe hiérarchique, il faut que l'un des membres du groupe joue le rôle de chef ou coordonnateur.

Cette approche nécessite une politique pour créer, supprimer des groupes et de rajouter des membres aux groupes.

2.4.7 Communication de groupe fiable

L'objectif de cette approche est de garantir que les messages sont délivrés à tous les membres. D'où la notion de communication multipoint (multicast). Le multicast doit être fiable en assurant que le message envoyé au groupe multicast de processus doit être délivré à chaque membre du groupe.

2.4.8 Diffusion atomique

Pour qu'une diffusion soit atomique, il faut qu'au bout d'un temps fini, tous les sites reçoivent la même valeur (le message émis) ou personne ne la reçoit. Pour assurer la vivacité de la diffusion atomique, il faut utiliser un protocole de calcul de la liste de présence qui permet de restreindre le groupe aux sites corrects. Ceci limite les attentes. Il peut donc arriver qu'un site soit exclu d'un groupe pour cause de lenteur et il devra exécuter une procédure de réadmission pour réintégrer le groupe.

2.4.9 Consensus

Le consensus est un mécanisme qui permet de n'entreprendre que des actions préservant la cohérence du système. Il n'y a consensus entre les sites (d'un groupe donné) sur une valeur ou une action à accomplir que si tous les sites s'entendent sur la même valeur au bout d'un temps fini.

Il est impossible de résoudre le consensus d'une manière déterministe dans un système asynchrone sujet à un seul crash de processus [33]. Cette impossibilité est due à la difficulté de déterminer si un processus est actuellement défaillant ou s'il est simplement très lent. [34]

Dans le paradigme du consensus, tous les processus corrects proposent une valeur et doivent prendre une décision irrévocable sur une seule valeur appartenant à l'ensemble des valeurs proposées [33]. Le consensus est composé de deux

primitives : *proposer(v)* et *decider(v)* où v est une valeur extraite de l'ensemble des valeurs proposables par les processus.

Le consensus est caractérisé par les quatre propriétés suivantes :

— **Terminaison :** chaque processus correct décide éventuellement une valeur au bout d'un temps fini.

— **Validité uniforme :** si un processus décide une valeur v alors v a été proposée par un processus. Ceci définit le domaine de définition de v.

— **Intégrité uniforme :** chaque processus décide au plus une fois.

— **Accord :** deux processus corrects ne décident pas deux valeurs différentes. Ceci permet de définir la sémantique du consensus.

Les protocoles assurant la diffusion atomique permettent de résoudre le problème du consensus et vice-versa. Pour les systèmes asynchrones ayant des comportements byzantins, ces problèmes n'ont pas de solution. [14]

2.4.10 Les Quorums

Le mécanisme de quorums et son cas particulier le vote, sont utilisés dans le problème d'exclusion mutuelle et de la duplication. Il représente l'une des rares techniques résistant au problème du partitionnement du système.

Le mécanisme repose sur le nombre de permissions que le site doit obtenir pour accomplir son action. Un tel ensemble de permissions s'appelle un quorum. Ces ensembles sont définis de sorte qu'un site soit le seul à pouvoir exécuter son action. Par exemple, l'intersection de quorums associés à deux sites distincts est toujours non vide.

2.5 Conclusion

La tolérance aux défaillances (pannes) est une propriété très importante à garantir lors de la conception des systèmes répartis. Dans ce chapitre, nous avons donné un aperçu général sur les systèmes distribué et le domaine de sûreté de fonctionnement qui englobe le domaine de la tolérance aux pannes. Nous avons présenté la terminologie nécessaire (fiabilité, disponibilité, types d'erreurs, classification des défaillances, ...). Nous avons abordé les mécanismes et les techniques de tolérance aux pannes les plus utilisées.

Nous pouvons dire que la stratégie de la tolérance aux pannes vise à éviter les

défaillances à travers les mécanismes de détection et par la suite, le rétablissement du système [17]. D'où, l'objectif principal des travaux de cette thèse : détecter les pannes et retablir l'application à travers le recouvrement arrière.

Dans le prochain chapitre, nous abordons l'environnement mobile. Nous explicitons la notion de mobilité et discutons de son impact sur la conception des protocoles de tolérance aux défaillances.

Chapitre 3

L'environnement Mobile

3.1 Introduction

Avec le nombre croissant des utilisateurs équipés de terminaux mobiles et les avancées technologiques dans les moyens de communication sans fil, la mobilité dans les systèmes est devenue une réalité. Ceci nécessite d'assurer à ces utilisateurs un déplacement libre tout en maintenant une connectivité avec le reste du réseau. [35]

L'évolution rapide de la technologie dans les domaines de la communication et de la télécommunication (le cellulaire avec ses différentes générations, les systèmes basés satellite, ...) et des réseaux locaux sans fil, a rendu l'extension des systèmes répartis possible. Cette extension permet à l'usager d'accéder à l'information n'importe où et n'importe quand. Ces utilisateurs sont munis d'unités mobiles de diverses configurations et équipés d'interface de communication sans fil [3].

Dans ce chapitre nous abordons les caractéristiques de l'environnement mobile. Nous donnons quelques définitions, et nous citons les principales contraintes apparues avec cet environnement. Nous présentons les deux grandes classes de réseaux mobiles, plus particulièrement ad hoc. Enfin, nous abordons les concepts de systèmes distribués et de tolérance aux pannes relativement à ces environnements.

3.2 Définition de l'environnement mobile

Un environnement mobile est un système composé de sites mobiles et qui permet à ses utilisateurs d'accéder à l'information indépendamment de leurs positions géographiques.

Les réseaux mobiles ou sans fil peuvent être classés en deux classes : *les réseaux avec infrastructure* et *les réseaux sans infrastructure*. Les réseaux avec infrastructure qui utilisent le modèle de la communication cellulaire, et les réseaux sans infrastructures appelés également Ad hoc. Ce dernier environnement introduit de nouvelles contraintes aux réseaux conventionnels.

3.2.1 Contraintes de mobilité

Les unités mobiles sont caractérisées par les déplacements, les déconnexions et re-connexions, et des capacités de calcul, d'énergie et de stockage peu importantes. Par ailleurs, la bande passante du support de communication sans fil est limitée. Dans ce qui suit, nous donnons plus de détails sur ces contraintes.

Hétérogénéité

Les unités mobiles peuvent être de diverses configurations matérielles et logicielles. Sur le plan matériel, nous pouvons avoir des différences dans la taille (petite comme un smartphone ou smartwatch ou grande comme un laptop).

Nous pouvons aussi avoir des unités avec des capacités de transmission différentes et variable ce qui fait que les liens de communication soit parfois asymétrique (unidirectionnels).

Cette hétérogénéité matérielle peut être observée aussi au niveau de la capacité de traitement au niveau des vitesses des processeurs et les capacités des mémoires.

La façon de se déplacer d'un nœud (par exemple lent, rapide, ...) peut le distinguer des autres.

Par ailleurs, l'hétérogénéité logicielle est constatée par exemple au niveau des systèmes d'exploitation ou les programmes qui s'exécutent au niveau de chaque unité mobile.

Ressources machine

Les unités mobiles peuvent avoir des capacités de stockage (disque), de mémorisation et de calcul plus ou moins modestes.

Malgré l'évolution de la technologie, les *ressources* machine des unités mobiles restent toujours *limitées* par rapport aux unités fixes. En effet, il est difficile de comparer les capacité d'un smartphone par exemple aux capacités d'un serveur ou station de calcul fixe.

Absence d'infrastructure

Cette contrainte est spécifique aux réseaux ad hoc. Ces réseaux ont la particularité de s'auto-organiser et s'auto-administrer. Ils ne se reposent sur aucune infrastructure fixe.

Bande passante

Les unités mobiles utilisent le réseau de communication sans fil qui est loin d'être aussi fiable et rapide que le réseau filaire (réseau en fibre optique par exemple). Ceci rend la *bande passante* peu disponible ; ce qui induit des délais de transmissions et de réceptions plus grands.

Par ailleurs, un temps supplémentaire de recherche de l'unité mobile (notée MH pour le mot anglais *Mobile Host*) est nécessaire. Ceci peut ralentir considérablement le calcul distribué.

Topologie dynamique

La principale caractéristique des systèmes mobiles est *le changement de localisation* ce qui complique le routage des messages et augmente le temps de communication et la complexité des messages. Les messages envoyés d'un nœud à un autre peuvent être re-routés à cause de la déconnexion ou du déplacement de l'unité mobile destinataire. Ce qui nécessite dans la majorité du temps une phase de recherche qui peut être très coûteuse en temps.

Il existe des stratégies pour localiser une MH. Une MH peut demander un service au moment où elle est connectée et en attendant la réponse elle peut se déplacer entre plusieurs cellules. Ainsi, pour délivrer un tel service, il faut connaître la nouvelle localisation de la MH. [36]

Le changement de localisation rend les structures logiques (arbre, étoile, graphe, anneau, grille,...), sur lesquels sont basés les résolutions de certains problèmes dans les systèmes répartis conventionnels, non adaptées [35].

L'énergie

Sur le plan *énergie*, les unités mobiles sont alimentées par une batterie qui se recharge (source d'énergie autonome). Les différents composants consomment de l'énergie ainsi que l'émission et réception de messages. Le système, durant la période de basse activité, éteint certains ou tous les composants. Ceci mettra à l'arrêt la majorité des fonctions de son système et reste seulement à l'écoute des messages entrants. Ce mode est appelé le mode veille (doze-mode) où le calcul ne reprend que lors de la réception d'un message dans le seul but d'économiser de l'énergie. Ceci amène à rechercher des solutions pour économiser l'énergie et éviter de solliciter la machine en mode de veille.

Déconnexions

Les systèmes mobiles sont aussi caractérisés par les *déconnexions fréquentes* et parfois sans avertir les autres processus. La déconnexion peut être aussi temporaire comme elle peut être permanente. Ainsi, la période de déconnexion est arbitraire. Ces déconnexions sont soit volontaires soit involontaires. Dans le premier cas (volontaires), elles peuvent être pour des raisons propres à la MH, dues au déplacement de la station ce qui engendre un protocole de handoff, se trouvant dans une zone non couverte ou pour économiser de l'énergie (mode veille). Dans le deuxième cas (involontaires), la déconnexion peut résulter d'une panne ou même de la vulnérabilité du réseau de communication sans fil.

Les conséquences de la déconnexion en mode veille diffère des autres déconnexions, car la MH reste joignable à partir du reste du système et toute autre station du système peut la remettre en mode normal si nécessaire.

Lors de la déconnexion, seuls les événements locaux sont exécutés sur le nœud mobile. C'est à dire, il n'y aura aucune émission ou réception de messages durant cette période. Dans le cadre du calcul mobile, il faut prendre en considération ce problème lors de la conception des protocoles et des applications.

Sécurité limitée

Les réseaux ad hoc sont par nature plus sensibles aux attaques qui menacent les données transmises. De plus, les techniques conventionnelles utilisées pour faire face à ces attaques ne sont plus applicables dans les réseaux ad hoc à cause de limitation de ressources connues dans ce type de réseau (puissance de calcul et mémoire).

Perte d'information

Contrairement aux réseaux fixes où les unités statiques, la probabilité de perdre l'information est plus importante dans un environnement mobile, cela se justifie par la mobilité des unités.

3.2.2 Avantages des réseaux mobiles

Malgré les contraintes introduites, les réseaux mobiles ont beaucoup d'avantages apportés au monde de l'informatique et de la communication. Nous citons ci-dessous quelques avantages de la mobilité.

La flexibilité

La possibilité de travailler là où l'utilisateur le désire, là où il doit être, là où se trouve son travail. Contrairement aux unités qui composent les réseaux statiques (les réseaux fixes), l'unité mobile est plus petite, plus légère. Ceci permet aux utilisateurs de se déplacer tout en restant connectés au réseau. Ainsi, la mobilité apporte un nouveau type d'information (l'informatique indépendante du lieu).

Applications nouvelles

La mobilité et la portabilité offertes par les environnements mobiles, ont permis le développement de nouvelles classes d'applications : Accès à des bases de données en tout lieu, émission et réception de messages à partir de n'importe quel endroit, etc.

Coûts réduits

Un réseau ad hoc est attrayant en coût et en temps d'installation.

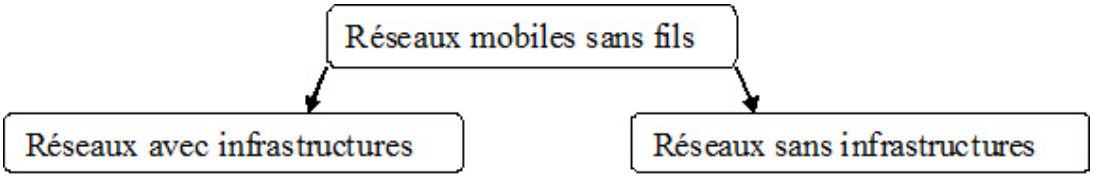

FIGURE 3.1 – Classification des réseaux mobiles.

3.3 Classes de réseaux mobiles

Comme nous l'avons déjà expliqué, il existe deux classes d'environnement mobile comme indiqué sur la figure 3.1. L'environnement **avec infrastructure** appelé *cellulaire* et l'environnement **sans infrastructure** appelé *ad hoc*.

3.3.1 Environnement mobile cellulaire.

Ce modèle est composé de deux ensembles d'entités distinctes : les *sites fixes* d'un réseau de communication filaire et les *sites mobiles*. Certain sites fixes, appelés *notée station de support mobile* (notés MSS pour *Mobile Support Station*) ou *station de base* (notée BS pour *Base Station*) sont munis d'une interface de communication sans fil pour la communication directe avec les sites ou unités mobiles (notée MH pour *Mobile Host*), localisés dans une zone géographique limitée, appelée *cellule* [37] (voir la figure 3.2). Les MSS représentent l'infrastructure du réseau.

Ainsi les MSS jouent le rôle de cellule et une MH est connectée à une seule cellule à un instant donné. En se déplaçant d'une cellule à une autre elle change de MSS. Une cellule est une zone logique ou géographique couverte par une MSS.

Les unités mobiles peuvent émettre et recevoir des messages à travers la MSS. Alors que les sites fixes sont inter-connectés entre eux à travers un réseau de communication filaire, généralement fiable et d'un débit élevé, les liaisons sans fil ont une bande passante limitée qui réduit sévèrement le volume des informations échangées [38].

Une unité mobile ne peut être, à un instant donné, directement connectée qu'à une seule station de base. Elle peut communiquer avec les autres sites à travers la station à laquelle elle est rattachée. L'autonomie réduite de sa source d'énergie peut lui occasionner une déconnexion du réseau. Sa reconnexion peut alors se faire dans un environnement nouveau voire dans une nouvelle localisation

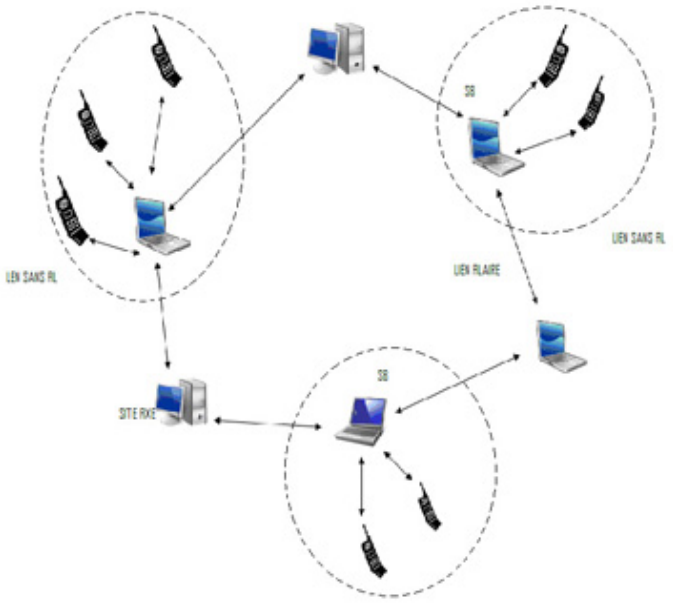

FIGURE 3.2 – Réseau mobile cellulaire.

[38].

Ainsi, un système réparti dans un tel environnement est décrit comme étant un ensemble de stations fixes et un ensemble de stations mobiles. Certains processus s'exécutent sur les MH et d'autres sur les MSS. De plus, les MSS jouent le rôle de points d'accès pour les sites mobiles à travers des lignes de communication sans fils.

3.3.2 Environnement mobile ad hoc

L'origine du mot "*Ad hoc*" est latine. Il s'agit d'un adjectif de la locution latine *ad* signifiant "*à*" et du pronom démonstratif *hoc* signifiant "*celà*", c'est à dire dans le sens de "*à cet effet*". En d'autres termes, qui convient à un usage déterminé. [39] Ainsi, les réseaux mobiles ad hoc peuvent exister temporairement pour répondre à un besoin ponctuel de communication.

L'étude de ce genre de réseaux a commencé au début des années 1990. Par contre les premières applications sont apparus bien après et étaient dans le domaine militaire. Par la suite d'autres applications civiles ont investi ce domaine.

Les réseaux ad hoc sont décrits par le groupe de travail Mobile Ad hoc NETworks (MANET) [40]. Ce sont des réseaux sans fils et sans infrastructure fixe comme l'illustre la figure 3.3.

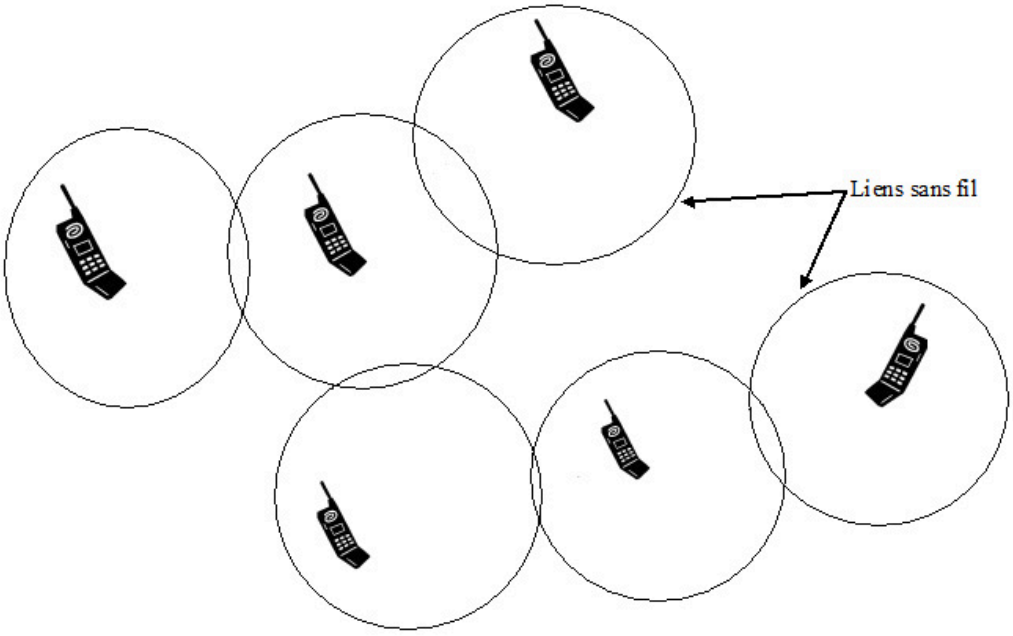

FIGURE 3.3 – Réseau mobile ad hoc.

En effet, chaque nœud combine les rôles de client et routeur. En d'autres termes, un nœud peut communiquer directement avec d'autres nœuds à condition qu'ils soient à sa portée radio. Par ailleurs, il peut servir de relais pour permettre à des nœuds se trouvant hors de la portée radio les uns des autres afin qu'ils communiquent. Il est possible d'avoir une série de relais pour faire communiquer deux nœuds.

Les réseaux ad hoc sont idéals pour les applications caractérisées par une absence ou la non fiabilité de l'infrastructure préexistante, tel que :

La communication tactique et tout ce qui concerne les applications militaires comme le partage des informations sur la position des cibles ou sur l'ennemi. Ceci peut minimiser le risque d'interception des messages et évite d'installer des infrastructures de communications qui sont cibles de l'ennemi.

Les opération de secours : Lors de catastrophes naturelles (incendies, inondations... etc.) afin de résoudre le problème de communication là ou l'installation filaire nécessite un coût très fort en temps.

Les systèmes de conférences où les réseaux ad hoc facilitent l'échange et

le partage d'information entre les participants à une conférence.

Les sites de patrimoines comme les sites archéologique, les anciens musées, châteaux, ... où il est généralement interdit d'installer des câbles (réseaux filaires) vu le risque de nuisance sur de tels environnements.

En privé : pour assurer un libre échange de données ou pour réaliser des connexions en peer-to-peer...

Du point de vue architectural, il existe deux classes de réseaux ad hoc : *plats* et *hiérarchiques*.

Les réseaux ad hoc plats sont caractérisés par le fait que tous les nœuds jouent le même rôle sans distinction. En d'autres termes, ils ont les mêmes responsabilités. Généralement, les architectures plates sont plus flexibles et plus simples. De plus, les solutions conçues sur ce genre d'architectures peuvent être appliquées à une large classe de réseaux.

Les réseaux ad hoc hiérarchisés sont destinés aux réseaux de grande taille (à grande échelle). L'hiérarchisation ou la clusterisation consiste à découper le réseau en un ensemble de groupes nommés *clusters*. Les clusters sont formés d'un nombre nœuds proches géographiquement dont l'un d'eux est le chef comme le montre la figure 3.4.

On distingue trois types d'éléments dans un cluster : le *chef* appelé *clusterhead*, la *passerelle* appelée (*gateway*) et les *membres de cluster*.

Le *clusterhead* (voir nœuds 1, 2, 4 et 7 sur la figure 3.4) est élu par les autres membres selon des critères bien définis au préalable. Il a pour mission principale d'administrer le cluster. Il joue aussi un rôle très important dans le routage inter et intra cluster.

La *passerelle* qui peut être *directe* (nœud appartenant à deux clusters différents comme par exemple le nœud 9 sur la figure 3.4) ou *indirecte* (nœud ayant comme voisin un nœud appartenant à un autre cluster comme par exemple les nœuds 3 et 5 sur la figure 3.4) permet de réaliser la connexion entre deux clusters.

Les autres nœuds (qui ne sont ni *clusterhead* ni *gateway*) sont appelés *membres*.

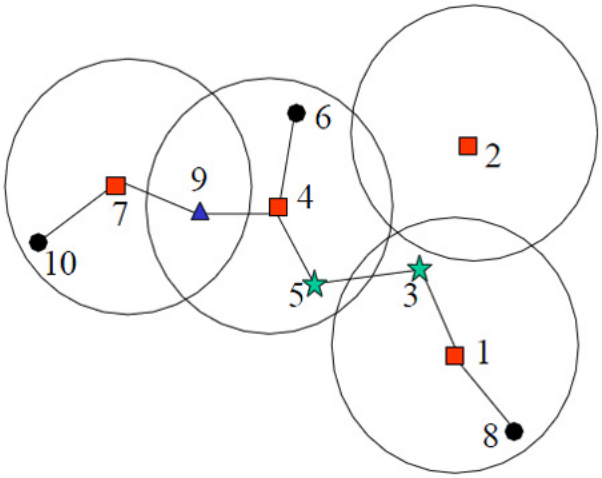

FIGURE 3.4 – Réseaux ad hoc avec clusters.

3.4 Système réparti en environnement mobile

L'intérêt de l'étude des systèmes distribués mobiles est dicté par l'évolution de la technologie des communications qui a eu toujours une influence considérable sur l'informatique et plus précisément sur les réseaux informatiques.

Un système réparti en environnement mobile doit offrir aux usagers des services comparables à ceux habituellement offerts par les systèmes statiques sans compromettre leur aptitude de se déplacer.

L'étude des problèmes de synchronisation et de communication dans les systèmes répartis a été abordée pour des architectures comprenant uniquement des sites statiques. En l'absence de panne de processeurs ou de liaison de communication, la structure topologique du réseau d'interconnexion ne varie pas. Cette structure est généralement modélisée par un graphe non orienté où les sommets symbolisent les unités de calcul (sites) et les arêtes les liaisons physiques entre les sites. C'est ainsi que les algorithmes distribués, développés jusque-là, sont souvent basés sur une topologie logique de type arbre, maille, anneau, ... où chaque lien correspond à un chemin dans le réseau physique sous-jacent.

Dans un environnement mobile où les sites sont capables de se déplacer entre plusieurs localités tout en restant connectés au réseau à travers des liaisons sans fil, la structure physique change souvent. L'utilisation de structures logiques comme plateforme de base pour la construction d'algorithmes distribués dans ce type d'environnement devient trop lourde, à cause des re-configurations continuelles engendrées par le déplacement des nœuds.

Les caractéristiques physiques intrinsèques des unités mobiles et leur capacité de fonctionnement dans les modes « veille » et « déconnexion », soulèvent des problèmes nouveaux dont il faut tenir en compte lors de la conception d'algorithme distribué en environnement mobile.

La conception de logiciels est fortement influencée par les caractéristiques déjà cités : source d'énergie limitée, grand risque de perte de l'information, peu de capacité de stockage.

3.5 Tolérance aux défaillances en environnement mobile

A cause des caractéristiques citées ci-dessus jusque là inconnues dans les systèmes distribués conventionnels et en plus de la fragilité des unités mobiles (le risque de dommages physiques est très élevé : la perte, le vol, la température, l'humidité, l'eau, l'exposition aux rayons ...) rendent le système plus vulnérable aux défaillances ; parfois c'est toute la machine portable qui est amenée à être remplacée. De ce fait, l'unité mobile n'est pas considérée comme un endroit stable de stockage des informations importantes.

Les méthodes traditionnelles de tolérance aux fautes ne peuvent pas être appliquées directement dans l'environnement mobile. D'un autre coté, la tolérance aux fautes dans cet environnement (à cause de la vulnérabilité et les autres contraintes) est de plus en plus importante.

Les premiers travaux dans ce domaine consistaient à adapter les mécanismes et techniques qui existaient déjà en tenant compte des contraintes de la mobilité. Nous citons à titre indicatif le travail de [36] sur l'impact de la mobilité sur le calcul distribué ou les travaux parus dans [3] qui ont étudié la tolérance aux pannes et l'ordre causal dans cet environnement.

D'autres travaux se sont axés sur d'autres techniques, tel que le calcul des points de contrôle dont les détails peuvent être trouvés dans [6] et [11].

Par contre de nouveaux problèmes sont apparus pour cet environnement tel que la localisation [41] ou la gestion des handoff [6].

Sur un autre axe, la mobilité génère de fréquentes déconnexions. On distingue deux (2) types de déconnexions : volontaires et involontaires. La première est décidée par l'utilisateur et la seconde est le résultat de l'absence de couver-

ture réseau sans fil. Cette problématique a donné naissance à deux nouveaux concepts : les détecteurs de connectivité et les détecteurs de déconnexions [34]. Les études dans ce domaine s'appuient sur deux techniques : le système de communication de groupe et les consensus [42].

3.6 Conclusion

Dans ce chapitre, nous avons présenté l'environnement mobile et les nouvelles contraintes qu'il introduit. Nous avons aussi expliqué l'importance de la tolérance aux pannes dans ce genre d'environnement.

Dans le prochain chapitre, nous abordons une des techniques de tolérance aux pannes. Il s'agit de la détection de pannes.

Chapitre 4

Détection de Pannes

4.1 Introduction

Les pannes (ou fautes) sont classifiées dans la littérature en : transitoire, intermittente ou permanente. Notre étude se focalise sur les pannes permanentes. Ce type de fautes continue à persister jusqu'à ce que le nœud en panne soit remplacé. Nous traitons particulièrement les pannes franches appelées aussi «crash». Cette dernière apparaît quand un nœud donné s'arrête de façon prématurée.

Pour une application distribuée, le dysfonctionnement d'un processus peut engendrer des conséquences graves qui se propagent sur toute l'application. En effet, déterminer le processus qui n'est plus en mesure de continuer le calcul, permet de lancer des opérations additionnelles comme par exemple le processus de recouvrement.

La détection des nœuds défaillants est une tâche importante et en même temps très complexe. En effet, il est difficile de distinguer entre un processus lent et un processus défaillant. Ainsi, dans [4], le paradigme de détecteurs de pannes non fiables est défini. Les *détecteurs de pannes non fiables* (en anglais : *unreliable failure detectors*) [4] représentent l'une des solutions définies pour implémenter un système tolérant aux fautes. Ces détecteurs consistent à gérer une liste de nœuds suspectés d'être défaillants.

Dans ce chapitre nous présentons le domaine de la détection des pannes. Nous expliquons l'origine du problème, les propriétés d'un détecteur de pannes, la classification des techniques de détection ainsi que la présentation et la discussion de quelques protocoles qui existent dans la littérature.

4.2 L'origine du problème

De nombreux services dans les systèmes répartis nécessitent la réalisation d'un accord (accord sur une valeur, accord sur un message, sur les processus en panne...). Ils suivent un schéma commun : tous les nœuds doivent arriver à une décision commune, laquelle dépend de la nature exacte du problème. Des travaux récents ont mis en évidence le lien qui existe entre les problèmes nécessitant l'obtention d'un accord (élection, diffusion ordonnée, validation atomique, etc.) et le problème abstrait du consensus. En effet, ces problèmes peuvent être résolus en utilisant une solution au problème du consensus comme brique de base.[16]

4.2.1 Le problème du consensus

Il s'agit d'un problème fondamental dans les systèmes distribués, il permet à un ensemble de nœuds, chacun possédant sa propre valeur initiale de décider d'une manière irrévocable sur une de ces valeurs. De manière plus formelle, le consensus est satisfait par les propriétés suivantes [16] :

- La terminaison : Si au moins un processus correct lance le consensus, tout processus correct décide au bout d'un temps fini.
- L'intégrité : Tout processus décide au plus une fois (une décision prise est définitive).
- L'accord : La valeur décidée est la même pour tous les processus corrects.
- La validité : Toute valeur décidée est l'une des valeurs proposées.

4.2.2 Impossibilité FLP

Fischer, Lynch et Paterson [33] ont prouvé que le consensus est impossible à réaliser dans un système asynchrone dès qu'au moins un nœud tombe en panne définitive. De façon non formelle la preuve est fondée sur le fait que dans un système asynchrone, il est difficile de différencier un nœud très lent d'un nœud en panne.

Afin de surmonter cette difficulté, Chandra et Toueg [4] ont introduit la notion de détecteur de défaillances non fiables.

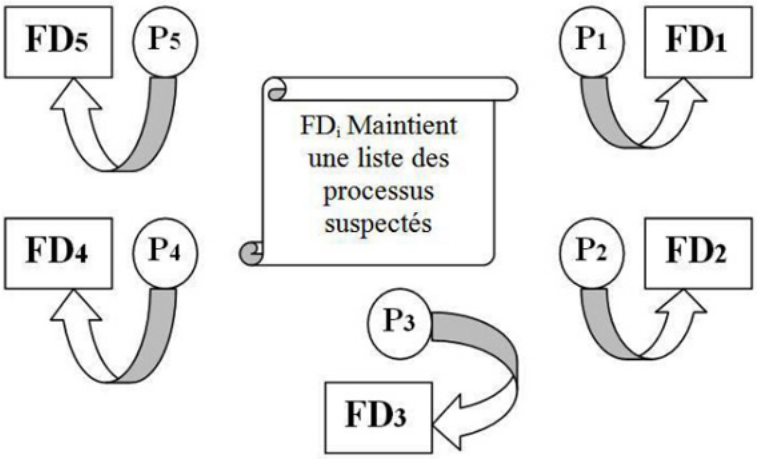

FIGURE 4.1 – Détecteurs de défaillances non fiables

4.3 Les détecteurs de pannes non fiables

Un détecteur de défaillances est un service réparti composé de détecteurs locaux attachés à chaque nœud. En effet, chaque nœud du système exécute localement un programme de détection qui consiste à explorer les autres nœuds dans le but de trouver ceux qui sont défaillants. Un détecteur fournit sur demande la liste des nœuds qu'il soupçonne d'être défaillants. Les divers détecteurs locaux coopèrent entre eux pour établir cette liste [7]. En outre, les listes de suspects sont sujettes à des mises à jour. Un nœud peut être rajouté ou retiré de cette liste. Les détecteurs de pannes sont illustrés dans le schéma représenté par la figure 4.1.

Cependant, dans le contexte de l'exécution asynchrone [1] du système distribué, il est difficile de distinguer entre les nœuds dont les processus sont lents de ceux qui sont défaillants. Par conséquent, le protocole de détection de pannes peut commettre de fausses suspicions. En d'autres termes, un nœud sain peut être considéré à tort comme défaillant [43]. D'autre part, un nœud peut aussi échouer à déterminer tous les nœuds qui sont effectivement défaillants.

4.3.1 Les propriétés d'un détecteur de pannes

Un détecteur de défaillances est caractérisé par deux propriétés : *La propriété de complétude* (en anglais *completness*), qui correspond au fait qu'un nœud défaillant doit être suspecté et la *La propriété de justesse* appelée aussi *propriété de*

précision (en anglais *accuracy*), qui signifie qu'un processus correct ne doit pas être suspecté. Pour plus de précision ces propriétés peuvent se décliner en différentes sous propriétés. Avant de présenter leurs définitions formelles, quelques notations s'imposent :

- L'historique d'un détecteur de pannes est noté H ; où $H(i,t)$ est la valeur du détecteur de défaillances du nœud i à l'instant t dans H. En d'autres termes, elle représente la liste des suspects du nœud i à l'instant t.
- Un détecteur de défaillance D fournit des informations sur le modèle de défaillance F. Tel que $crashed(F)$ donne l'ensemble des processus qui sont réellement défaillants et $correct(F)$ donne l'ensemble des processus qui sont réellement corrects. D est une fonction qui associe à chaque modèle de défaillance F un ensemble d'historique de détecteurs de défaillances $D(F)$. $D(F)$ est l'ensemble de tous les historiques pouvant exister durant les exécutions avec le modèle F et le détecteur de défaillances.
- Π représente l'ensemble des nœuds dans le système.

La Complétude

Complétude forte (Strong completeness) : Il existe un instant après lequel un nœud défaillant est suspecté d'une manière permanente par tout nœud correct. D'une manière formelle, un détecteur satisfait la complétude forte si,

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

Complétude faible (Weak completeness) : Il existe un instant après lequel un nœud défaillant est suspecté d'une manière permanente par au moins un nœud correct. D'une manière formelle, un détecteur satisfait la complétude faible si,

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

La Justesse

Justesse forte (Strong accury) : Aucun nœud n'est suspecté avant qu'il ne devienne défaillant. D'une manière formelle, un détecteur satisfait la justesse forte si,

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

Précision finalement forte (Eventual strong accuracy) : Il existe un instant après lequel tout nœud correct n'est pas suspecté par un autre nœud correct. D'une manière formelle, un détecteur satisfait la justesse finalement forte si,

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

Justesse faible (Weak accury) : Certains nœuds corrects ne sont jamais suspectés. D'une manière formelle, un détecteur satisfait la justesse faible si,

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

Justesse finalement faible (Eventual weak accuracy) : Il existe un instant après lequel il existe des nœuds qui ne sont pas suspectés par un autre nœud correct. D'une manière formelle, un détecteur satisfait la justesse finalement faible si

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

4.3.2 Les classes de détecteur de défaillances

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 [4]. Leurs définitions et les notations correspondantes sont données dans le tableau 4.1.

TABLE 4.1 – Les classes de détecteurs de pannes

	Justesse			
	Forte	Faible	Finalement forte	Finalement faible
Complétude Forte	P	S	$\Diamond P$	$\Diamond S$
Complétude Faible	Q	W	$\Diamond Q$	$\Diamond W$

— La classe des *détecteurs de pannes parfaits* est notée P. Les protocoles de cette classe sont caractérisés par le fait que toutes les pannes sont détectées de manière exacte. Aucune fausse suspicion n'est à signaler dans cette

classe. Il est quasiment impossible de concevoir un protocole appartenant à cette classe et ce à cause de l'asynchronisme du système.

— la classe notée S est appelée classe de *détecteurs de pannes forts*. Ils assurent une complétude forte et une justesse faible. Ainsi, tous les nœuds défaillants sont suspectés et au moins un nœud correct n'est pas suspecté.

— Les détecteurs de pannes qui assurent une forte complétude et une justesse finalement forte font partie de la classe des *détecteurs de pannes finalement parfaits* et sont notés $\diamondsuit P$. Tous les nœuds défaillants sont détectés de manière exacte au bout d'un temps fini (complétude forte) et toute fausse suspicion est corrigée dans un délai borné (justesse finalement forte).

— Un détecteur de pannes de la classe $\diamondsuit S$ est appelé *détecteur de pannes finalement fort*. En effet, il assure une complétude forte et une précision finalement faible. En d'autres termes, tous les nœuds défaillants sont suspectés et à partir d'un certain instant au moins un nœud correct n'est plus suspecté.

— La classe des *détecteurs de pannes quasi-parfaits* (notée Q) assure une précision forte et une faible complétude.

— La classe des *détecteurs de pannes faibles* qu'on note W assure une complétude faible et une justesse faible. Ainsi, on peut considérer dans cette classe qu'un nœud est défaillant (suspecté) alors qu'il fonctionne correctement et ne jamais suspecter un nœud réellement défaillant.

— Un détecteur de pannes de la classe $\diamondsuit Q$ est appelé *détecteur de pannes finalement quasi-parfait*. En effet, il assure une complétude faible et une précision finalement forte. En d'autres termes, un nœud défaillant est détecté par au moins un des nœuds corrects et à partir d'un certain instant aucun nœud correct n'est plus suspecté.

— La classe notée $\diamondsuit W$ et appelée *détecteur de pannes finalement faibles* est semblable à la classe W seulement la justesse est finalement faible.

4.3.3 Classification des techniques de Détection de Pannes

Nous distinguons deux grandes approches dans la conception des protocoles de détection de pannes : l'approche basé sur les *battements de cœur* (ou *heartbeat* en anglais) [44] et l'approche basée sur les *requêtes/réponses* (ou *pinging* en anglais) [45][46].

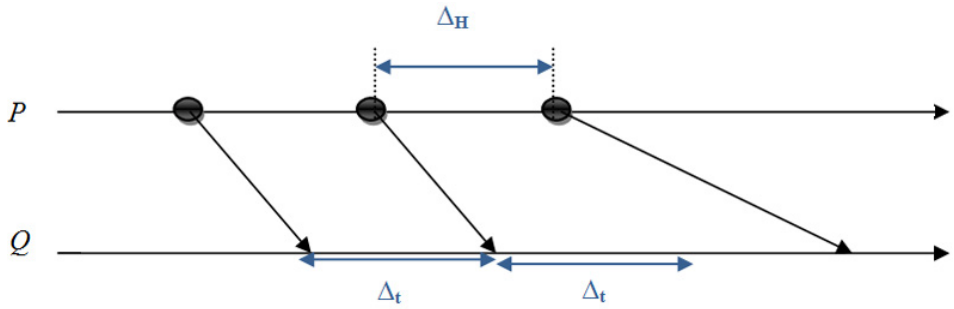

FIGURE 4.2 – Technique basée sur les battements de cœeur.

Technique basée sur les heartbeats

Cette approche [44] exige pour chaque nœud P d'envoyer périodiquement (Chaque Δ_H unités de temps) un message de vivacité (appelé *battement de cœur*) aux autres nœuds impliqués dans le calcul. Quand le message est reçu par un nœud Q, un temporisateur (timer noté Δ_t) dédié à P est déclenché (armé) par Q afin d'attendre le prochain message de vivacité. Si aucun message n'est reçu par Q après expiration du délai dédié à P, le nœud Q rajoute le nœud P à sa liste de suspects. Le nœud P est supprimé de cette liste, dès qu'il re-donne signe de vie (un message de vivacité reçu de sa part). [7] La figure 4.2 illustre cette technique.

Technique basée sur les pinging

L'idée de base qui est derrière cette proche [45][46] est que chaque nœud Q envoie périodiquement (Chaque Δ_H unités de temps) un message (appelé *requête* ou *Query*) à un nœud P qui doit répondre immédiatement à travers un message appelé *réponse* (ou *Reply*). Si le nœud Q ne reçoit pas de réponse de la part de P après un certain délai (Δ_t), il rajoute P à sa liste de suspects. Le nœud P est retiré de cette liste dès qu'une réponse est reçue de sa part. [7] La figure 4.3 illustre cette technique.

Techniques d'implémentation asynchrones

Il existe plusieurs travaux dans le domaine asynchrone qui se basent sur les techniques décrites ci-dessus sans utiliser de temporisateur. En d'autres termes, éviter la notion de délai pour les messages. Parmi ces travaux, on peut citer les travaux de Grève *et al.* [47] qui se basent sur la technique de pinging. Chaque

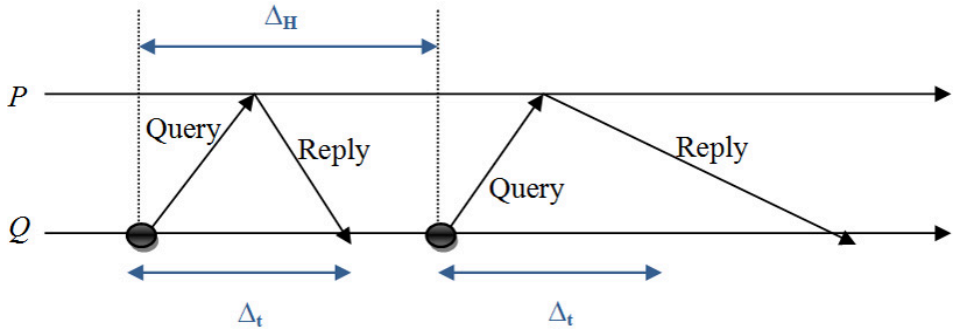

FIGURE 4.3 – Technique basée sur les requêtes/réponses.

nœud Q envoie un message *Query* à tous les autres nœuds, ensuite il attend $(n - f)$ réponses tel que f est le nombre maximal de nœuds qui peuvent tomber en panne. Q ajoute les f nœuds n'ayant pas répondu à sa liste de suspects.

Il y a aussi les protocoles asynchrones orientés technique de battement de cœur. Nous citons à titre d'exemple les travaux de Arantes *et al.* [48]. Le principe de base consiste à ce que chaque nœud P embarque dans le message de vivacité une structure dynamique appelée *path*. Cette structure contient une séquence d'identités de nœuds. Elle représente le chemin par lequel est passé ce message. Ainsi, chaque nœud se trouvant dans *path* est vivant pour le moment. Lors de la réception d'un tel message par un nœud Q, il vérifie si son identité apparaît dans *path*. Si elle n'apparaît pas, il s'ajoute dans *path* et transfère le message à tous ses voisins. Autrement (si Q apparait dans *path*), il saura que son message a été transféré à travers un cycle, tous les nœuds qui apparaissent après Q dans *path* sont mutuellement accessibles et il les considère comme vivants.

4.4 Travaux antérieurs

Plusieurs protocoles de détection de pannes ont été définis dans la littérature. Le premier protocole a été présenté par Chandra et Toueg[4]. Il se base sur la technique de battements de cœur. Il a été conçu pour les réseaux filaire puis il a subi plusieurs améliorations dans différents travaux tel que les protocoles de Aguilera *et al.* [49][50], Raynal et Tronel[51], Mostéfaoui *et al.* [52] et Larrea *et al.* [43].

Tous ces travaux ont ciblé les réseaux conventionnel, c'est à dire dans le contexte de l'environnement filaire. Cependant, même s'il n'a pas été dit dans

les papiers cités ci-dessus, les algorithmes de détection de pannes basés sur les battements de cœur sont pratiques pour tolérer la mobilité des nœuds [45] mais au détriment de leur efficacité, car certaines contraintes ne sont pas prises en compte.

Dans le contexte des MANETs, et à notre connaissance, il existe peu de travaux dans la littérature. Par exemple, Bhatti et Conan[53] ont proposé un protocole de détection de pannes basé sur les battements de cœur qui gère les déconnexions des nœuds. Seulement, les auteurs supposent que le voisinage de chaque nœud reste inchangé pendant la période de sa déconnexion. Ceci représente une hypothèse trop forte qui restreint la mobilité des nœuds. En outre, Friedman et Tcharny [54] ont proposé d'estampiller logiquement les messages de battement de cœur afin d'augmenter le justesse du protocole. Ainsi, un vecteur d'horloges est embarqué dans chaque message de vivacité. Toutefois, ce mécanisme fait augmenter la taille des messages transmis dans le réseau.

Parmi les protocoles qui se basent sur la technique pinging (requête/réponse), nous citons les travaux de Conan *et al.* [45] et Sens *et al.*[46]. Dans [45], les auteurs supposent que le nombre et les identités des nœuds impliqués dans le calcul distribué sont connus au préalable par tous. Ceci empêche l'arrivée de nouveaux nœuds dans le système. Cette hypothèse est tout à fait contradictoires avec le contexte MANETs où chaque nœud ne possède jamais de connaissance globale du reste du réseau. De plus, le protocole utilise la diffusion comme mécanisme pour transmettre les messages de vivacité, ce qui fait augmenter le nombre de messages circulant dans le réseau. Ceci influe négativement sur les performances globales du système (très forte consommation de bande passante et d'énergie). Dans [46], les auteurs supposent que chaque nœud a au minimum d voisins et le nombre maximal de nœuds pouvant tomber en panne dans le voisinage est f. Les valeurs de d et f sont supposées connues par tous les nœuds.

Arantes *et al.* [48] ont proposé un protocole plus efficace qui prend en considération la grande majorité des contraintes des MANETs. Par contre, ce protocole met l'accent pricipalement sur la détection du partitionnement. En général, la grande partie des protocoles de détection de pannes proposés dans la littérature sont conçus pour résoudre le problème du consensus [55], [56], [57].

Notre étude a exploré plusieurs autres travaux en environnement mobile. Nous citons les travaux de Jiménez *et al.* [58], Delporte-Gallet *et al.* [59], Leners *et al.* [60], Grève *et al.* [47], Lim et Conan [61], Guerraoui *et al.* [62], Mostéfaoui

et al. [63]...

Nous donnons ci-après quelques détails sur les principaux protocoles de détection de pannes.

4.4.1 Algorithme de Chandra et Toueg (1996)

Chandra et Toueg ont proposé dans [4] le premier algorithme de détection de pannes utilisant la technique HeartBeat (battement de cœur) qui implémente la classe de détecteur $\diamondsuit P$ assurant la complétude forte et la justesse finalement forte. Ce protocole est destiné aux réseaux filaires. Ci-dessous quelques détails sur ce protocole.

Hypothèses

- Le réseau de communication est supposé être connexe.
- Chaque paire de nœuds est supposée connectée par un lien de communication fiable et bidirectionnel.
- Le modèle de communication considéré est partiellement synchrone.

Principe de fonctionnement

L'algorithme est basé sur trois tâches :

1. Chaque nœud i envoie périodiquement un message i_is_alive pour informer les autres nœuds du système, de sa vivacité.
2. Chaque nœud j gère un timeout pour chaque nœud du système. Lors de l'expiration du timeout associé à i, noté $\Delta_i(j)$, sans réception du message de vivacité de i, le nœud j ajoute i à sa liste de nœuds suspects.
3. Lors de la réception par j d'un message i_is_alive, alors que i est suspecté par j, le nœud j retire i de sa liste de nœuds suspects et incrémente le timeout associé à i ($\Delta_i(j)$).

Discussion

Avantages :
- Dans cet algorithme le problème de consensus est résolu grâce à l'utilisation d'un modèle partiellement synchrone.
- Une gestion personnalisée des timeouts. Le détecteur attribue un timeout à chaque nœud.

Inconvénients : — Le protocole génère un nombre très important de messages de vivacité. En effet, chaque nœud envoie périodiquement un battement de cœur à tous les autres nœuds. En d'autres termes, le protocole fait une diffusion générale (inondation) dans le réseau. Ceci peut ne pas garantir un bon passage à l'échelle (sacalability).

— L'incrémentation du délai d'attente à chaque réception du message de vivacité hors délai et sans jamais le décrémenter peut générer des timeout de valeurs très grande. Ainsi, un nœud peut tomber en panne et n'est jamais détecté vu que le délai risque de n'expirer qu'après la fin du calcul.

— Les listes de suspects sont sauvegardées localement et il n'y a aucun échange entre les nœuds dans la perspective de synchroniser les différentes listes. Ceci risque de ne pas garantir la propriété de complétude.

4.4.2 Algorithme de Mostéfaoui *et al.* (2004)

Dans [52], les auteurs ont proposé d'hybrider les deux techniques de détection de pannes (Heartbeat et pinging). L'objectif est de bénéficier des avantages des deux techniques.

Hypothèses

— Le système distribué est composé d'un ensemble fini de n nœuds tel que $n \geq 3$.

— Le nombre maximal de nœuds défaillant, noté Nf, est connu au préalable $(1 \leq Nf < n)$.

— Le graphe sous-jacent est toujours connexe.

Principe de fonctionnement

L'algorithme construit deux listes de suspects : la liste PS ($Partial\ Synchrony$) qui contient les nœuds suspectés en utilisant la technique de battement de cœur et la liste MP ($Message\ Pattern$) obtenue en utilisant la technique pinging. Ces deux listes sont gérées comme suit :

1. Chaque nœud i émet périodiquement un message $Query\_Alive$ à tous les autres nœuds puis attend $(n - Nf)$ réponses. Il associe à chaque nœud du système un timeout.

2. Lors de la réception par j d'un message $Query\_Alive$ de la part de i, il envoie un message de réponse muni de la liste not_rec_from contenant la liste des Nf nœuds n'ayant pas répondu à sa dernière requête. Si i appartient à la liste des suspects alors il est retiré de cette liste et son timeout est incrémenté d'une unité.
3. Après réception des $(n - Nf)$ réponses, j construit la liste des suspects MP qui est l'intersection des listes not_rec_from reçues.
4. Si le timeout associé à i par j expire et j ne reçoit pas de $Query\_Alive$ de la part de i ; j rajoute i à la liste PS.
5. La liste de suspects finale est l'intersection des listes MP et PS.

Discussion

Avantages : — Il s'agit du premier protocole qui a réussi a hybrider les deux techniques de détection.
— L'échange des listes de suspects permet de propager rapidement l'information sur une suspicion ou la correction d'une fausse suspicion.

Inconvénients : — La technique pinging engendre beaucoup de messages. De plus, chaque nœud fait une diffusion générale de sa requête (à tous les autres nœuds du système). Ceci peut causer un problème de passage à l'échelle comme pour le protocole précédent.
— L'envoi des listes dans les messages de réponses rend les messages trop volumineux. Ceci influe de façon négative sur la bande passante.
— Le protocole souffre du même problème que celui du protocole de Chandra et Toueg [4] concernant l'incrémentation continue des délais d'attente pour les nœuds lents.

4.4.3 Algorithme de Larrea *et al.* (2005)

Les auteurs de [43] ont proposé un protocole tout en définissant une nouvelle classe de protocoles de détection de pannes. Il s'agit de la classe notée $\diamondsuit C$ et appelée *détecteurs finalement cohérents* (en anglais : *eventually consistent detectors*). Cette nouvelle classe est une combinaison entre les caractéristiques de la classe $\diamondsuit S$ et la caractéristique de sélection de leader (l'élection du leader est faite par l'algorithme proposé par les mêmes auteurs dans [64]).

Ce protocole crée une sorte de hiérarchie dans le réseau et classe les nœuds en deux niveaux : *leader* et *nœud normal*. Le leader est chargé de la construction de la liste des suspects, et les autres nœuds ne font que transmettre leurs messages de vivacité à leur leader. Le nœud leader est le seul apte à prendre une décision envers ces nœuds, soit les suspecter (les rajouter à sa liste des suspects) soit les considérer comme corrects.

Hypothèses

— Chaque paire de nœuds est supposée connectée par un lien de communication fiable et bidirectionnel.
— Le modèle du système est partiellement synchrone.

Principe de fonctionnement

L'algorithme est constitué de cinq tâches concurrentes :

1. Si un nœud p est leader alors, il diffuse périodiquement la liste des suspects à tous les nœuds du système.
2. Quand le leader reçoit un message de vivacité i_is_alive, et si ce processus appartient à sa liste de suspects, alors il corrige sa fausse suspicion et enlève i de cette liste. Il lui incrémente aussi la valeur du timeout.
3. Si le processus j est un leader et le timeout associé à i a expiré, alors il le rajoute à sa liste de suspects.
4. Si le processus j n'est pas un leader, il envoie le message de vivacité à son leader.
5. Quand i reçoit une liste de suspects de j, il vérifie si j est son leader. Si c'est le cas, il met à jour sa propre liste.

Discussion

Avantages : — Dans cet algorithme, le consensus sur la mise d'un processus dans la liste des suspects est assuré. Ceci est justifié par le fait que le leader est le seul qui peut construire cette liste. En effet, le leader est le seul habilité à prendre la décision d'ajouter un processus à sa liste ou pas.

— Cette solution possède aussi comme avantage la réduction des communications entre les processus, et l'économie des ressources que ce soit en termes d'énergie ou en bande passante car la communication se fait uniquement avec le leader.

Inconvénients : — Les processus reçoivent périodiquement la liste des suspects du leader, ce qui leur permet d'avoir une même vision que celle du leader vis-à-vis des défaillances. Mais si le leader commet de fausses suspicions alors tous ceux qui reçoivent la liste vont suspecter à tort des processus corrects et pourraient prendre des décisions envers ses derniers même s'ils ne sont pas réellement défaillants.

— Par ailleurs, l'algorithme reste efficace pour un nombre fini de processus mais ne permet pas l'arrivée de nouveaux nœuds dans le réseau.

— Nous pouvons constater que le leader envoie la liste des suspects à tous les autres nœuds. Ainsi, plus on a de processus qui se considèrent comme leader, plus le nombre de messages envoyés est grand. De plus, les messages reçus d'un émetteur qui n'est pas leader sont ignorés, donc beaucoup de messages sont envoyés inutilement.

4.4.4 Algorithme de Bhatti et Conan (2005)

Dans [53], les auteurs ont présenté leur protocole HBFD qui est constitué de trois types de détecteurs : défaillances, déconnexions et partitions. Le but est de distinguer les nœuds défaillants, de ceux seulement partitionnés ou déconnectés. Ce protocole a été conçu pour l'environnement mobile.

Hypothèses

— Les liens peuvent perdre des messages.

— Initialement chaque processus connaît G (la topologie globale du réseau).

— La topologie ne change pas de manière implicite, un processus déconnecté doit se reconnecté aux mêmes voisins.

Principe de fonctionnement

HBFD tente d'apporter des améliorations au protocole proposé par Aguilera *et al.* [65]. L'algorithme est basé sur deux tâches concurrentes :

1. Un nœud i émet périodiquement à tous ses voisins un message de battement de cœur $(HB, path)$.
2. A la réception d'un tel message, le nœud j incrémente le nombre de Heart-Beat pour tous les processus qui apparaissent après j dans $path$. Ensuite, si j n'est pas présent plus d'une fois dans path, il s'ajoute à $path$, et fait suivre le message de battement de cœur à tous ses voisins. Sinon cela indique que le message est parti une première fois de j et qu'il a traversé un certain nombre de nœuds pour revenir à j, ou que les processus qui apparaissent après j dans $path$ sont mutuellement accessibles par j.

La construction de la liste des suspects est réalisée par le détecteur de partition.

Le détecteur de déconnexions utilise un détecteur de connectivité qui offre la possibilité de prévoir les déconnections involontaires, donc de les anticiper et de les préparer de décider si la requête de l'utilisateur peut être transmise ou non sur le réseau sans fil. 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. Le détecteur de déconnexion exécute trois tâches concurrentes :

1. La première tâche s'exécute suite à la notification d'un changement de mode émis par le détecteur de connectivité, telle que :
   – $Mode =' d'$ indique le mode déconnecté (un message de déconnexion est diffusé aux voisins).
   – $Mode =' c'$ indique le mode connecté (un message de re-connexion est diffusé aux voisins).
2. La deuxième est en écoute des messages de déconnexion. Le processus ajoute l'émetteur à son ensemble de processus vus déconnectés. Pour tolérer les duplications de messages, les déconnexions sont numérotées.
3. La troisième tâche est en écoute des messages de re-connexion. Le processus enlève l'émetteur de son ensemble de processus vus déconnectés. Pour tolérer les duplications de messages, les re-connexions sont numérotées.

Le détecteur de partitions : La construction de la liste des suspects est réalisée par ce détecteur. Pour gérer cette liste, une variable k, qui représente le

nombre de battement de cœur devant être reçus pendant une période de temps, est utilisé.

1. La première tâche de l'algorithme pour un nœud p opère la détection de défaillance. Elle consiste en premier lieu à comparer le nombre de battement de cœur à un certain seuil k. Lorsqu'un processus est suspecté, il est aussitôt mis dans l'ensemble des processus défaillants f_p.

   Ensuite, le graphe est parcouru pour détecter si des processus deviennent partitionnés à cause de la nouvelle défaillance : la recherche est effectuée en enlevant les processus déconnectés (ensemble d_p), défaillants (ensemble f_p) et déjà partitionnés (ensemble P_p) à Π s'il n'existe plus dans ce graphe de chemin équitable entre j et un processus i, alors i est ajouté à l'ensemble P_p.

   La boucle suivante effectue l'opération similaire, mais cette fois-ci pour détecter les fausses suspicions. Dans ce cas, le processus faussement suspecté i est enlève de f_p et l'algorithme teste si des processus partitionnés suite à cette fausse suspicion sont dans P_p par erreur : il s'agit de trouver un chemin entre j et un processus s de P_p en supposant que s est à nouveau vivant. A la fin de cette première tâche, si l'un des ensembles f_p, d_p ou P_p a été modifié, alors le détecteur de partitions envoie une notification de possible changement de vue de la partition.

   Enfin, la détection de défaillances doit respecter les propriétés de complétude forte et de précision forte finale.

2. La deuxième traite les notifications de détection de déconnexion provenant du détecteur de déconnexion. Puisque les connexions sont sures, lorsque i est dit défaillant et une déconnexion est détectée, alors i est enlevé de l'ensemble de processus défaillants et ajouté à l'ensemble de processus déconnectés (et de même pour les partitions).

3. La troisième tâche traite les notifications de détection de re-connexion provenant du détecteur de déconnexion. Puisque les connexions sont sures, lorsque i est dit défaillant et une re-connexion est détectée alors i est enlevé de l'ensemble de processus défaillants et ajouté à l'ensemble de processus déconnectés (et de même pour les partitions).

Discussion

Avantages : — Dans cet algorithme nous pouvons distinguer les nœuds défaillants, de ceux seulement partitionnés ou déconnectés. Donc l'ajout des détecteurs de déconnexions et de partitionnements est un pas énorme pour l'amélioration et l'élimination d'un grand nombre de fausses suspicions.

— Le principe d'utiliser un détecteur de connectivité est de surveiller les ressources locales pour éviter les manques (de bande passante, d'énergie...etc.) rendent les communications difficiles. Dans le but d'éliminer les messages inutiles et de réduire le nombre de message dans le réseau.

— Enfin, l'utilisation d'un détecteur de partitionnement dispose des avantages. Car lorsqu'un processus i est suspecté (le nombre de battement de cœur de i et inférieure à une valeur donnée), il est mis dans l'ensemble des processus défaillants. Donc le parcours du graphe (pour détecter les processus qui deviennent partitionnés à cause de cette nouvelle défaillance), évite les fausses suspicions.

Inconvénients : — Cet algorithme ne traite pas le cas du changement du graphe sous-jacent qui est un événement fréquent vus qu'un processus déconnecté doit se reconnecté aux mêmes voisins.

4.4.5 Algorithme de Jiménez *et al.* (2006)

L'algorithme [58] implémente le détecteur de défaillance Ω en supposant que chaque nœud ne connait au départ que son identité. L'idée est un peu différente des autres algorithmes. Elle consiste à élire le nœud le moins suspecté par les autres qui construira une liste de nœuds corrects.

Hypothèses

— Les nœuds n'ont aucune connaissance préalable sur l'ensemble des nœuds dans le système.

— Pour envoyer les messages, il y a une primitive de diffusion (broadcasting) qui permet à un nœud i d'envoyer simultanément le même message au reste des nœuds du système.

— Le nombre de défaillances pouvant avoir lieu dans le système n'est pas limité et non connu.

— Chaque paire de nœuds est connectée par un lien bidirectionnel, qui peut perdre et dupliquer des messages.

Principe de fonctionnement

Avant de décrire le protocole, nous donnons les principales structures de données utilisées :

— $Punish_i$: Il s'agit d'ensemble de paires (v, r) de i, et r est un nœud que i croit être vivant et $v \geq 0$ est le nombre de fois que r a été suspecté dans le système.

— $Mship_i$: Cette structure représente l'ensemble des identités des nœuds connus par i, y compris sa propre identité.

L'algorithme est destiné à l'environnement mobile et se base sur la technique heartbeat. Il est constitué des tâches ci-dessous.

1. Chaque nœud i diffuse périodiquement un message de vivacité pour informer les autres qu'il est toujours vivant.

   Ce message est de type $broadcast(i, punish_i)$.

2. Chaque nœud j attribue un délai d'attente à tout autre nœud i. Lors de l'expiration de ce délai, le nœud j, incrémente la valeur du délai d'attente de i, et le retire de la liste $punish_j$, puis diffuse la liste $(j, punishj)$. Ensuite, il change le leader en sélectionnant la plus petite paire dans la liste $punish_j$.

3. À la réception d'un message $(j, punish_j)$ par r, r vérifie que ce message n'a pas été déjà reçu (pour éviter la redondance), puis le diffuse immédiatement aux autres. Ensuite il vérifie si le nœud j n'appartient pas à sa liste des nœuds qu'il connait, si c'est le cas, alors il le rajoute à sa liste de connaissance et lui attribue un délai d'attente.

Discussion

Avantages : — Contrairement aux protocoles présentés précédemment où chaque nœud connaît et doit connaître toutes les identités des nœuds du système pour qu'il puisse fonctionner. E. Jiménez *et al.* [58] ont implémenté ce détecteur de défaillances sans avoir une connaissance préalable des membres du système ce qui est adéquat pour les réseaux Ad Hoc.

— L'utilisation de la diffusion (broadcasting) risque d'inonder le réseau par des messages et d'augmenter le trafic.

Inconvénients : — L'algorithme ne construit pas une liste de suspects, mais celle-ci peut être déduite à partir des listes *mship*, *punish*. La liste *mship* contient tous les nœuds connus par un nœud donné. La liste *punish* est celle qui contient les nœuds corrects.

— Forcément les nœuds qui appartiennent à la liste *mship* et n'appartiennent pas à la liste *punish* sont considérés comme suspects.

4.4.6 Algorithme de Sens *et al.* (2008)

L'algorithme présenté dans [46] représente une implémentation asynchrone de la technique requête/réponse (Pinging) dans les réseaux MANET où le nombre de nœuds est inconnu. L'algorithme met en œuvre des détecteurs de défaillances de la classe $\diamondsuit S$.

Hypothèses

— Ni la topologie, ni le nombre de nœuds dans le système ne sont connus au préalable par les nœuds.
— Les liens (communication entre voisins) sont fiables.
— Le nombre maximal de nœuds qui peuvent tomber en panne, noté Nf est connu au préalable.
— Soit d, la densité minimale d'un rang (la densité d'un rang est le nombre de nœuds dans ce rang), tel que $d > Nf$.
— Il a au moins $(Nf + 1)$ nœuds dans l'intersection de deux rangs.
— Il n'y a ni déconnexion, ni partitionnement.

Principe de fonctionnement

Avant d'expliquer le principe de base de fonctionnement de ce protocole, nous expliquons quelques notations utilisées dans cet algorithme.

— $Rang_i$: Ensemble des voisins direct de i, union i.
— $Suspected_i$: Ensemble des processus suspectés par i.
— $Mistake_i$: Ensemble des processus faussement suspectés par i.
— Rec_from_i : Ensemble des processus desquels i a reçu des réponses.

L'algorithme est constitué des trois tâches concurrentes décrites ci-dessous.

1. A chaque tour (c'est à dire périodiquement), i envoie un message $Query$ muni de la liste des suspects ($Suspected_i$) à tous ses voisins, ensuite attend $(d - Nf)$ réponses. Ces $(d - Nf)$ nœuds sont considérés corrects par i et insérées dans Rec_from_i.
2. Lorsque j reçoit un $Query$ de i, il ajoute la liste $Suspected_i$ reçue à son ensemble $Suspected_j$ et i à l'ensemble K_j (ensemble de nœuds ayant envoyé un $Query$ à j). Par la suite, il envoie une réponse dans laquelle il embarque les ensembles Rec_from_j, $Suspected_j$ et $Mistake_j$.
3. Lors de la réception des $(d - Nf)$ réponses, j fait des calculs ensemblistes pour déterminer les nœuds suspectés ($Suspected_j$), faussement suspecté ($Mistake_j$), etc. Un nœud i est considéré comme suspect s'il a été suspecté par j ou par l'un des nœuds ayant envoyé une requête ou une réponse à j sauf si i fait partie des nœuds ayant envoyé une requête ou réponse à j.

Discussion

Avantages :
- Cet algorithme est proposé pour les réseaux ad hoc et traite quelques caractéristique de cet environnement, parmi lesquelles la mobilité.
- L'envoi de message qui se fait uniquement entre voisins.

Inconvénients :
- Il y a trop d'hypothèses qui font que le réseau doit être assez dense pour que le protocole soit performant.
- Le nombre de défaillances Nf dans la réalité est imprévisible, ceci peut engendrer le blocage du protocole dans l'attente de $(d - Nf)$ réponses.
- De plus ce protocole ne traite pas les cas de déconnexions et de partitionnement.
- De plus, nous pouvons avoir un nombre important de messages engendrés par celui-ci, car le nombre de messages générés dépend du nombre de processus dans le système.

4.4.7 Algorithme de Arantes *et al.* (2010)

Dans [48], Arantes et al ont proposé un modèle pour mieux s'adapter à la dynamicité des réseaux ad hoc. Leur modèle considère que les chemins entre les nœuds sont construits dynamiquement.

Hypothèses

— Le système distribué est composé d'une infinité de processus.
— Il existe des partitions stables et finies.
— Les liens sont unidirectionnel.
— Les nœuds ne connaissent pas la topologie du réseau.

Principe de fonctionnement

Le protocole est une implémentation asynchrone de la technique de battement de cœur. Nous commençons par expliquer les principales structures de données utilisées dans ce protocole.

— $PART_i$: La partition stable associée à i.
— ST_i : Le temps de stabilisation d'un nœud stable i.
— $inPART$: L'ensemble des participants dans une partition donnée.

Le détecteur local exécute une phase d'initialisation, puis deux tâches concurrentes. Lors de la phase d'initialisation, il initialise son timer et envoie à tous ses voisins un message $ALIVE(path)$ où $path$ est une liste ordonnée d'identités de nœud qui est initialisée à son identité i. Les tâches concurrentes sont décrites ci-dessous.

1. Lors de la réception par j du message $ALIVE(path)$, il vérifie la structure $path$ reçue. S'il est en tête, cela veut dire que c'est lui l'initiateur de ce message et qu'il a été transmis en cycle. Ainsi, tous les nœuds qui le suivent dans $path$ sont mutuellement accessibles. De ce fait, il rajoute tous les nœuds apparaissant après j dans $inPART$. Par contre, si j n'est pas premier dans $paths$ (il est au milieu ou n'apparaît pas du tout), j se rajoute à la fin de $path$ et diffuse le message $ALIVE(path)$ à tous ses voisins. Le message est bloqué par j s'il apparait plus d'une fois dans $path$. C'est à dire le message est passé plus de deux fois par j.
2. Si le nouvel ensemble de nœuds $inPART_j$ (ensemble de nœuds que j croit appartenir à sa partition) est différente de la précédente, il incrémente la valeur du timeout. Cela signifie que, si j est un nœud *stable*, soit le ST_j n'est pas encore atteint ou bien qu'il a été atteint mais la valeur de timeout n'est pas suffisante pour que le message $ALIVE(path)$ envoyé par j puisse voyager à travers le plus long cycle de j. Lorsque les deux conditions se produisent (temps de stabilité et délai nécessaire pour passer par le plus

grand cycle), l'ensemble des processus dans $inPart_j$ et celui dans *output* sont toujours identiques. A la fin, le détecteur de j initialise son timer et la variable $inPart_j$ et diffuse le message $ALIVE(j)$, comme dans la phase d'initialisation.

Discussion

Avantages : — L'utilisation de la liste *path* permet à un nœud p de connaitre l'ensemble des nœuds qui sont dans sa partition.
— Minimisation de l'effet du timeout sur la décision.

Inconvénients : — Le protocole s'intéresse seulement à déterminer les participants d'une partition. Or Chaque nœud ne connait que les membres de sa partition. Donc ce protocole ne traite pas la communication entre les différentes partitions stables.
— Comme le nombre de nœuds est infini donc nous pouvons avoir un nombre important de messages. De plus dans ce cas le chemin traversé par path est long. L'incrémentation du timeout à chaque expiration peut atteindre des valeurs très grandes ce qui peut engendrer un retard de détection de défaillance par la suite.

4.4.8 Algorithme de Grève *et al.* (2012)

Les auteurs de [47] ont présenté un protocole utilisant la technique d'implémentation asynchrone de l'approche requête/réponse tolérant l'environnement mobile. Ce détecteur appartient à la classe $\Diamond S$ (complétude forte et précision faible).

Hypothèses

— Les nœuds connaissent au préalable uniquement leur identité.
— Le graphe sous-jacent est connexe.
— Les liens sont fiables.
— Environnement mobile.
— Le nombre maximal des nœuds qui peuvent tomber en panne, noté Nf,est supposé connu.

Principe de fonctionnement

L'algorithme utilise les structures suivantes :

— $KnownTo_i$: Ensemble des nœuds connus par i.

— $Susp_i$: Cette structure représente l'ensemble des nœuds suspects. Chaque élément de cet ensemble est un tuple (id, ct), où id est l'identité du nœud et ct est l'étiquette associée à cette information.

— $Mist_i$: Cet ensemble contient les identités des nœuds faussement suspectés par i.

— X_i : Il s'agit de l'ensemble des nœuds desquels i a reçu des réponses pour sa dernière requête.

— $Susp\_from_i$: Cette structure est un vecteur qui contient à l'indice j, l'ensemble $Susp_j$ envoyé par le nœud j dans sa réponse à la dernière requête du nœud i.

— $Mist\_from_i$: Il s'agit d'un vecteur d'ensembles. L'élément d'indice j contient l'ensemble $Mist_j$ envoyé par le nœud j dans sa réponse à la dernière requête du nœud i.

L'algorithme est constitué des tâches décrites ci-dessous.

1. Périodiquement (à chaque *tour*), i envoie un message $Query$ muni de ses listes $susp_i$ et $mist_i$ à tous ses voisins, ensuite il attend les $(d - Nf)$ réponses.
2. Lors de la réception d'un message $Query$ de j par le nœud i, ce dernier met à jour ses listes $susp_i$ et $mist_i$. Il ne prend en considération que les informations les plus récentes selon la valeur de l'étiquette ct. Ensuite, il envoie un message $Response(susp_i, mist_i)$ à j.
3. Lors de la réception d'un message $Response$ de j par le nœud i, ce dernier ajoute j à l'ensemble X_i, et il met l'ensemble $susp_j$ à l'indice j du vecteur $Susp\_from_i$ et l'ensemble $mist_j$ à l'indice j du vecteur $Mist\_from_i$.

Discussion

Avantages : — Ce détecteur de défaillances est implémenté sans avoir une connaissance préalable des membres de système ce qui est adéquat pour les réseaux ad hoc.

— Ce protocole n'utilise pas un timeout. La satisfaction de ses propriétés se base sur un modèle d'échange de message suivi par les nœuds.

Inconvénients : — La technique pinging (Requête/Réponse) génère trop de messages ce qui peut provoquer une saturation du réseau.

— Les informations sur les nœuds suspectés défaillants et sur les nœuds faussement suspectés sont propagées dans le système par l'envoi des listes *suspect* et *mistake* dans les messages requête et réponse. Ceci risque d'augmenter le trafic.

4.4.9 Algorithme de Lafuente *et al.* (2015)

Les auteurs du papier [66] proposent trois algorithmes de détection de pannes basés sur la technique de battements de cœur. L'objectif principal est d'optimiser l'utilisation des liens de communication unidirectionnels. En effet, ils estiment que le nombre de liens unidirectionnels utilisés est c, où c est le nombre de nœuds corrects dans le système.

Dans les trois algorithmes, les nœuds sont arrangés en anneau logique. un nœud i envoie un battement de cœur uniquement à son suivant dans l'anneau et arme un timer uniquement pour son successeur dans le but d'attendre son battement de cœur.

Principe de fonctionnement

Le premier algorithme est constitué de quatre tâches concurrente et implémente la classe $\diamond P$.

Chaque nœud i maintient une variable $balance_i$ qui mesure la différence entre le nombre de suspicions et le nombre de fausses suspicions corrigées (appelées réfutations dans l'algorithme) au niveau du nœud i.

Quand un nœud i suspecte son prédécesseur j, il diffuse un message de suspicion dans le réseau et incrémente $balance_i$ et une mise à jour de l'anneau est déclenchée. Par contre, si le nœud j n'est pas défaillant, il diffuse un message de réfutation. En recevant un tel message, le nœud i décrémente $balance_i$. Il est à signaler que la diffusion est supposée fiable à travers un algorithme proposée dans le même papier.

Ainsi, si $balance_i$ est nulle, alors aucun nœud n'est suspecté de défaillance.

Le deuxième protocole est constitué de cinq tâches concurrentes. Il implémente la classe $\diamond Q$. Il remplace chez chaque nœud i la variable $balance_i$ par une liste de suspects L_i. Le principe d'échange de battement de cœur se fait de la

même manière (selon l'anneau logique).

Lorsque i suspecte son prédécesseur j, il le rajoute à L_i et lui envoie un message de suspicion et met à jour son prédécesseur dans l'anneau. Vu que ce protocole ne diffuse pas le message de suspicion, le nouveau prédécesseur de i ne pourra pas savoir qu'il doit commencer à envoyer des battements de cœur à i. Ceci génère automatiquement la suspicion de ce nouveau prédécesseur par i et qui est corrigée par la suite. Ceci fait augmenter le nombre de fausses suspicions dans le système.

Si j est vivant et en même temps il reçoit un message de suspicion de la part de i, il rajoute tous les nœuds intermédiaire dans l'anneau logique (entre j et i) à sa liste de suspects (L_i). Il devient de ce fait le nouveau prédécesseur de i et il pourra lui signaler sa vivacité par la suite et ceci corrige la fausse suspicion. Le j doit sonder les nœuds qu'il vient de suspecter en leur envoyant un message *probe* pour savoir lequel parmi ces nœuds est défaillant. En recevant un tel message, un nœud k doit répondre par un battement de cœur à l'émetteur (j).

Le troisième algorithme composé de cinq tâches est une transformation du deuxième afin d'implémenter la classe $\diamond P$. Un ensemble G_i est géré par chaque nœud i qui l'embarque dans le message de battement de cœur. Toute mise à jour de L_i est aussi répercutée sur G_i. De plus, l'ensemble G_i est reconstruit à chaque réception d'un battement de cœur de la part du prédécesseur de i dans l'anneau logique. Ceci est dans le but d'éliminer les messages de sondage *probe* utilisés dans le deuxième algorithme.

Discussion

Avantages : — Les protocoles sont efficaces sur le plan communication. En effet, il minimise le nombre de liens unidirectionnels utilisés sauf lors de la propagation d'une suspicion.

Inconvénients : — Les protocoles se basent sur une structure logique qui est l'anneau. Ceci peut s'avérer coûteux surtout lors de la création et la maintenance de cette structure.

— Le délai est incrémenté lorsqu'un nœud répond hors délai. Il n'est jamais décrémenté ou remis à sa valeur initiale. Ceci peut affecter le délai nécessaire pour détecter une vraie défaillance.

— Le premier protocole utilise la diffusion (broadcast) pour alerter les autres d'une suspicion. Ceci peut être coûteux dans le contexte des

MANETs. De plus, toute fausse suspicion est propagée rapidement dans le réseau.

— Les protocoles ne gèrent pas les déconnexions.

— Le troisième protocole embarque l'ensemble G_i dans les messages de battement de cœur. Ceci risque de faire augmenter la quantité de données transférées, ce qui peut affecter la bande passante.

4.5 Conclusion

Nous avons présenté dans ce chapitre, les notions fondamentales de la détection de pannes : les différentes propriétés des classes de détecteurs de pannes et les trois techniques les plus utilisées pour la détection. Nous avons remarqué qu'une implémentation de type battement de cœur engendre moins de messages que la méthode requête. Cependant, la stratégie de type requête peut s'avérer intéressante afin d'éviter au maximum des fausses suspicions. Par la suite, nous avons présenté quelques protocoles de détection de pannes suivant leur environnement filaire ou mobile.

Dans le chapitre suivant, nous parlons d'une autre technique de tolérance aux pannes qui est le calcul de points de contrôle en vue du recouvrement après défaillance d'un nœud. Une telle approche nécessite l'implémentation de trois protocoles complémentaires : détection de pannes (qu'on vient de décrire dans ce chapitre) ainsi que les protocoles de checkpointing et de recouvrement arrière qui font l'objet du prochain chapitre.

Chapitre 5

Points de Contrôle et Recouvrement

5.1 Introduction

Le checkpointing (calcul des points de contrôle) basé recouvrement arrière est l'une des approches de tolérance aux pannes qui a été définie et appliquée pour garantir le bon déroulement d'une application distribuée en présence de pannes. Cette technique est utilisée dans les bases de données distribuées et les applications critiques. Elle est aussi utilisée afin de débugger les programmes distribués, le suivi et le contrôle et pour détecter certaines propriétés du système tel que la terminaison ou le blocage.

Un protocole de calcul de points de contrôle se charge de capturer l'état global du système. Par contre, un protocole de recouvrement se charge, après l'occurrence d'une panne, à faire revenir le système, par retour arrière (rollback), à un état global cohérent le plus récent possible.

Dans ce chapitre, nous nous intéressons à ce genre de protocoles. Nous donnons les définitions nécessaire puis nous présentons quelques protocoles de checkpointing que nous comparons.

5.2 Définitions préliminaires

5.2.1 Point de contrôle local

L'état local sauvegardé par le processus d'un nœud est appelé *pointdecontrôle* (appelé en anglais *checkpoints* or *snapshots*). Chaque état local consiste en un ensemble d'événements dont l'occurrence est survenue durant l'exécution de l'application distribuée. Nous distinguons trois types d'événements : événements internes, événements d'envoi de messages et événements de réception de messages.

Seuls certains états locaux sont définis en fonction de certains besoins comme étant des points de reprise qui sont habituellement sauvegardés en mémoire stable. Cette sauvegarde se fait selon le protocole de calcul de points de reprises.

Ainsi, l'ensemble des points de contrôle locaux n'est qu'un sous ensemble de l'ensemble des états locaux d'un processus [5].

5.2.2 Point de contrôle global

L'ensemble des état locaux (un par nœud) constiutuent un é*tat global*. En outre, un point de contrôle global est un état global composé uniquement de points de contrôle locaux [5] à partir duquel un calcul distribué peut redémarrer après un échec.

Parmi tous les états globaux, nous nous intéressons à ceux qui sont *cohérents* (en anglais *consistent global states*). Un état global est dit cohérent s'il ne contient pas de *messages orphelins*. Ce dernier dénote le cas où la réception d'un message est enregistré dans l'état local du nœud destinataire et l'événement d'émission n'existe dans aucun autre état local constituant le même état global.

Dans l'exemple que montre la figure 5.1, les états globaux C_1 et C_3 sont cohérents, alors que C_2 ne l'est pas à cause du message orphelin m_3.

5.2.3 Mémoire stable

Pour supporter les pannes, les points de reprise doivent être sauvegardés sur un support stable (*Stable storage*). Ce support doit être implémenté de telle façon qui permet aux données de rester accessibles même après une défaillance. Cela suppose une redondance des données à conserver.

Par ailleurs, lors de la création d'un nouveau point de contrôle le précèdent ne doit pas être invalidé qu'après la finalisation du nouveau. Ainsi, s'il y a échec

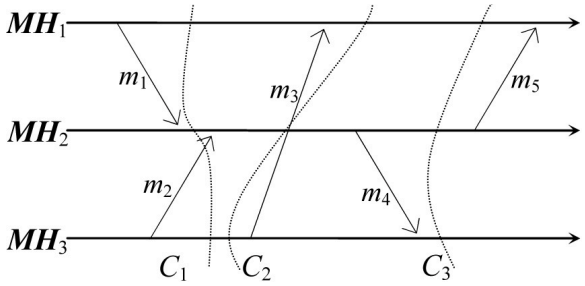

FIGURE 5.1 – Etats globaux cohérents et non cohérents

au cours de la sauvegarde d'un point de contrôle ; le précèdent est toujours disponible et considéré comme point de contrôle de référence. En fonction du nombre de fautes à tolérer, des performances et des moyens disponibles, plusieurs solutions sont possibles pour implanter une mémoire stable.

Le gestionnaire de fichiers doit être tolérant aux pannes pour assurer l'accès aux fichiers des checkpoints. Il doit assurer la continuité d'accès aussi bien en lecture qu'en écriture. Chaque point de contrôle pourra être dupliqué sur des disques séparés et probablement sur des sites différents. Cela nécessite une gestion des copies et des versions de backup (serveur de sauvegarde). Le nombre de copies identiques de points de reprise dépend de l'environnement (probabilité de crash) [6].

5.2.4 Effet domino

Lors de l'opération de recouvrement, il est préférable de considérer le dernier état global cohérent enregistré ou le plus récent possible. Ainsi, après l'occurrence de la panne, le calcul doit s'arrêter. Après la réparation ou le remplacement du nœud défaillant, tous les nœuds font un retour arrière vers leurs états locaux formant un état global cohérent. Donc, le système va redémarrer le calcul à partir de ce point. Par conséquent, ce dernier doit être le plus récent dans le but de minimiser la quantité de calcul perdu.

Par ailleurs, pour former des états cohérents, les processus peuvent coordonner leurs points de contrôle locaux, ou peuvent les considérer indépendamment les uns des autres. Dans ce dernier cas, il est possible que l'effet domino [67] survienne au cours de l'opération de recouvrement arrière. Ainsi, le recouvrement peut se faire de manière cascadée jusqu'à l'état initial (effet domino), ce qui oblige le système à reprendre le calcul dès les premiers points de contrôle

(point de contrôle initial) menant à la perte du travail accompli jusque là par tous les processus.

Pour éviter l'effet domino, les processus doivent coordonner leurs actions de façon à ce que chaque point de contrôle local soit associé avec un point de contrôle global cohérent qui lui appartient.

5.2.5 Ramasse miettes

Les points de contrôle sont stockés sur des supports de stockage (mémoires) stables. Après un certain temps, un sous ensemble d'informations ne devient plus utile au recouvrement arrière. Ceci nécessite de le supprimer. Mais cela morcelle la mémoire à force des opérations d'allocation et de dés-allocation. Si ce morcellement n'est pas maîtrisé, il peut conduire à l'impossibilité de trouver une zone de taille suffisante pour y allouer un nouveau point de contrôle. Un service nommé $ramasse - miettes$ (*garbage collector*) doit être mis en place. Le rôle d'un tel service consiste à libérer l'espace mémoire alloué devenu inutile.

Les ramasses miettes identifient la ligne de reprise dans le cas d'une défaillance, et suppriment toutes les informations ayant un rapport avec des événements qui se sont produits avant cette ligne. [32][68]

5.2.6 Protocole de calcul de points de contrôle

La plupart des protocoles de points de reprise définis dans le contexte de l'environnement mobile ont été conçus pour les réseaux cellulaires. Dans ces protocoles, la station de base (appelée en anglais mobile support station et notée MSS décharge les nœuds mobiles des principales tâches (synchronisation, certains calculs, mémoire stable, gestion des connexions et des déconnexions nœuds, ...). Cela facilite la gestion des contraintes de mobilité. Par conséquent, ces protocoles ne peuvent être appliquées dans le cadre de MANET, car aucune infrastructure filaire n'est disponible dans cet environnement. En effet, les nœuds mobiles peuvent être soumis à de fortes charges de calcul et de communication, qui peuvent affecter négativement les performances de l'ensemble du système.

Dans le cadre des MANETs, il y a de nouveaux défis qui doivent être relevés dans la conception de ces protocoles. Ci dessous les principaux à prendre en charge :

— Comment calculer un état global cohérent, et le plus récent, si possible ?

— Comment réduire le temps de latence et la surcharge du réseau en messages de contrôle ?
— Comment gérer les contraintes de l'environnement : la consommation d'énergie, la limitation de la bande passante, les déconnexions et reconnexions des nœuds... ?
— Comment mettre en œuvre la mémoire stable ? Il faut noter qu'une mémoire locale d'un nœud ne peut pas être considérée comme un espace de stockage stable à cause de sa vulnérabilité.

A notre connaissance, tous les protocoles de calcul de points de contrôle qui ont été définis dans la littérature pour les MANETs, ont été conçus pour fonctionner dans le contexte des réseaux hiérarchiques (avec clusters).

La clustérisation a été proposée comme solution pour gérer les réseaux à grande échelle. Cependant, le maintien d'une telle architecture est consommateur en temps et en énergie car elle nécessite la création de clusters et l'élection des clusterheads. Plusieurs protocoles de clustérisation ont été définis et diffèrent principalement sur la manière dont sont construits les clusters et comment est élu le clusterhead (appelé aussi le leader).

Presque tous les protocoles de clustérisation sont associés à un protocole de routage spécifique. Ainsi, un protocole de calcul de points de contrôle conçu pour les MANETs clusterisés (hiérarchiques) peut ne pas être applicable à de petits réseaux et de plus il est étroitement lié à la fois au protocole de clustérisation et au protocole de routage correspondant. Par conséquent, il devient difficile de déployer n'importe quelle application distribuée sur de tels réseaux. De plus, la plupart des protocoles de checkpointing qui sont définis supposent que le nombre de nœuds impliqués dans l'application distribuée doit être connu à l'avance par tous les nœuds, et ne doit pas varier au cours de l'exécution. Cependant, une telle hypothèse ne peut être satisfaite dans le contexte des MANETs plats. En effet, un nœud peut subir des déconnexions volontaires ou involontaires et de nouveaux nœuds peuvent rejoindre le réseau à tout moment.

TABLE 5.1 – Performances des classes de protocoles de checkpointing.

	Protocoles coordonnés	**Protocoles non-coordonnés**	**Protocoles induits communication**
Nombre de points de contrôle par processus	Un seul	Plusieurs	Plusieurs
Effet domino	Non	Possible	Non
Ramasse miettes	Simple	Complexe	Complexe
Recouvrement complexe ?	Non	Oui	Moyen
Retour arrière	Dernier checkpoint	Non borné	Un ou plusieurs checkpoint

5.3 Classification des protocoles de calcul de points de contrôle

Les protocoles de calcul de points de contrôle peuvent être classés en trois catégories principales :

Protocoles *non coordonnés*, *induits communication* et *coordonnés*.

Avant de détailler ci-dessous ces trois classes, nous donnons un tableau récapitulatif (voir table 5.1) qui dresse une comparaison de performances des ces trois classes de protocoles de checkpointing.

5.3.1 Classe de protocoles non coordonnés

Les protocoles de cette classe (en anglais *uncoordinated*), appelés aussi *indépendants* ou *asynchrones*, permettent aux différents nœuds de sauvegarder leurs états locaux indépendamment les uns des autres. Ainsi, chaque nœud décide de l'instant le plus approprié pour prendre un point de contrôle local.

Dans une telle technique, la surcharge en messages est sensiblement réduite vu qu'elle ne nécessite pas de coordination ou de synchronisation lors de la prise de points de contrôle. Cependant, plusieurs points de contrôle locaux doivent être pris et sauvegardés pour garantir l'existence d'un point de contrôle globale cohérent et récent. Durant le *recouvrement arrière* (*Rollback recovery*), l'état global cohérent (*ligne de recouvrement* ou *recovery line* [69]), est calculé en utilisant des estampilles temporelles basés sur les dépendances causales [70].

Le checkpointing non coordonné vise à maximiser les performances en réduisant la surcharge du réseau avec les messages de contrôle, mais peut ne pas garantir un point de contrôle global cohérent et récent pour le processus de recouvrement. Principalement, les inconvénients de l'utilisation de cette technique sont les suivants :

- Le risque de la survenue de l'$effet\ domino$ [67], qui consiste en une cascade de retours en arrière vers un ancien état, qui peut mener le système (dans le pire des cas) jusqu'à son état initial. Cela donne la perte de la majeure partie des calculs déjà effectués précédant la survenue de la panne.
- Un grand nombre de points de contrôle sont enregistrés dans la mémoire stable. Par conséquent, sa capacité doit être élevée.
- Certains points de contrôle ne sont pas utiles, car ils ne peuvent pas faire partie d'un quelconque état global cohérent [26]. Cela nécessite un $ramasse - miette$ (*garbage collector*) qui supprime tous ces points de contrôle inutiles.

Pour cette classe, plusieurs protocoles ont été proposés dans le contexte de réseaux filaires, comme par exemple, dans [71], [72] et [73], mais, à notre connaissance, aucun protocole n'a été proposé pour les MANETs. En effet, comme la mise en œuvre et le maintien d'une mémoire stable (SS) avec des capacités de stockage élevées est très difficile à réaliser dans le cadre des MANETs, il en résulte que ces protocoles ne sont pas appropriés pour cet environnement.

5.3.2 Classe de protocoles coordonnés

Dans les protocoles *coordonnée* ou *synchrones*, un nœud appelé l'*initiateur* ou le *coordinateur*, est censé coordonner la tâche de calcul des points de contrôle. Par conséquent, les nœuds sont invités à prendre les points de contrôle locaux uniquement lorsque cela est nécessaire. Ceci garantit le calcul d'un point de contrôle global cohérent et de plus assez récent. Par ailleurs, il simplifie ainsi le processus de *recouvrement*. En outre, il évite l'effet domino et réduit l'utilisation de la *mémoire stable* (SS). Cependant, cette technique induit un temps de latence plus élevé et une surcharge du réseau en message de contrôle. La latence est due au temps écoulé lors de la coordination qui génère des coûts en temps de calcul supplémentaires. En outre, des messages supplémentaires sont nécessaires pour gérer le processus de coordination.

Plusieurs protocoles au sein de cette classe ont été proposés dans la littérature. Certains protocoles synchrones proposent de bloquer toutes les communications liées au calcul distribué durant le processus de coordination [28]. Chandy et Lamport dans [30] ont proposé un protocole qui ne bloque pas le calcul sous-jacent.

Dans le contexte des réseaux cellulaires, des travaux similaires on été proposés dans [74], [67] and [75]. Dans le contexte des MANETs, nous pouvons citer les travaux de Gupta *et al* [76] et Tuli et Kumar [77].

Ces travaux n'ont pas discuté la mise en œuvre de la mémoire stable, malgré le fait que cette question est importante dans le contexte des MANETs.

5.3.3 Classe de protocoles induits communication

Les protocoles de cette classe (appelée en anglais *Communication-induced protocols*) sont aussi appelés protocoles *quasi-synchrones* (en anglais *quasi-synchronous* protocols). Elle a été définie principalement dans la perspective de réduire l'occurrence de l'effet domino et la surcharge en messages de contrôle.

A cet effet et afin de maintenir son autonomie, chaque nœud prend un *point de contrôle indépendant* (*independent checkpoint*), appelé *basique* ou *spontané* (en anglais *spontaneous* ou *basic*). En outre, dans chaque message de calcul transmis, des informations de contrôle supplémentaires sont embarquées, comme par exemple, l'estampille temporelle [70] du message et/ou le numéro de séquence du dernier point de contrôle pris. En plus des points de contrôle réguliers (spontanés), des *points de contrôle forcés* (*forced checkpoints*) [78] peuvent être pris, suite à la réception d'informations de contrôle récentes de la part d'un autre nœud. Un tel fonctionnement permet de réduire l'utilisation des messages de contrôle qui se traduit par une faible surcharge en messages. En outre, le processus de recouvrement est simplifiée et de plus la survenue de l'effet domino devient rare.

Toutefois, une telle technique nécessite d'enregistrer un grand nombre de points de contrôle. En outre, l'embarquement des informations de contrôle dans les messages de calcul peut induire une certaine latence sur les applications distribuées qui pourraient avoir une incidence dans certains cas sur ses performances.

Les protocoles de cette classe tentent de réaliser un compromis entre les avantages des protocoles de classes précédentes. Principalement, ils essayent de réduire le nombre de points de contrôle inutiles tout en assurant un état global

cohérent récent, évitant ainsi l'effet domino lors du recouvrement par retour arrière.

Les protocoles *induit-communication* peuvent être classifiés en deux types : protocoles de checkpointing *Basés modèle* (*Model-based*) et *basés index* (*Index-based*).

- Les protocoles de calcul de points de contrôle ***basés modèle*** utilisent un modèle de schéma (motif) de communication dans le but de déterminer s'il est possible de prendre un point de contrôle ou non [15][26]. Tous les points de contrôle pris sont utiles. Par contre, leur nombre peut être très important.
- Les protocoles de calcul de points de contrôle ***basés index*** maintiennent l'information nécessaire et requise pour vérifier la cohérence. Pour garantir au vol (on-the-fly) la cohérence [79][31] entre les différents points de contrôle, les protocoles de cette sous-classe embarquent l'index de chaque point de contrôle dans les messages de calcul. Ainsi, l'ensemble des points de contrôle (un par nœud) ayant le même index définit un point de contrôle global cohérent [68].

Dans [26] les auteurs, ont montré que les protocoles de checkpointing basés index génèrent moins de points de contrôle forcés comparativement aux protocoles de checkpointing basés modèle.

Plusieurs protocoles ont été conçus pour cette classe. Dans l'environnement mobile cellulaire, nous citons les travaux décrits dans [80], [78] and [81].

Pour les MANETs, nous pouvons citer les travaux de Ono et Higaki [82].

5.4 Travaux antérieurs

Plusieurs protocoles de calcul de points de contrôle et de recouvrement ont été proposés dans la littérature. Depuis le premier protocole proposé par Chandy et Lamport [30], plusieurs travaux ont été publiés dans les réseaux conventionnels (filaires), cellulaires ou ad hoc. Dans le contexte des MANETs, très peu de travaux ont été proposés dans la littérature pour aborder les défis du calcul de points de contrôle. Dans ce qui suit, nous discutons les principaux protocoles proposés pour l'environnement mobile (cellulaire ou MANETs).

5.4.1 Travaux de Acharya et Badrinath (1994)

Le premier protocole de checkpointing destiné à un système distribué avec des unités mobiles (environnement cellulaire) a été proposé dans [80]. Il est de type non coordonné, c'est à dire aucun processus n'est chargé de coordonner la prise de points de contrôle (Pas de processus initiateur). Ce protocole est aussi appelé « protocole à deux phases ». Une variable $Phase_i$ est utilisée au niveau de chaque MH (mobile host) pouvant prendre deux valeurs $SEND$ ou bien $RECV$. Initialement cette variable est à $RECV$.

Principe de fonctionnement

Cet algorithme suit la règle des deux phases, qui est la suivante :

— Si $Phase_i = RECV$ et que le nœud i veut envoyer un message m au nœud j alors la priorité est donnée à l'envoi de m ensuite $Phase_i$ prend la valeur $SEND$.

— Si $Phase_i = SEND$ et i reçoit un message m alors i prend un point de contrôle local, remet $Phase_i$ à $RECV$ et cela avant de délivrer le message m à l'application.

Un point de contrôle local est pris par un nœud à chaque réception d'un message uniquement s'il y a eu envoi d'un message après la prise du dernier checkpoint. Ce point de contrôle local est ensuite transféré vers la station de base (notée MSS) locale où il est enregistré. Lors de l'envoi d'un massage d'application m de i vers j, le nœud i embarque dans m deux vecteurs $CKPT_i[]$ et $LOC_i[]$.

— $CKPT_i[j] = p$: signifie que i voit que j a pris son $p$ème checkpoint.

— $LOC_i[j] = k$: signifie que i voit que j est à la MSS_k.

Chaque message m envoyé est enregistré dans une structure de donnée Log. Si Log contient tous les messages envoyés alors sa taille va très vite devenir très imposante. Afin de réduire sa taille, il est possible d'effacer tous les messages qui ne sont pas considérés comme "En Transit" (In-Transit) i.e. tous les messages envoyés par i et pas encore reçus par j, et j ayant déjà pris un checkpoint.

Pour la prise d'un point de contrôle global, la MSS envoie à chaque nœud j du système le message : $Request(H_j, CKPT_i[j])$ à la MSS_k (si $LOC_i[j] = k$) demandant son point de contrôle dont le numéro séquentiel est $CKPT_i[j]$.

Si le point de contrôle demandé est disponible au niveau de la MSS_k alors

cette dernière l'envoie à la MSS initiatrice. Dans le cas contraire, elle devra attendre jusqu'à ce que j prenne son checkpoint de numéro $CKPT_i[j]$ puis le lui envoie. Un point de contrôle global cohérent consiste en la collecte des points de contrôle locaux de tous les nœuds qui ont le même numéro de séquence.

Discussion

Avantages : — Economie d'énergie (Pas de messages de contrôle).
— Les nœuds en mode veille ne sont pas dérangées lors de la procédure de collecte de points de contrôle.

Inconvénients : — Surcoût dû au checkpointing à cause des vecteurs embarqués à chaque fois avec les messages de l'application.
— Très grand nombre de points de contrôle.
— Coût du recouvrement élevé.
— Latence de la collecte de checkpoint.
— Fait intervenir tous les processus dans la procédure de checkpointing global.
— Il est non scalable, car il utilise des vecteurs.

5.4.2 Travaux de Manivanan et Singhal (1996)

Le checkpointing quasi-synchrone proposé dans [78] est l'une des nombreuses techniques conçues pour les systèmes distribués mobiles en environnement cellulaire. L'algorithme proposé a comme avantages sa simplicité, le coût bas des checkpointing asynchrones, et le temps de recouvrement réduit. Le protocole quasi-synchrone préserve l'autonomie des processus, en leur permettant de prendre les points de contrôle de façon asynchrone.

Principe de fonctionnement

La synchronisation dans cet algorithme est réalisée grâce à la technique induit communication pour faire progresser de la ligne de recouvrement. D'autres points de contrôle sont pris à la réception de messages. Un index est utilisé et est incrémenté après chaque point de contrôle local ou forcé. Les points de contrôle forcés sont pris lors de la réception d'un message contenant un index supérieur au sien et cela dans le but de faire avancer la ligne de recouvrement. De là, l'idée de sauvegarder les points de contrôle dans les MSSs est apparue. Dans ce cas

la MH doit être capable de localiser la MSS qui stocke les points de contrôle requis durant le recouvrement. Un mécanisme pour résoudre ce problème est proposé dans ce protocole. Ce mécanisme consiste en une structure de donnée $location-info(pid, checknum, location)$ où pid est l'identificateur du processus, $checksum$ est l'index du point de contrôle et $location$ est l'identificateur de la MSS où est sauvegardé chaque point de contrôle de numéro $checknum$. Le nœud enregistre $location-info$ dans la MSS à chaque prise de points de contrôle valable pour la reprise après la panne. Le protocole assure qu'il y a une ligne de recouvrement avec chaque point de contrôle de chaque processus à tout instant.

Discussion

Avantages : — Absence de messages de contrôle (surcoût bas)
- — Recouvrement simple et évite l'effet domino
- — Réduit le nombre de points de contrôle inutiles lors du recouvrement arrière.
- — Economie d'énergie car les nœuds ne sont pas dérangés en mode veille par les messages de contrôle.
- — L'augmentation du nombre des processus n'influe pas sur les performances du protocole. On dit qu'il est scalable.

Inconvénients : — Enregistrement d'un grand nombre des points de contrôle.
- — Le transfert de points de contrôle d'un nœud vers son MSS, ce qui augmente le trafic sur le réseau sans fil.
- — Le protocole ne prend pas en considération les cas de la déconnexion et handoff (changement de cellule), ce qui retarde le temps de réponse du checkpointing.

5.4.3 Travaux de Quaglia *et al.* (1998)

Dans [81], les auteurs ont proposés une optimisation du travail publié dans [79] pour qu'il soit appliqué aux systèmes distribués en environnement mobile cellulaire.

Principe de fonctionnement

Il consiste à réduire la différence entre les numéros de séquence des points de contrôle notée sn_i sur des nœuds différents. Cela réduit considérablement

le nombre de points de contrôle forcés. Cette technique est une technique non coordonnée basée index, qui permet la prise de points de contrôle spontanés et d'autres forcés.

Un point de contrôle spontané est pris lorsqu'un nœud i change de cellule (Handoff) ou bien lors de l'opération de déconnexion, ce qui fait que le numéro de séquence sn_i s'incrémente.

Dans ce protocole, une nouvelle variable est introduite n_i qui représente le numéro de séquence maximum reçu par i dans un message d'application. Si lors de la prise d'un point de contrôle spontané C, nous nous retrouvons dans le cas où $rn_i < sn_i$, alors C ne dépend d'aucun point de contrôle de la ligne de recouvrement dont le numéro de séquence est sn_i. Le numéro de séquence de C est alors mis à sn_i puis C remplace son prédécesseur dans la ligne de recouvrement dont le numéro de séquence est sn_i. Un point de contrôle forcé est pris si lors de la réception d'un message m, le sn embarqué dans m (noté $m.sn$) est supérieur au sn_i local de i.

Discussion

Avantages : — Moins de points de contrôle pris que lors de la méthode précédente grâce à l'incrémentation lente des numéros de séquences.
— Economie d'énergie car il n'utilise pas de messages de contrôle.
— Pas de contrainte sur le nombre de processus participants.

Inconvénients : — Surcoût de checkpointing due à la prise de points de contrôles à chaque déconnexion et à chaque changement de cellule donc il y a un grand nombre de points de contrôle sauvegardés.
— Tous les processus doivent participer à la prise de checkpoint.
— Recouvrement complexe car c'est un protocole induit communication.
— Risque de l'effet domino.
— Latence de la collecte de checkpoint pour cause de prise de points de contrôle à chaque handoff et à chaque déconnexion.

5.4.4 Travaux de Prakesh et Singhal (1996)

Les auteurs ont proposé dans [74] un algorithme de checkpointing pour l'environnement cellulaire ayant comme objectif de faire participer le minimum de processus pour le calcul des points de contrôle globaux sans arrêter le calcul

distribué.

Principe de fonctionnement

Ce protocole utilise des messages de contrôle qu'il embarque sur les messages de calcul. Pour cela, il doit faire face à la faiblesse de la bande passante et à la source d'énergie limitée des nœuds, en diminuant le nombre de messages de contrôle. Donc, le protocole doit forcer le minimum de processus à prendre des points de contrôle locaux ce qui diminue le nombre de messages échangés, coûteux en énergie.

La déconnexion et le handoff (changement de cellule) sont pris en charge par ce protocole. Lorsqu'un nœud i veut se déconnecter d'une station MSS_k, il doit tout d'abord enregistrer son état local puis l'envoyer à la MSS_k, qui garde une copie à son niveau pour le calcul du point de contrôle global, et enfin se déconnecter de la station fixe.

Quand i se reconnecte à une autre station de base fixe MSS_j, il envoie une requête à son ancienne station MSS_k à travers la nouvelle. La station statique MSS_k envoie à son tour les informations dont à besoin le processus i, et dans le cas où, lors de la déconnexion de i, il y avait eu collecte d'un point de contrôle global, la MSS_k enverra aussi l'état et les variables.

Pour ne pas bloquer le calcul, quand un processus i envoie un message, il embarque la valeur courante de $csn_i[i]$. Quand un processus j reçoit le message m de i, j traite le message directement si et seulement si $m.csn < csn_j[i]$, sinon j prend un point de contrôle et met à jour sa valeur du csn ($csn_j[i] = m.csn$), puis traite le message. Cette méthode peut aboutir à un grand nombre de points de contrôle. Plus encore, elle peut mener à un effet d'avalanche qui tourne récursivement dans le système demandant aux autres processus de prendre des points de contrôle.

Le problème qui se pose est : comment forcer seulement une partie des processus à prendre un point de contrôle, car autrement le csn de quelques uns des processus peut être dépassé et ne peuvent pas éviter l'incohérence. Dans ce protocole, une solution a été proposée qui consiste à ce que chaque processus maintienne un tableau pour sauvegarder les numéros de séquence csn, où $csn_i[j]$ représente le csn de j attendu par i. En utilisant csn et l'identification du numéro par l'initiateur, ils prétendent que leur algorithme est non bloquant et peut prévenir l'incohérence et minimiser le nombre de points de contrôle durant

le checkpointing. Cependant, il a été prouvé dans [83] que cet algorithme peut mener à une incohérence et qu'il n'existait pas d'algorithme non bloquant qui force seulement le minimum de processus à prendre un point de contrôle.

Discussion

Avantages : — Minimum de processus impliqués dans le calcul de point de contrôle global.
— Sauvegarde de deux points de contrôle dans la mémoire stable (pas de ramasse miettes).
— Recouvrement simple (pas d'effet domino) car ce protocole est coordonné.
— Collecte rapide du point de contrôle global car il y a une coordination entre les processus.

Inconvénients : — Consommation d'énergie à cause des messages de coordinations.
— Surcoût de recherche des nœuds par les MSSs, car les MSSs et les nœuds sont traités de la même manière.
— Transfert d'une quantité d'information très importante.
— Ce protocole n'est pas scalable (le protocole devient très lent si le nombre des processus augmente, à cause de la taille $m + n$ (où m est le nombre de MSS et n le nombre de nœuds mobiles) embarquée dans les messages de calcul).
— Il a été prouvé dans [83] que le protocole peut mener à une situation d'incohérence.

5.4.5 Travaux de Cao et Singhal (2001)

Dans [67] et suite aux résultats obtenus dans [83], les auteurs ont proposé une amélioration au protocole de Prakash et Singhal [74]. Le protocole proposé est destiné aux systèmes distribués en environnement cellulaire. L'algorithme ne bloque pas le calcul sous-jacent et ne force pas tous les processus à prendre un point de contrôle.

Principe de fonctionnement

L'idée du protocole consiste à exploiter les messages de calculs afin de réaliser un checkpointing coordonné, pour transférer certaines informations liées à la prise de points de contrôle.

Dans ce protocole, il existe trois types de points de contrôle :

— Les points de contrôle permanents : Ils sont pris à la suite de la coordination.

— Les points de contrôle tentatives : Ils sont pris à la réception de message de calcul.

— Les points de contrôle mutables : Ils ne sont ni tentatives, ni permanents, mais peuvent être changé en checkpoints tentatives. Ils sont pris après réception d'un message de calcul où une variable embarquée dans ce dernier l'informe qu'une requête de checkpointing va bientôt lui parvenir.

Quand un processus prend un point de contrôle mutable, il n'envoie pas de requête aux autres processus pour leur demander de prendre des points de contrôle et il n'a pas besoin d'enregistrer le point de contrôle dans la mémoire stable. Il peut le sauvegarder n'importe où, dans la mémoire principale ou le disque dur du nœud par exemple. Ceci permet d'éviter les dépassements dus aux transferts de grande quantité d'informations vers un support de stockage stable sur une MSS à travers le réseau de communication sans fil.

Supposons qu'un processus i a pris un point de contrôle mutable. Quand i reçoit la requête de checkpointing, il transfère le point de contrôle mutable dans la mémoire stable et force tous les processus dépendant à prendre un point de contrôle tentative. Dans ce cas, i change son point de contrôle mutable en checkpoint tentative.

Si i n'a pas reçu de requête de checkpointing après la fin de l'opération de prise de points de contrôle, il se débarrasse de son checkpoint mutable. Le traitement de la mobilité se fait de la même façon que dans [74] et le calcul des points de reprise est réalisé sans faire de distinction entre les processus mobiles et les processus statiques. L'algorithme considère que les canaux de communication sont FIFO.

Discussion

Avantages : — Nombre de checkpoints sauvegardés sur le support de stockage stable est minimal (Deux checkpoints : un permanent et un autre tentative)
— Enregistrement uniquement des points de contrôle utiles garce aux points de contrôle mutables
— Recouvrement simple car le protocole est coordonné.

Inconvénients : — Surcoût de checkpointing dans ce protocole : les nœuds mobiles et les MSSs sont traitées de la même façon ce qui engendre un surcoût de recherche des nœuds.
— Transfert d'une quantité importante d'information (car il embarque des tableaux structure de donnée de taille $m + n$ où m est le nombre de MSS et n est le nombre de MH). Ceci rend le protocole non scalable. Ainsi, plus le nombre de stations augmente plus l'algorithme devient lent.
— Consommation d'énergie (les MHs peuvent être dérangées lorsqu'elles sont en mode veille en recevant un message de contrôle).
—

5.4.6 Travaux de Benkaouha (2003)

Nous avons proposé dans [6] un protocole de checkpointing destiné aux systèmes répartis en environnement mobile cellulaire.

Principe de fonctionnement

Cet algorithme est agencé niveau (tow-tier structured), c'est-à-dire qu'il traite les nœuds et les MSSs différemment et chacune a son propre protocole. Ce qui permettra de décharger les nœuds d'une grande partie de traitement qui est relégué aux MSSs. On peut classer cet algorithme comme étant un algorithme hybride puisque la première phase est non coordonnée car l'enregistrement des points de contrôle se fait localement et la seconde phase est coordonnée et sauvegarde les points de contrôle sur un support de stockage (mémoire) stable. Les deux phases sont décrites ci-dessous.

1. Phase normale : Cette phase consiste à ce que chaque processus prenne à sa propre initiative des points de contrôle spontanés comme dans le cas du

checkpointing indépendant. Cela permettra une simplicité du protocole et une certaine liberté pour les nœuds. Durant cette phase, pour assurer la cohérence, certains points de contrôle forcés sont pris, suite à la réception de certaines informations de contrôle transmises via les messages de calcul. Cette synchronisation est obligatoire. Pour éviter de transférer à chaque fois une grande quantité d'information vers le support de stockage stable se trouvant sur la MSS, il est proposé que les points de contrôle pris lors de cette phase soient sauvegardés localement (c.à.d. dans la mémoire ou le disque dure de l'unité mobile). Ces points de contrôle sont semblables au checkpoints mutables [67], et vont permettre d'éviter de gaspiller de la bande passante du réseau sans fil. La combinaison du checkpointing indépendant et de checkpoint induit communication nous rappellent l'approche des protocoles quasi-synchrones [78][84]. Ainsi, à chaque point de contrôle pris localement est assigné un numéro séquentiel à travers la variable locale *csn*. La variable *csn* est incrémentée à la prise d'un point de contrôle spontané si le processus a reçu un message depuis le dernier checkpoint pris. Un processus, en recevant un message de calcul comportant une valeur de *csn* supérieure à la sienne, il prend un checkpoint forcé et mettra à jour sa variable *csn*. Cela nécessite de sauvegarder plusieurs points de contrôle localement et pour optimiser l'espace de stockage, la technique de checkpointing incrémental a été utilisée. C'est-à-dire entre deux checkpoints on n'enregistre que les changements intervenus.

2. Phase de coordination : Le principal avantage des techniques de checkpointing coordonné est le nombre de points de contrôle sauvegardés sur mémoire stable qui est au maximum 2 (tentative et permanent). Cela simplifie le protocole de recouvrement et évite l'effet domino. Cette phase consiste donc à synchroniser les différents processus pour récolter un état global cohérent dans un support de stockage stable à partir des points de contrôle enregistrés localement. Pour diminuer le nombre de messages de coordination vers les unités mobiles, cette phase va se dérouler en trois étapes : La première consiste à réaliser la coordination entre les $MSSs$. La seconde étape se déroule au niveau de chaque MSS. Pour éviter le surcoût dû à la recherche d'une station mobile, chaque MSS synchronise uniquement les nœuds qui sont dans sa cellule. La dernière étape consiste à ce que chaque MSS informe le processus coordinateur de la fin de la

sauvegarde des tentatives de points de contrôle sur un support de stockage stable et le coordinateur met fin au protocole par l'envoi d'un message de type *commit* pour rendre toute tentative de point de contrôle permanente. Cela est bénéfique, si la coordination se limite aux $MSSs$ seulement. De là, le processus coordinateur doit être sur une MSS. Si un nœud veut initier la coordination, il charge sa MSS locale du rôle de coordinateur. Une MSS coordinatrice diffuse la requête à toute les autres $MSSs$ à travers les canaux de communication filaires. Chaque MSS doit prendre son checkpoint local et doit informer les MHs. Pour éviter que les nœuds soient dérangés, un délai donné T est attendu avant de le transmettre aux processus mobiles dans les cellules de la MSS. Durant ce temps, certains nœuds peuvent recevoir un message de calcul et la MSS pourra en profiter pour transmettre la requête de checkpointing à travers ce message. Ce délai passé, il faut obliger les processus mobiles n'ayant pas reçus la requête à envoyer leur checkpoint.

Discussion

Avantages :
- Sauvegarde locale pour les points de contrôle ce qui diminue le transfert d'information à travers le réseau sans fil.
- Sauvegarder uniquement les points de contrôle nécessaires.
- Réduction du nombre de points de contrôle.
- Les nœuds ne sont pas dérangées lors de leur mise en veille (économiser l'énergie).
- L'utilisation d'un algorithme coordonné rend le recouvrement simple
- L'algorithme est scalable.

Inconvénients :
- L'utilisation de timer pouvant augmenter la latence lors de la collecte du point de contrôle global.

5.4.7 Travaux de Gupta *et al.* (2006)

Dans le contexte des MANETs, Gupta *et al* ont proposé dans [76] un algorithme de checkpointing synchrone (coordonné) non-bloquant. Il visent à travers ce travail à gérer de façon efficace les disponibilités des ressources telles que la bande passante, l'énergie et la mémoire.

Principe de fonctionnement

Chaque nœud maintient une variable booléenne c_i qui est initialisée à 0 et mise à 1 lorsque le nœud envoie son premier message après l'enregistrement du dernier point de contrôle. Après chaque prise de point de contrôle, cette variable est remise à 0. Le nœud initiateur lance la coordination en diffusant un message M_c à tous les autres nœuds du système. L'initiateur ou tout autre nœud ne prend de point de contrôle que si sa variable c_i est à 1. Par ailleurs, le numéro de séquence des points de contrôle est embarqué dans les messages de calcul. Un nœud prend un point de contrôle lors de la réception d'un message de calcul seulement si le numéro de séquence embarqué est supérieur au numéro de séquence local.

Discussion

Avantages : — L'algorithme est non bloquant.
— Réduction du nombre de points de contrôle pris.

Inconvénients : — L'algorithme utilise la diffusion (broadcast) afin d'envoyer les messages de contrôle.
— Aucune solution n'est proposée pour la mémoire stable.
— En outre, certains problèmes ne sont pas abordées comme les déconnexions, la forte mobilité des nœuds ... etc.

5.4.8 Travaux de Li et Shu (2006)

Dans [85], les auteurs proposent un algorithme de calcul de points de contrôle coordonné bloquant qui réduit le délai induit par la prise du checkpoint global pour les réseaux mobiles.

Principe de fonctionnement

Une technique d'embarquement d'information est utilisée pour pister et enregistrer les renseignements de dépendance entre les points de contrôle parmi les processus, pendant la transmission du message de calcul.

Durant le checkpointing, une méthode de calcul de points de contrôle simultanée est conçue. Elle utilise les informations pré-enregistrées sur la dépendance des points de reprise pour minimiser le temps de bloquage des processus et cela

en envoyant au processus dépendants la requête de checkpointing en une seule fois. Cela permet de minimiser le temps de localisation de l'arbre de dépendance.

Discussion

Avantages : — Il force un nombre minime de processus à prendre un point de contrôle
— Réduction de la latence associée à la propagation de la requête de coordination par rapport à d'autres algorithmes coordonnés.

Inconvénients : — Bloque le calcul sous-jacent.
— Surcoût dû au checkpointing à cause des informations de dépendances embarquées à chaque fois avec les messages de l'application.

5.4.9 Travaux de Ono et Higaki (2007)

Ono et Higaki [82] ont proposé un protocole qui vise à réduire la surcharge des communications pendant la tâche de calcul de points de contrôle. Les auteurs ont proposé un protocole de prise de points de contrôle pour les réseaux ad hoc « sans mémoire stable » et avec une large bande passante pour la communication.

Par ailleurs, ce protocole utilise la mémoire de chaque nœud voisin comme une partie de la mémoire stable, qui n'est pas le moyen le plus sûre de son implémentation dans le contexte des MANETs.

Principe de fonctionnement

Dans ce protocole, une requête de checkpointing est propagée par diffusion. Les informations sur l'état d'un nœud sont embarquées dans cette requête et sauvegardées chez les nœuds voisins. Un nœud qui a perdu le message est détecté et conservé par un autre nœud qui est sur son chemin de transmission. Ici, le surcoût induit par la communication pour la prise du point de contrôle global est réduit.

Discussion

Avantages : — Réduction du surcoût dû à la collecte du point de contrôle global.

Inconvénients : — Ne traite pas le cas des déconnexions et re-connexions volontaires.
— Embarque une grande quantité d'informations dans les messages de calcul.
— A besoin d'une large bande passante qu'il occupe trop souvent à cause de la diffusion de requête.
— N'utilise aucune mémoire stable.
— Toutefois, les demandes sont transmises par diffusion dans le réseau, ce qui peut être coûteux dans le contexte de l'environnement ad hoc.

5.4.10 Travaux de Tuli et Kumar (2011)

Dans [77], Tuli et Kumar ont proposé un protocole coordonné non-bloquant, dans lequel ils supposent que le réseau est hiérarchique. Le protocole de routage utilisé est le CBRP (Cluster Based Routing Protocol) qui est un protocole spécifique à base de clusters. Leur objectif est d'impliquer le minimum possible de nœuds lors de la coordination.

Principe de fonctionnement

Chaque nœud i maintient un vecteur DV_i de taille n, où n est le nombre de nœuds dans le cluster. Toutes les entrées du vecteur Dv_i sont mises à 0 à chaque prise d'un point de contrôle par le nœud i. Une entrée $Dv_i[j]$ est mise à 1, lorsque le nœud i reçoit le premier message de calcul de la part de j après la prise du dernier checkpoint.

Seul le chef de cluster est autorisé à lancer la coordination. Chaque point de contrôle est numéroté grâce au compteur csn_i. Le clusterhead i envoie une requête de prise de point de contrôle tentative à un nœud j, s'il y a une dépendance entre eux (c'est à dire $Dv_i[j] = 1$

Par ailleurs, le protocole exploite les messages de calcul pour embarquer son csn afin de synchroniser l'évolution de la collecte des points de contrôle au niveau des autres nœuds.

Discussion

Avantages : — Protocole non-bloquant impliquant le minimum de processus lors de la collecte du point de contrôle global.

Inconvénients : — Ce protocole est destiné uniquement aux réseaux hiérarchisés.
— Le protocole est dépendant du protocole de routage.
— Les auteurs ne proposent pas de solution pour implémenter la mémoire stable dans les MANETs. Le protocole ne gère pas les initiatives concurrentes.

5.4.11 Travaux de Jaggi et singh (2011)

Jaggi et Singh [86] ont proposé un algorithme pour enregistrer un état global en construisant un arbre recouvrant auto-stabilisant sur une topologie de réseau en cluster. L'objectif est de réduire le nombre de messages dans le système.

Principe de fonctionnement

Dans ce protocole, seuls les chefs de cluster (clusterheads) qui ont le droit d'initier la coordination. Les messages de coordination vont circuler à travers l'arbre de recouvrement. En effet chaque nœud envoie à son père et à son fils dans cet arbre. Le protocole prend en considération les initiatives de calcul point de contrôle concurrente et les traite.

Discussion

Avantages : — La possibilité d'avoir des initiatives concurrentes. Ceci peut faire converger la coordination plus rapidement.

Inconvénients : — Le coût de la clusterisation et sa maintenance peuvent avoir des conséquences négatives.
— La construction des arbres et leur maintenance peuvent altérer les performances du calcul distribué ainsi que les disponibilités de ressources.

5.4.12 Travaux de Singh et Jaggi (2013)

Dans [87], Les auteurs ont proposé un protocole de checkpointing asynchrone destiné aux réseaux ad hoc clusterisé multisauts.

Le papier présente un algorithme qui permet un recouvrement arrière asynchrone d'un nœud en environnement mobile après une panne utilisant le mouvement basé de points de contrôle et la journalisation des messages (message

logging). Les objectifs de leurs travaux sont principalement la résolution des problèmes rencontrés par le processus de recouvrement dans les MANETs dues aux contraintes suivantes :

— Mémoire stable inadéquate.
— Bande passante limitée.
— Topologie dynamique et par conséquent les partitions du réseau.

Principe de fonctionnement

Le protocole découpe le réseau en clusters disjoints. Il utilise un arbre de couverture pour mieux gérer la situation et réduire les messages de recouvrement. Le protocole de calcul de points de contrôle est asynchrone. Ainsi, la partie essentielles des traitements se fait lors du recouvrement.

Pour implémenter la mémoire stable, certains clusterhead sont chargés de jouer ce rôle et ils sont appelés CMC (checkpoint and movement Coordinator).

Discussion

Avantages : — Le protocole tente de trouver une solution pour la mémoire stable.

Inconvénients : — Recouvrement complexe.
— Plusieurs points de contrôle enregistrés.
— Le protocole est destiné uniquement aux MANETs hiérarchisés.
— La création et la maintenance de l'arbre de couverture peut s'avérer coûteuse.

5.5 Conclusion

Dans ce chapitre, nous avons exposé les différentes définitions de base couramment utilisées dans le domaine du checkpointing qui sert à assurer la tolérance aux pannes dans les systèmes distribués filaires (conventionnels).

Par la suite, nous avons vu la classification des différents protocoles de calcul de points de contrôle. Ceux-ci sont divisés en trois grandes classes à savoir les algorithmes non coordonnés, les algorithmes induit communication et les algorithmes coordonnés Les principaux objectifs à atteindre par toutes ces techniques sont la minimisation du temps de calcul en exécution sans faute, la réduction du

surcoût de stockage des checkpoints afin d'éviter d'engendrer l'effet domino lors du recouvrement.

Les principaux algorithmes ont été discutés en fin de chapitre. Nous avons conclu que les protocoles coordonnés étaient à éviter car ils utilisent beaucoup de messages de contrôle et cela n'est pas conseillé à cause d'une consommation de ressources importante, jumelée avec une augmentation de la latence du calcul distribué. Les protocoles non-coordonnés quant à eux ont besoin de prendre plusieurs points de contrôles sans coordination, cela pose le problème de l'effet domino qui peut survenir et augmente le coût de calcul de la ligne de recouvrement. Certaines techniques combinent les deux méthodes (coordonnée et non coordonné) pour éviter les problèmes engendrés.

A notre connaissance, peu d'algorithmes de checkpointing pour les réseaux ad hoc, ont été proposés. Partant de ce constat, nous proposons, dans les prochains chapitres (6 et 8), un nouveau protocole de calcul de points de contrôle pour les réseaux mobile ad hoc permettant de prendre en charge les contraintes de ces derniers.

Chapitre 6

Un Outil de Tolérance aux Pannes

6.1 Introduction

Notre travail entre dans le domaine de la tolérance aux pannes dans les systèmes distribués en environnement mobile ad hoc. Il s'agit de développer un outil de tolérance aux pannes pour assurer le bon fonctionnement des applications distribuées.

L'outil qui consiste en l'exécution de plusieurs protocoles en parallèle et en arrière plan par rapport à l'application distribuée sous-jacente. Les protocoles qui sont définis dans ce chapitre et les deux suivants permettent de définir un outil de tolérance aux pannes qui permet à une application distribuée en environnement mobile ad hoc de se dérouler, atteindre ses objectifs et se terminer correctement malgré la présence de la panne.

L'outil est constitué d'un protocole de détection de pannes, d'un protocole de calcul de points de contrôle, un protocole de recouvrement, la mise en place d'une mémoire stable ainsi que la synchronisation entre tous ces protocoles et l'application distribuée.

Dans ce chapitre, nous décrivons le fonctionnement global de notre outil de tolérance aux pannes qui constitue nos différentes contributions dans le cadre de cette thèse. Nous présentons l'environnement de travail, quelques hypothèses ainsi que quelques idées conceptuelles et la synchronisation entre les protocoles.

6.2 Contexte de travail

Le contexte traite les applications distribuées qui sont supposées s'exécuter sur au maximum N nœuds mobiles (notés MH pour Mobile Host). Il s'agit d'un système distribué en environnement mobile ad hoc (MANET). Il est à noter que nous ne prétendons pas de gérer la mobilité des nœuds mais juste de traiter et faire face aux contraintes spécifiques à cet environnement. Ces contraintes ont été expliquées en détail dans le chapitre 3.

Nous évitons dans notre solution d'imposer l'utilisation d'un quelconque protocole de routage ou structure logique (cluster, anneau, arbre, ...). Ainsi, notre protocole et ses performances ne risquent pas d'être affectés, par exemple, par l'organisation du réseau (plate ou hiérarchique) et/ou par le protocole de routage.

Pour être plus proche de la réalité, nous admettons que le nombre de nœuds actifs peut varier durant l'exécution de l'application. En effet, à tout moment, nous pouvons observer l'arrivée, le départ, la déconnexion, la re-connexion ou la panne d'un ou plusieurs nœuds.

Cependant, pour garantir une certaine tolérance aux pannes pour l'application distribuée, il faut s'assurer qu'elle s'exécute, qu'elle puisse se terminer et qu'elle atteigne les objectifs pour lesquels elle a été conçue malgré la présence de pannes. La panne peut affecter un ou plusieurs nœuds durant l'exécution.

Parmi les solutions les plus connues dans les systèmes distribués pour ce genre de situations est le *recouvrement arrière* (en anglais *rollback recovery*). Pour réaliser un retour en arrière, il faut développer un protocole de recouvrement qui lui-même nécessite un protocole de *calcul de points de contrôle* (*checkpointing*). En effet, le protocole de *recouvrement* se base sur les états enregistrés sur la *mémoire stable* par le protocole de *calcul de points de contrôle* pour pouvoir réaliser le *retour arrière*. A cet effet, une solution pour l'implémentation de la *mémoire stable* en environnement ad hoc doit être proposée.

De plus, pour lancer une opération de recouvrement arrière, il est nécessaire de détecter la panne. Ainsi, un protocole de *détection de pannes* qui fait le minimum de fausses suspicion est primordiale pour notre cas.

Dans les sections qui suivent nous expliquons l'interaction entre ces différents protocoles qui constituent notre outil de tolérance aux pannes.

6.3 Description de l'environnement

Nous décrivons dans cette section l'environnement de calcul. Nous nous situons dans le cas de l'exécution d'une application distribuée.

L'application distribuée s'exécute dans un système supposé asynchrone. En d'autres termes, il n'y a aucune hypothèse sur le temps physique, c'est à dire qu'il n'y a :

— ni horloge globale,
— ni mémoire physique partagée,
— ni hypothèse sur la vitesse relative des processus,
— ni de délais de transmission des messages.

En effet, nous pouvons observer cet asynchronisme à travers les points suivants :

— les nœuds communiquent uniquement par échange de messages. Il s'agit du seul et unique moyen pour synchroniser les nœuds dans le système.
— Chaque processus (localisé sur un nœud MH) s'exécute à sa propre vitesse.

Par ailleurs, il faut noter que dans le contexte des détecteurs de pannes non fiables, il est difficile, voire impossible de ne pas émettre d'hypothèses sur les délais de transmission des messages. Nous avons expliquer ce principe dans le chapitre 4. Pour cet effet, nous parlons plutôt d'un système *partiellement synchrone*.

Nous admettons aussi que le support de communication du système réparti est un réseau mobile ad hoc (MANET). Ainsi, les messages sont échangés à travers des canaux de communication sans fils. Un nœud i peut envoyer directement des messages (via le canal sans fil) à un nœud j, si et seulement si, la distance entre i et j est inférieure à la portée du signal du nœud i. Par conséquent, chaque nœud connaît tous ses voisins directs grâce aux échanges fait au niveau de la couche MAC (Messages $Hello$ par exemple).

Par contre, un nœud i peut envoyer des données à un nœud k, même s'il n'est pas à sa portée, et ce en exploitant le protocole de routage disponible sur le réseau ad hoc.

De plus, l'environnement peut être décrit par les points que nous citons ci-dessous :

— Le réseau est constitué de nœuds mobiles hétérogène (matériellement et

logiciellement).

— Chaque nœud possède un identificateur unique dans le réseau.
— Les protocoles ne doivent pas faire d'hypothèses sur le nombre de nœuds. Il n'y a aucune hypothèse sur la connaissance préalable du nombre et l'identité des nœuds dans le système.
— Comme nous l'avons précisé plus haut, les nœuds communiquent directement uniquement avec leurs voisins. En d'autres termes, leurs connaissances initiales se réduisent à leurs voisinages. Cette connaissance peut s'enrichir au fur et à mesure que l'application évolue. Ceci dit, et suite aux échanges de messages, tous les nœuds peuvent finir par se connaître.
— Nous considérons que chaque nœud du système est une unité mobile où s'exécute un processus du calcul distribué.
— A tout moment un nœud peut se déconnecter volontairement ou involontairement (batterie ou signal faible).
— Un nœud peut se reconnecter à tout moment, à n'importe quel endroit du réseau.

6.4 Hypothèses

Nous donnons les hypothèses communes entre les différents protocoles dans cette section. Nous essayons de minimiser les hypothèses pour se rapprocher des contraintes réelles d'un réseau mobile ad hoc.

— Les déconnexions et les reconnections se font par l'envoi de messages *disconnect* et *reconnect* aux voisins, qui permet de distinguer les nœuds déconnectés des nœuds défaillants.
— Les nœuds sont organisée en réseau ad hoc et sont mobiles. Leur nombre est variable mais au minimum nous avons trois nœuds dans le réseau ($N \geq 3$). Mais comme nous l'avons expliqué ci-dessus la valeur de N n'est pas définie au préalable.
— Nous supposons que s'il y a partitionnement dans le réseau, sa durée dans le temps est limitée.En effet, si le partitionnement dure dans le temps, on risque de faire de fausse suspicions de pannes qui ne seront pas corrigées rapidement, ce qui engendre un recouvrement inutile.
— Dans le cas du risque d'épuisement de la batterie d'un nœud ou l'affaiblissement de l'intensité du signal reçu, nous supposons que tous les nœuds

sont dotés de mécanisme de connectivité leur permettant de diffuser un message de déconnexion dans le voisinage. Ce message est exploité par le système distribué pour mieux manipuler ce genre de situations. Ce mécanisme permet aussi de gérer le re-connexion des nœuds.

— Le type de défaillance considéré est toujours la panne franche (crash). Toutes les données du nœud défaillant sont perdues définitivement.

— Nous supposons que tous les nœuds collaborent et délivrent des résultats corrects pour l'application distribuée et les protocoles formant l'outil de tolérance aux pannes. En d'autres termes, aucun nœud n'est compromis (point de vue sécurité) ou malicieux. Ceci sous-entend que le système est doté au préalable de mécanismes permettant de détecter et d'isoler tout comportement malicieux.

Les hypothèses citées ci-dessus sont communes aux différents protocoles qui sont détaillés dans les chapitres 7 et 8. Par ailleurs, certaines hypothèses particulières à certains protocoles sont émises dans les chapitres correspondants.

6.5 Fonctionnement global de l'outil

Après un large parcours des travaux similaires (voir chapitres 4 et 5) qui existent dans la littérature et à notre connaissance, ce genre de travaux n'ont jamais été abordés. En effet, nous pouvons trouver des détecteurs de pannes mais qui sont destinés pour résoudre d'autres problèmes tel que le consensus ou l'accord. Nous trouvons aussi des protocoles de calcul de points de contrôle qui ne proposent pas de synchronisation avec le détecteur de pannes et généralement ne donnent aucune solution pour l'implémentation de la mémoire stable.

Ainsi, nous visons à concevoir :

— Un protocole de détection de pannes qui doit faire le minimum de fausses suspicions pour éviter de faire des retours arrière inutiles.

— Le protocole de calcul de points de contrôle (checkpointing) doit générer un état global cohérent du système et qui soit le plus récent possible afin de perdre le minimum possible de calcul déjà réalisé en cas de défaillance.

— Ces deux protocoles doivent se synchroniser pour pouvoir lancer le protocole de recouvrement qui doit être le plus simple possible et assez rapide.

Tous ces protocoles doivent prendre en considération les contraintes des MANETs tel que : la limite des ressources, l'énergie, les déconnexion, le nombre inconnu

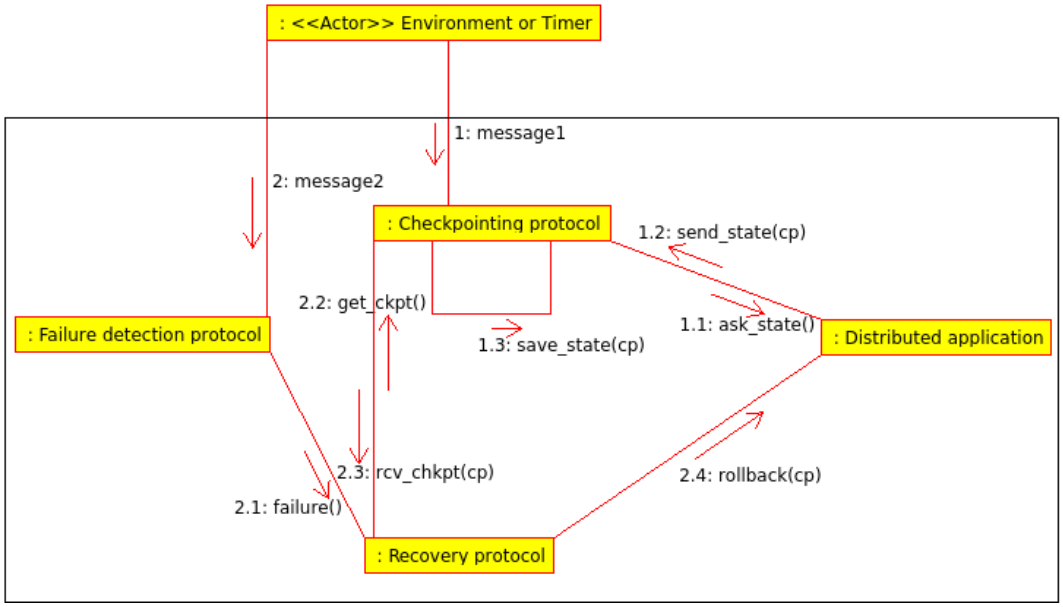

FIGURE 6.1 – Fonctionnement global du protocole de checkpointing et de recouvrement arrière.

des nœuds, ...

Toute défaillance d'un nœud doit être détectée. Cette détection doit mener vers l'arrêt de l'application distribuée et par la suite l'initialisation de l'opération de recouvrement (voir action 2.1 dans la figure 6.1).

Par conséquent, il est nécessaire voire primordial de doter l'application distribuée d'un mécanisme de détection de pannes. Ceci dans la perspective d'informer tous les autres nœuds (voir *message*2 de l'action 2 dans la figure 6.1) d'arrêter momentanément le processus de calcul. Ainsi, un *protocole de recouvrement arrière* est déclenché (voir actions 2.2, 2.3 et 2.4 dans la figure 6.1), dès que le nœud défaillant est remplacé ou réparé.

Pour mieux illustrer comment progresse les différents protocoles de tolérance aux pannes par rapport à l'application distribuée, un diagramme de séquences est donné dans la figure 6.2. On peut voir que le protocole de calcul de points de contrôle, comme le protocole de détection de pannes fonctionnent en parallèle avec l'application distribuée. Quand le protocole de détection de pannes détecte la défaillance d'un nœud, l'application distribuée et l'ensemble des protocoles arrêtent leurs exécutions. A ce niveau, le protocole de recouvrement intervient pour remplacer le nœud défaillant puis fait retourner l'état actuel de chaque nœud vers le plus récent état dans la mémoire stable (stable storage notée SS)

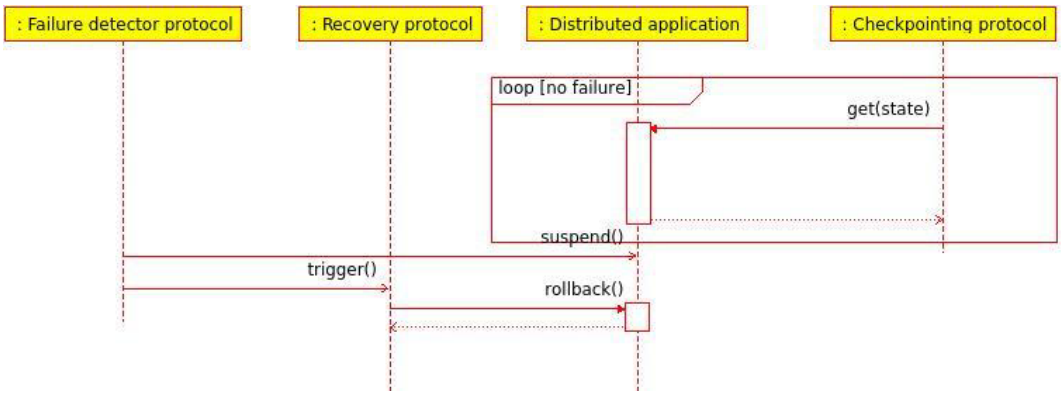

FIGURE 6.2 – Diagramme de séquences de l'outil de tolérance aux pannes.

formant un état global cohérent.

L'application distribuée peut, par la suite, continuer son exécution en parallèle avec les protocoles de calcul de points de contrôle et de détection de pannes.

6.6 Synchronisation des protocoles

Les protocoles de calcul de points de contrôle et de détection de pannes peuvent s'exécuter en arrière plan de l'application distribuée comme nous l'avons expliqué ci-dessus. En d'autres termes, nous avons trois protocoles qui s'exécutent en parallèle au niveau de chaque nœud qui participe au calcul distribué.

Quand le module de détection de panne d'un nœud i met un nœud j dans sa liste de suspects finale, il attend un délai d puis avertit les autres nœuds pour lancer le recouvrement. Le délai d peut être défini comme étant le délai maximal nécessaire pour corriger une fausse suspicion. Ce délai a été étudié dans la partie expérimentation (Voir résultats de simulations du chapitre 7).

Lors de la réception d'un tel message, les nœuds doivent arrêter leurs calculs sous-jacents pour lancer le recouvrement par retour arrière (rollback recovery) comme cela a été expliqué précédemment. En effet, le calcul que devait réaliser le nœud défaillant est attribué à un autre nœud qu'on appelle le nœud remplaçant (soit un nouveau nœud, soit un nœud qui existe déjà dans le système).

Le nœud remplaçant se charge de synchroniser les autres nœuds pour bien orchestrer l'opération de recouvrement. L'opération de recouvrement a pour but de minimiser la quantité de calcul à refaire. En d'autres termes, à minimiser le travail perdu et de ce fait minimiser le temps d'exécution de l'application

distribuée.

A la fin du recouvrement, les nœuds vont reprendre le calcul normalement ainsi que l'exécution des deux autres protocoles en arrière plan (détection de pannes et checkpointing).

6.7 Idée de base du protocole de détection de pannes

Notre objectif est de concevoir un protocole de détection de pannes en vue de lancer par la suite le protocole de recouvrement. Pour cela, nous aspirons à minimiser le nombre de fausses suspicions (Les nœuds corrects qui sont considérés par les autres comme étant défaillants). Les fausses suspicions peuvent causer le risque du lancement de retours arrière dans le calcul qui s'avèrent inutiles.

Nous avons opté pour la technique de battements de cœur (heartbeat). Nous utilisons deux niveaux de listes de suspects (une liste temporaire et une liste finale) et des niveaux différents de temporisateurs (timeout) pour fixer des délais d'attente ainsi que des messages particuliers de correction.

Nous nous sommes basés dans notre conception sur les idées ci-dessous illustrées par le diagramme d'états représenté dans la figure 6.3. :

— Nous nous basons principalement sur l'échange de messages d'informations dans le but de prouver la vivacité des entités constituant le système et afin de permettre de distinguer entre les entités correctes de celles qui ne le sont pas (c'est à dire celles qui sont défaillantes).

— La communication se fait uniquement entre nœuds voisins. La limitation de la communication aux voisins permet de réduire le nombre de messages transmis dans le réseau. Il permet aussi une meilleure gestion de la mobilité des nœuds. En outre, le protocole de routage est déchargé de ces opérations de transmissions. Ainsi, les performances de la détections des pannes devient indépendante du protocole de routage.

— Comme nous l'avons déjà précisé, la technique utilisée pour la détection de pannes est celle de *battements de cœur* (*heartbeat*). Cette technique se base sur l'envoi de messages de battement de cœur aux nœuds. Elle parait la plus convenable car elle engendre moins de messages que la technique requête. La diffusion des heartbeats est décrite dans la *région* 1 de la

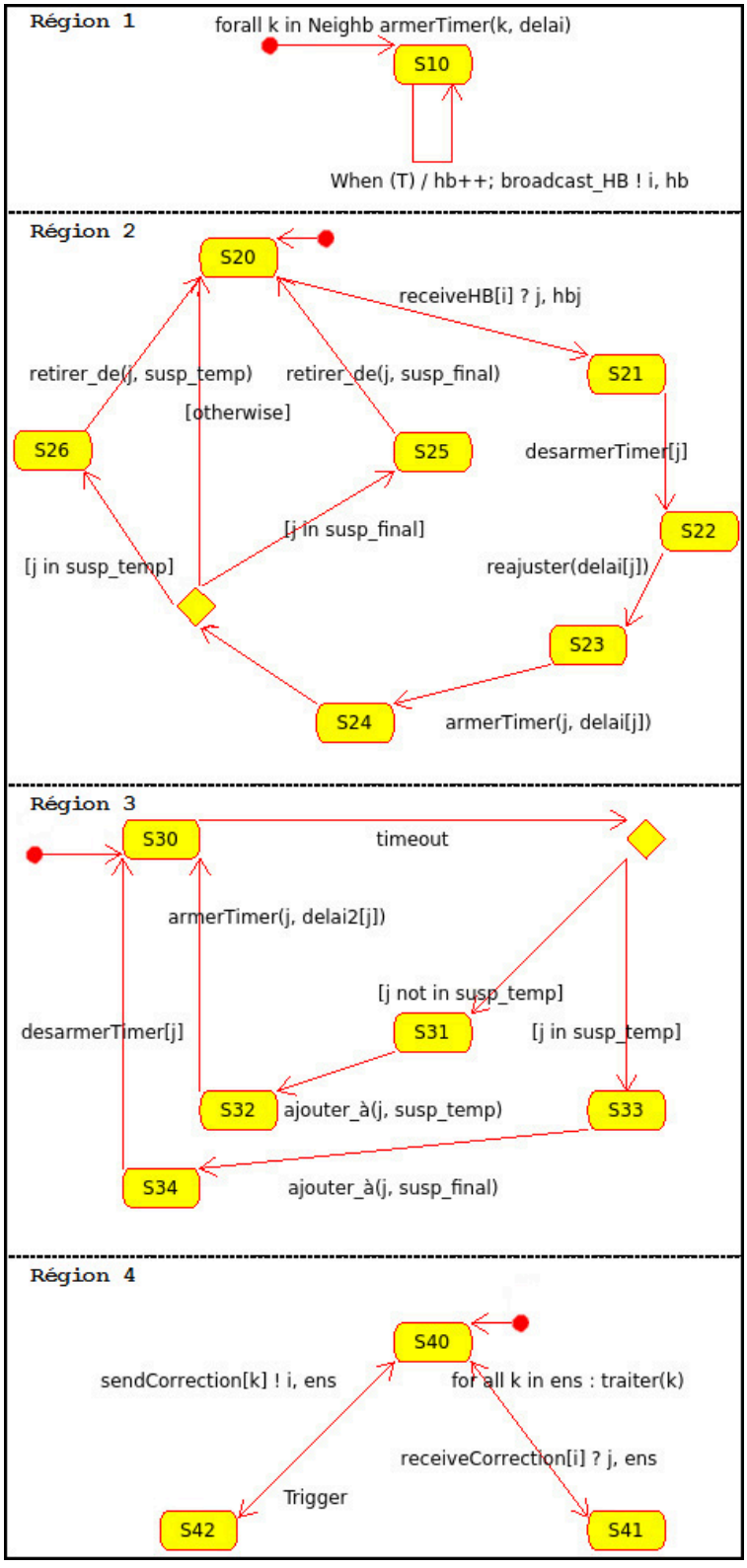

FIGURE 6.3 – Diagramme d'états de la détection des pannes.

figure 6.3.

— Nous mettons en œuvre deux types de messages : des messages de *vivacité* qui permettent à chaque nœud de montrer sa présence au prés de ses voisins et des messages de *correction* pour propager l'information sur la détection dans le réseau et surtout pour corriger les fausses suspicions. Ainsi, la réception de ses messages permet de mettre à jour toutes les listes utilisées. La *région* 2 de la figure 6.3 décrit la réaction d'un nœud i lors de la réception d'un battement de cœur de la part de j et comment se fait la mise à jour des listes. Par contre la *région* 4 de la même figure décrit à travers les transitions entre les états $S40$ et $S41$.

— Le message de vivacité est diffusé périodiquement (tous les β unités de temps) aux voisins. Dans les protocoles basés sur la technique de battements de cœur, la valeur de β est généralement supposée constante et la même pour tous les nœuds. Comme indiqué dans [65], il n'y a pas de moyen pour savoir comment la valeur choisie influe sur le protocole mais dépend de l'occurrence de la défaillance. Afin de minimiser le nombre de messages de contrôle, nous pouvons exploiter les messages $Hello$ pour jouer le rôle d'un heartbeat. Ainsi, la valeur de β peut être égale à la durée entre deux messages $Hello$ émis par un nœud. β est exprimé dans la figure 6.3 par l'argument *delai* de la fonction $armerTimer$ dans la *région* 1.

— L'envoi des messages de *correction* est déclenché (transition de l'état $S40$ à l'état $S42$ dans la figure 6.3) par le système selon la situation. Dans le chapitre 7, nous précisons en détail comment traite chaque protocole les messages de correction. La gestion des messages de correction est décrite à travers la *région* 4 de la figure 6.3).

— Contrairement aux autres protocoles qui existent dans la littérature, nous considérons que chaque nœud du système gère deux délais (timeouts) pour chacun de ses voisins. Ceci est dans le but d'éviter de pénaliser les nœuds dont les processus sont lents en les suspectant par erreur. Dans la *région* 3 de la figure 6.3, nous définissons deux niveaux de timeouts.

  Initialement le premier timer est armé (transition de $S23$ à $S24$). Quand le premier délai est atteint pour le nœud concerné et aucun message de *vivacité* n'a été reçu de sa part alors il est placé dans la première liste de nœuds suspects dite *liste temporelle*. L'expiration du premier timeout et la mise des nœuds suspectés dans la liste temporaire est représentée dans

le diagramme d'actiovité (figure 6.3) par la transition de l'état $S31$ vers l'état $S32$. La liste temporelle est créée pour donner une autre chance à ce nœud qui peut ne pas être réellement en panne. En effet, il peut être lent ou il s'est déplacé loin du voisinage. Par la suite, le second timeout est armé (transition de l'état $S32$ vers $S30$).

À l'expiration du deuxième timeout, ces nœuds sont mis dans la liste de suspects finale (transition de l'état $S33$ à $S34$). Par conséquent, un nœud est considéré comme défaillant et mis dans la seconde liste (appelée *liste finale*) de suspects une fois le délai relatif a expiré sans recevoir un message de vivacité.

Pour être plus explicite, étant donné un nœud i, tous les nœuds qui ont répondu à i avant l'expiration du premier délai, noté $TO1$ sont considéré par i comme étant vivant et rapides. Ceux qui ont répondu avant l'expiration du second délai, noté $TO2$, sont aussi considérés vivant mais lents.

Finalement, ceux qui n'ont pas répondu avant $TO1 + TO2$ sont suspectés. Plus de détails sur sur ces deux temporisateurs sont donnés dans la description détaillée ainsi que les algorithmes associés aux protocoles de détection de pannes dans le chapitre 7.

— Les différents temporisateurs nécessitent une gestion particulière des délais attribués à chaque nœud.

Le premier timeout est dynamique. En effet, il est incrémenté après la réception du message de vivacité de la part des nœuds lents (après le premier délai et avant le second) et décrémenté pour ceux qui sont rapides.

Le second délai (timeout) est supposé statique et la même valeur est maintenue pour les processus de tous les nœuds dans le système.

— Les *déconnexions* et les *reconnections* se font par l'envoi de messages particuliers aux voisins, qui permettent de distinguer les nœuds déconnectés des nœuds défaillants.

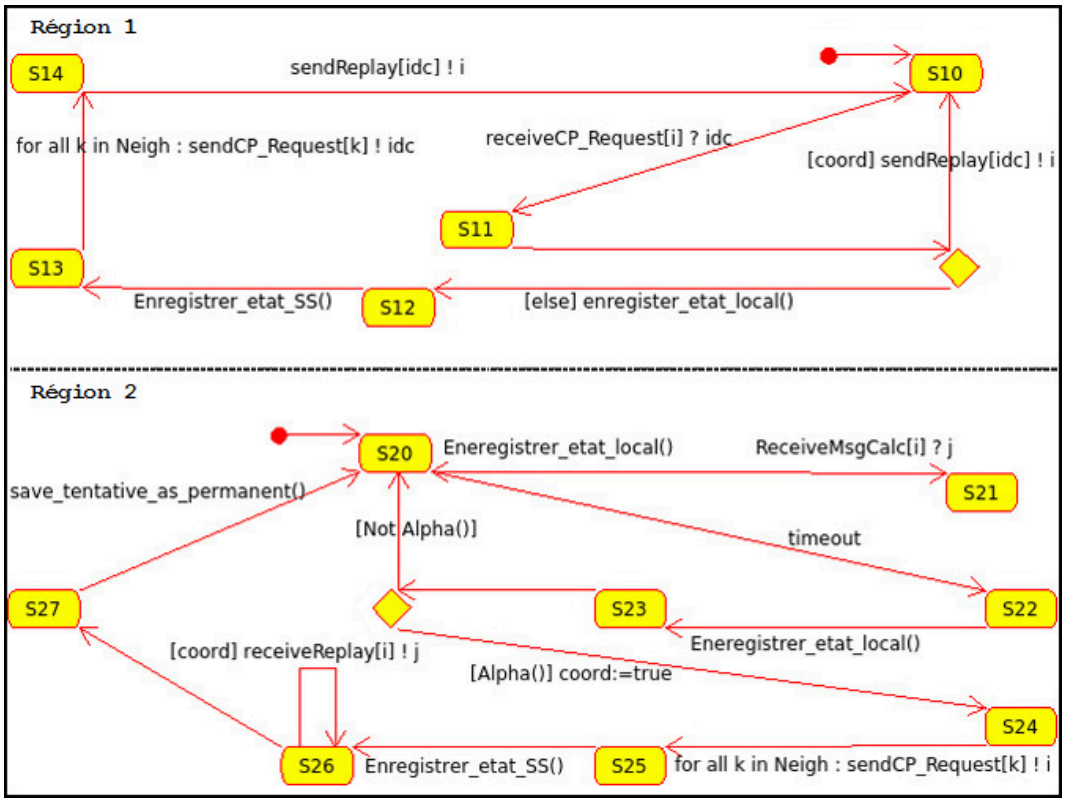

FIGURE 6.4 – Diagramme d'états du checkpointing.

6.8 Idée de base du protocole de calcul de points de contrôle

Au mieux de notre connaissance, tous les protocoles proposés supposent une topologie spécifique du réseau comme la clusterisation. Par ailleurs, nous constatons dans la littérature que les algorithmes proposés exploitent exclusivement une seule classe de protocoles.

Nous proposons dans le cadre de nos travaux un protocole hybride qui fonctionne en deux phases et qui combine deux techniques de calcul de points de contrôle. L'utilisation de deux phases au lieu d'une seule (comme c'est le cas dans la majorité des protocoles de checkpointing qui existent), permet de trouver un compromis entre la gestion des disponibilités des ressources et de la précision de la tâche de calcul des points de contrôle qui influe sur l'efficacité du protocole de recouvrement.

Nous montrons, plus loin, comment notre protocole peut fonctionner de ma-

nière indépendante des contraintes de mobilité, de la topologie du réseau, des structures logiques et de l'éventuel protocole de routage. Nous montrons aussi comment le *recouvrement arrière* (*rollback recovery*) peut être une tâche simple et allégée en utilisant notre protocole de calcul de points de contrôle.

Pour atteindre ces objectifs, nous avons considéré les points conceptuels suivants comme le précise le diagramme d'états de la figure 6.4 :

— La transmission des messages de contrôle est effectuée uniquement dans le voisinage de chaque nœud. Cela rend le protocole de calcul des points de contrôle indépendant de la topologie physique ou l'architecture logique du réseau.

  Sur un autre plan, ça nous permet également une utilisation efficace de la bande passante contrairement à la technique d'inondation utilisée par la majorité de protocoles de checkpointing coordonnés.

  Par ailleurs, les changements dans la topologie du réseau (dûe à la mobilité de certains nœuds et les déconnexions et re-connexions des autres nœuds), n'ont pas d'effet sur le protocole de checkpointing.

— Nous fournissons des traitements spécifiques pour les déconnexions et reconnexions des nœuds. En effet, la déconnexion d'un nœud est un phénomène ordinaire dans les MANETs qui ne doit pas être confondu avec l'occurrence d'une panne.

— Nous proposons d'utiliser la *réplication* dans la perspective de résoudre le problème de la mise en place de la notion de *mémoire stable* (*stable storage*) dans l'environnement ad hoc.

— Nous minimisons le nombre de variables dans la perspective de minimiser l'espcae mémoire utilisé. Nous utilisons uniquement des structures de données simples et scalaires pour assurer un bon passage à l'échelle (scalability).

Notre protocole 2PACA (Two Phases Algorithm of Checkpointing for Adhoc mobile networks) [10][11] est un protocole de checkpointing (calcul de points de contrôle) hybride qui se déroule en deux (02) phases et combine deux (02) techniques différentes de calcul de points de contrôle. Notre protocole est conçu pour les réseaux mobiles ad hoc plats. Ainsi, il traite les contraintes de mobilité tout en étant indépendant de toute topologie, structure logique ou protocole de routage.

Lors de la première phase, nous optons pour la technique de *checkpointing*

quasi-synchrone (classe de protocoles *induits communication*) dans le but de minimiser la latence, le surcoût en messages de contrôle tout en garantissant une *ligne de recouvrement cohérente*. Voir la région 2 de la figure 6.4.

- Un nœud i réalise un calcul local qui rentre dans le cadre du calcul distribué global. Trois événements importants peuvent survenir : calcul interne, envoi de messages et réception de messages.
- Périodiquement, le nœud i enregistre localement son état (point de contrôle / checkpoint) indépendamment des autres nœuds. Ceci est représenté par la transition entre les états $S20$ et $S22$ puis la transition de l'état $S22$ vers $S23$ sur la figure 6.4.
- Chaque point de contrôle est identifié par son numéro de séquence dont le dernier est toujours embarqué dans le message de calcul. Un nœud peut être aussi forcé à prendre un point de contrôle s'il reçoit un message de calcul ayant un numéro de séquence supérieur au sien (voir les transitions entres les états $S20$ et $S21$ sur le diagramme de la figure 6.4).

Le problème de cette phase est la génération d'un nombre important de points de contrôle. Dans un but de réduire ce nombre, nous avons opté pour une phase de synchronisation.

La seconde phase se base sur la technique de *checkpointing coordonné* (*synchrone*). Cette phase est lancée quand le nombre de points de contrôle sauvegardés localement atteint un certain seuil α (voir état $S24$ du diagramme d'états de la figure 6.4). L'objectif est de constituer un point de contrôle global cohérent qui est sauvegardé en mémoire stable (notée SS pour stable storage). Ceci est représenté sur le diagramme d'états de la figure 6.4 à travers la transitions de $S25$ vers $S26$ (pour le coordinateur) et de $S12$ vers $S13$ (pour un nœud ordinaire).

6.9 Conclusion

Nous avons présenté le fonctionnement global de notre outil de tolérance aux pannes. Cet outil qui est constitué de plusieurs protocoles qui s'exécutent en arrière plan de l'application distribuée. Nous avons expliqué comment ces différents protocoles se synchronisent.

Nous avons présenté quelques diagrammes pour décrire les différents protocoles et leurs interactions. Les descriptions détaillées de ces protocoles sont données dans les deux prochains chapitres.

Chapitre 7

Protocoles de Détection de Pannes

7.1 Introduction

Plusieurs protocoles de détection de pannes ont été définis dans la littérature. Quelques uns ont été cités dans le chapitre 4. La majorité de ces travaux sont définis dans le but de résoudre le problème du *consensus* ou le problème d'*accord de* $k-ensembles$ ($k-set\ agreement$).

Par ailleurs, le problème majeur des détecteurs de pannes en environnement mobile est de confondre entre la panne et les déconnexions qu'elles soient volontaires ou involontaires. Ainsi, il y a risque de suspecter faussement des nœuds saints (vivants) qui passent en mode déconnecté. L'augmentation du nombre de fausses suspicions dans le système risque de faire revenir le calcul distribué en arrière inutilement. A cet effet, dans la suite de nos protocoles, que nous présentons dans ce chapitre, nous tentons de réduire au maximum le nombre de fausse suspicions. Par voie de conséquence, notre objectif consiste à concevoir un protocole qui assure la justesse (précision) la plus élevée possible.

Dans ce chapitre, nous détaillons nos différentes contributions dans le domaine de la détection de pannes. Trois protocoles sont présentés : $FDAN$ [7], $AFDAN$ [8] et $FDRAM$ [9]. Nous présentons aussi une analyse de performances faite à l'aide du simulateur $NS2$ [88].

7.2 Environnement et hypothèses

Nous reprenons l'environnement de calcul ainsi que les hypothèses citées dans la section 6.4 du chapitre 6. Nous citons :

- Nous supposons que le système est en environnement mobile ad hoc.
- Nous considérons que les nœuds sont des unités mobiles hétérogènes où s'exécutent les processus du calcul distribué.
- Le système progresse de façon asynchrone avec la possibilité que des nœuds tombent en panne.
- Le réseau est constitué de N ($N \geq 3$) nœuds. La valeur de N n'est pas définie au préalable. En effet, des nœuds peuvent quitter le système et d'autres peuvent arriver à tout moment. Par conséquent, le nombre de nœuds participants au calcul distribué varie durant l'exécution.
- Cependant, au cours du processus de calcul, les nœuds peuvent se déconnecter [1] temporairement ou définitivement pour quitter les système. Il peuvent aussi être éliminés ou exclus et remplacés par de nouveaux et ce après l'occurrence d'une défaillance.
- Chaque nœud possède un identificateur unique dans le réseau.
- Initialement, un nœud ne connait pas nécessairement le nombre et les identités des nœuds dans le système.
- Nous admettons uniquement que chaque nœud doit connaitre les identités de ses voisins directs grâce aux échanges fait au niveau de la couche MAC.
- Notons aussi que les processus des différents nœuds communiquent par envoi et réception de messages à travers des canaux de communication sans fils.
- Nous évitons l'obligation d'utiliser un protocole de routage bien particulier. Ainsi, notre protocole et ses performances ne sont pas affectés par l'organisation du réseau (plate ou hiérarchique) et du protocole de routage.
- Les nœuds peuvent se déconnecter ou se reconnecter à tout instant.
- Nous supposons que tous les nœuds sont dotés d'un mécanisme de connectivité capable de détecter le niveau de batterie et l'intensité du signal. Dans le cas où le niveau de l'un de ces deux paramètres descend sous un certain seuil, le mécanisme de connectivité est capable de diffuser un mes-

1. La déconnexion peut être volontaire ou non volontaire.

sage particulier ce qui permet au niveau applicatif de diffuser un message de déconnexion dans le voisinage.

De plus, nous rajoutons les hypothèses spécifiques aux protocoles de détection de pannes.

— Par ailleurs, un nœud peut devenir défaillant uniquement à travers une panne franche (crash). La panne franche est permanente. Toutes les données du nœud défaillant sont perdues définitivement. En d'autres termes, un nœud défaillant via un crash ne peut pas recouvrir après sa défaillance. Il peut être réparé ou remplacé par un autre nœud mais il perd localement tous ses états dans le calcul distribué.

— Comme nous l'avons indiqué ci-dessus, le système est supposé asynchrone mais le fait d'utiliser des temporisateurs dans le mécanisme de décision qui est expliqué plus loin rend le système partiellement synchrone. Par ailleurs, nous notons que la transmission des messages dans les canaux de communication ne suit pas nécessairement la politique FIFO.

7.3 Principes de fonctionnement des protocoles

Nous considérons dans la conception de notre protocole les points suivants :

— Dans notre cas, nous avons opté pour la technique de battements de cœur (heartbeat) [4]. Ce choix est justifié par le fait que cette technique génère beaucoup moins de messages que la technique requête (pinging).
En effet, la réduction du nombre de messages de contrôle générés par un protocole est très importante dans un environnement mobile. Il contribue à sauvegarder l'énergie et la bande passante.

— Notre protocole utilise principalement l'échange de messages afin de prouver la vivacité des nœuds. C'est à dire, pour distinguer entre un nœud vivant et un nœud défaillant.
Pour cet effet, Comme pour les protocoles existants basés sur la technique de battements de cœur, un message particulier de vivacité est implémenté. Le rôle de ce message est d'informer les processus des nœuds voisins qui peuvent le recevoir, que le nœud émetteur est vivant.

— Le message de *vivacité* est insuffisant pour réaliser un consensus dans tout le système autour de la liste des nœuds suspects.
En effet, ce message contient seulement le nombre de battements de cœur

(heartbeats) de l'émetteur. Ainsi, nous introduisons un autre message particulier de *correction*. Dans ce message, des informations supplémentaires sont embarquées. Ce message permet aux destinataires de mettre à jour leurs différentes listes et plus particulièrement la liste de suspects.
Par voie de conséquence, ce message va permettre de propager rapidement l'information dans le réseau ce qui permet d'assurer une bonne complétude.

7.4 Protocole FDAN

Dans le protocole FDAN (Failure Detection protocol for Ad hoc Networks) [7], nous proposons d'utiliser deux niveaux de listes de suspects : Une qui est temporaire et l'autre est finale. Le passage de la liste temporaire vers la liste finale est géré par deux temporisateurs (timeouts). Ceci exige de personnaliser la gestion des timers pour chaque nœud.

En plus du message de vivacité (I_Alive), nous utilisons des messages de correction ($need\_correction$) qui ont comme objectif de propager l'information sur les nœuds défaillants et surtout de corriger rapidement les fausses suspicions quand elles sont identifiées.

Par ailleurs, afin d'éviter la congestion du réseau avec ces messages, nous proposons d'utiliser une variable compteur, appelée R. Chaque nœud incrémente son compteur R à chaque envoi du message I_Alive. Lorsque sa valeur atteint un certain seuil prédéfini K, le message $need\_correction$ est diffusé aux voisins à la place du message I_Alive. Plus concrètement, de manière cyclique, chaque séquence de $K-1$ messages I_Alive est suivie par un message $need\_correction$.

La détermination de la valeur de K est très importante. Si la valeur considérée est très grande, alors les messages $need\_correction$ deviennent rares dans le réseau et par conséquent ils n'ont pas une utilité importante. Ainsi, le nombre de fausses suspicions risque d'augmenter et il est difficile voire impossible de corriger ces fausses suspicions. De plus, toute panne réelle détectée prend un temps considérable pour qu'elle soit propagée dans le réseau.

Par contre, si la valeur qui est prise en considération pour le seuil K est très petite, alors le nombre des messages $need\_correction$ dans le réseau devient important. Ces messages qui sont de tailles assez grande (contenant plusieurs listes) risquent de faire augmenter le trafic dans le réseau (congestion, collisions,

...) et font augmenter la consommation de certaines ressources importantes dans le contexte MANET comme la bande passante et l'énergie.

Le second aspect de notre propositions traite les contraintes de l'environnement mobile. Nous admettons que tout échange de messages dans notre protocole se fait uniquement entre voisins. Ceci rend le protocole indépendant de toute structure ou protocole de routage et évite de diffuser l'information dans tout le réseau ce qui peut être coûteux dans le contexte des MANETs. Par ailleurs, pour économiser la bande passante et réduire la consommation d'énergie nous nous sommes intervenus sur les échanges de messages. Nous avons utilisé le minimum possible de messages de contrôle et en même temps réduit la quantité de données embarquées dans ces messages. De plus, nous fournissons des traitements spécifiques dans le cas des déconnexions (volontaires ou involontaires) et re-connexions des nœuds.

Dans cette section, nous présentons notre protocole, appelé $FDAN$: *Failure Detection for Ad hoc mobile Networks* [7].

Nous décrivons dans ce qui suit notre protocole en donnant les principales structures de données, messages ainsi que le fonctionnement des différentes tâches.

7.4.1 Structures de données

Chaque processus P_i au niveau d'un nœud i maintient les structures de données suivantes :

- H_i : Tableau d'entiers initialisés à 0. Chaque élément $H_i[j]$ est un compteur de heartbeats du nœud j selon le message I_Alive ou $need\_correction$, envoyé par j à i
- K : Constante entière. Tous les processus utilisent la même valeur pour K.
- R_i : Variable entière initialisée à 0. Il s'agit d'un compteur qui est incrémenté quand i envoie un message I_Alive. Lorsque R_i atteint la valeur K, il est remis à 0.
- $Neighbors_i$: Liste de nœuds. Cette liste contient les identificateurs des voisins du nœud i à un instant donné.
- $Participant_i$: Liste de nœuds. Elle contient les identités des nœuds participant au calcul distribué et qui sont connus par i. Initialement, nous avons $Participant_i = Neighbors_i$. La réception du message $need\_correction$ permet d'enrichir les connaissances de i sur les nœuds participants au cal-

cul.

— $susp\_temp_i$: Liste de nœuds. Elle contient les identificateurs des nœuds suspectés temporairement par i. Cette liste peut contenir les nœuds dont les processus sont lents, les nœuds défaillants ou ceux qui se sont éloignés du voisinage de i.
— $susp\_final_i$: List de nœuds. Elle contient les identificateurs des nœuds suspectés définitivement de défaillance par i.
— $TO2$: Constante entière. Elle dénote la valeur du second timeout.
— $TO1_i$: Tableau d'entiers. $TO1_i[j]$ contient la valeur du premier délai associé à j par le nœud i. Initialement, on affecte à $TO1_i[j]$ la valeur de $TO2$.
— $deadline_i$: Tableau d'entiers ; $deadline_i[j]$ contient le temps absolu (réel) des timeouts (délais) associés à j. Soit T le moment où le dernier heartbeat (message de vivacité) de j a été reçu par P_i. Si $P_j \notin susp\_temp_i$, alors $deadline_i[j] = T + TO1_i[j]$, dans le cas contraire nous avons $deadline_i[j] = T + TO1_i[j] + TO2$.

7.4.2 Messages

Chaque processus P_i qui s'exécute au niveau du nœud i gère les messages suivants :

— $I\_Alive(i, H_i[i])$: Il s'agit du message de battement de cœur pour prouver la vivacité du nœud émetteur. Il contient l'identificateur de l'émetteur et la valeur actuel de son heartbeat. Notons que ce message est diffusé seulement aux voisins directs (voisins MAC) de i.
— $need\_correction(i, H_i[i], Participant_i, susp\_final_i)$: Il s'agit d'un message de heartbeat particulier. Il contient les listes $Participant_i$ et $susp\_final_i$. Chaque élément de ces listes est une paire $(j, H_i[j])$. Ce message est diffusé seulement aux voisins de i.
— $Disconnect(i, H_i[i])$: Ce message est diffusé par i dans son voisinage quand son mécanisme de connectivité détecte un niveau de batterie en dessous d'un certain seuil ou l'affaiblissement du signal. Il est aussi diffusé dans le voisinage lors des déconnexions volontaires (un nœud qui décide volontairement de quitter l'application temporairement ou définitivement)
— $Reconnect(i, H_i[i])$: Ce message est envoyé par i à son voisinage quand

```
    // Task 1
1   Every T units of time {
2     H_i[i] ++;
3     R_i ++;
4     if  (R_i = = k) {
5       P = S = ∅;
6       for all (j ∈ Participant_i)
7         P = P ∪ {(j, H_i[j])};
8       for all (P_j ∈ Susp_final_i)
9         S = S ∪ {(j, H_i[j])};
10      Broadcast need_correction (i, H_i[i], P, S) to Neighbors_i;
11      R_i = 0;
12    } else Broadcast I_Alive(i, H_i[i]) to Neighbors_i;
13    foreach ( (entry j in TO1_i[j]) and (not (j ∈ Neighbors_i)) )
14      delete TO1_i[j] and deadline_i[j];
15  }
```

FIGURE 7.1 – FDAN : Tâche 1.

il décide de se reconnecter et réintégrer l'application.

7.4.3 Description détaillée du protocole

L'algorithme consiste en quatre tâches concurrentes.

Ces tâches nécessitent d'appeler les procédures définies dans la figure 7.5. Dés qu'un nœud est connecté au réseau, il exécute la procédure d'initialisation. Par la suite, il réalise de manière répétitive les quatre tâches concurrentes.

— **Tâche 1 :** Chaque processus s'exécutant sur un nœud i incrémente périodiquement la valeur de son battement de cœur ainsi que celle de R_i (lignes 2 et 3 de la figure 7.1). Par la suite, il envoie soit le message *I_Alive* soit le message *need_correction* vers tous ses voisins (lignes 10 et 12 de la figure 7.1) selon la valeur de R_i (ligne 4 de la figure 7.1).

— **Tâche 2 :** Quand un nœud i reçoit un message *I_Alive* de la part de j (lignes 1 et 2 de la figure 7.2), il met à jour le heartbeat correspondant à j (ligne 14 de la figure 7.5) et la valeur du premier temporisateur (timeout) associé à j ainsi que ses divers listes : $Participant_i$, $susp\_temp_i$ et $susp\_final_i$. Quand un nœud i reçoit un message *need_correction* de la part de j (ligne 4 de la figure 7.2), il effectue le même traitement que lors de la réception d'un message *I_Alive* (procédure *hb_reception* définie dans la figure 7.5), puis il compare ses propres listes avec les listes de suspects et de participants reçues de la part de j (lignes 6 à 15 et lignes 16 à 21 de la figure 7.2). Il prend l'information la plus fraiche (récente)

```
    // Task 2
1   When i receives I_Alive(j, h) from j {
2     hb_reception(j, h);
3   }
4   When i receives need_correction(j, h, P, S) from j {
5     hb_reception(j, h);
6     for all ( (q, hb) ∈ P ) {
7       if (q≠i) {
8         if ((q ∈ susp_final_i) or (q ∈ susp_temp_i)) and (H_i[q] < hb) {
9           H_i[q]=hb;
10          Update (1,q);
11        } else
12          if ((q ∈ Participant_i) and (H_i[q] < hb)) {
13            H_i[q]=hb;
14            update(1,q);
15    } } }
16    for all ( (q, hb) ∈ S ) {
17      if (q≠i){
18        if (H_i[q] < hb) {
19          H_i[q]=hb;
20          update(2,q);
21    } } }
22  }
```

FIGURE 7.2 – FDAN : Tâche 2.

selon la valeur du heartbeat (lignes 8, 12 et 18 de la figure 7.2).

En termes plus concrets, si i reçoit de la part de j une information conflictuelle concernant une autre nœud s, alors il fixe la valeur locale du heartbeat associé au nœud s à la plus grande valeur (entre la valeur locale et celle reçue).

D'autre part, le consensus sur la défaillance d'un nœud est élaboré de façon implicite comme suit : Lorsque i reçoit de la part de j un état différent sur le voisin s alors il prend en considération l'état le plus récent selon le heartbeat de s et plus exactement selon $H_i[s]$ et $H_j[s]$. Si i et j signalent deux états différents pour s mais avec les mêmes valeurs du heartbeat de s, alors s est considéré comme défaillant par i.

— **Tâche 3 :** Après l'expiration du premier temporisateur (timeout) associé à j par i (ligne 1 de la figure 7.3), i ajoute j à sa liste de suspects temporaires (lignes 3 et 4 de la figure 7.3) puis arme (déclenche) le second timeout. A l'expiration du second timeout, le nœud i ajoute j à sa liste de suspects finale (lignes 7 à 8 de la figure 7.3).

Néanmoins, l'utilisation efficace d'un tel mécanisme nécessite une bonne gestion des temporisateurs. Pour cet effet, le traitement de tous les délais

```
    // Task 3
1   When deadline_i[j] reached {
2     if (j ∈ susp_temp_i) {
3       susp_final_i = susp_final_i ∪ {j};
4       susp_temp_i = susp_temp_i - {j};
5       delete deadline_i[j] and TO1_i[j];
6     } else {
7       update(2,j);
8       deadline_i[j] = current_time() + TO2;
9   }}
```

FIGURE 7.3 – FDAN : Tâche 3.

d'attente (temporisateurs) est complexe et a besoin de procédures particulières dont les fonctionnalités sont définies dans la procédure *hb_reception* de la figure 7.5. Si le nœud j répond à i avant l'expiration du premier délai $TO1_i[j]$, alors le premier timeout qui lui est associé $TO1_i[j]$ est décrémenté par i (voir ligne 14 de la figure 7.5). Toutefois, s'il répond après l'expiration du premier délai, alors $TO1_i[j]$ est incrémenté (voir ligne 18 de la figure 7.5).

— **Tâche 4 :** Lorsqu'un nœud décide de se déconnecter volontairement ou de façon non vonlontaire (information fournie par le mécanisme de connectivité : affaiblissement de la batterie ou du signal reçu), il incrémente son heartbeat puis envoie un message de déconnexion à tous ses voisins (lignes 1 à 3 de la figure 7.4). A la réception par i du message de déconnexion envoyé par j, le nœud j est retiré des listes de suspicions temporaire et finale (s'il existe dans ces listes) et tous ses timeouts sont fixés à la valeur infinie (∞). Ceci lui évite d'être suspecté pendant sa période de déconnexion. Tout ceci est indiqué sur les lignes 5 à 9 de la figure 7.4.
Par ailleurs, lorsqu'un nœud i décide de se re-connecter au système, il diffuse un message de *reconnect* dans son voisinage (lignes 11 à 13 de la figure 7.4). A la réception d'un tel message par un nœud i de la part de j, alors i ajoute j à sa liste de ses voisins ($Neighbors_i$) et à la liste des nœuds participants au calcul ($Participant_i$) puis réinitialise son premier timeout (voir lignes 15 à 19 de la figure 7.4).

```
    // Task 4
1   When i decides to disconnect {   // voluntary or due to connectivity mechanism
2     H_i[i]++;
3     Broadcast disconnect (i, H_i[i]) to Neighbors_i;
4   }
5   When i receives disconnect (j,h) from j {
6     TO1_i[j] = ∞;
7     deadline_i[j] = ∞;
8     H_i[j] = h;
9     update(1,j);
10  }
11  When i decides to reconnect to the network {
12    H_i[i]++
13    Broadcast reconnect (i, H_i[i]) to Neighbors_i;
14  }
15  When i receives reconnect (j, h) from j {
16    TO1_i[j]= TO2;
17    deadline_i[j]= current_time () + TO1_i[j];
18    H_i[j] = h;
19    update(1,j);
20  }
```

FIGURE 7.4 – FDAN : Tâche 4.

7.5 Protocole AFDAN

Nous proposons d'améliorer notre protocole FDAN [7] en modifiant la phase de correction. Par conséquent, nous proposons d'utiliser des moyens asynchrones.

Dans *AFDAN* [8], les timeouts sont utilisés seulement pour détecter les nœuds dont les processus sont lents et les distinguer de ceux qui sont actifs. Ainsi, il est possible de considérer (temporairement) un nœud comme défaillant suite à l'expiration du délai alors qu'en réalité il est lent ou il s'est déplacé. D'ailleurs, des mécanismes particuliers sont introduits pour apporter les corrections nécessaires. Ceci est obtenu par l'échange et la gestion des messages de correction notés $Corr$.

Le message de correction a pour objectif la vérification et la correction des fausses suspicions. On embarque dans ce message une variable appelée *path* qui représente une liste de triplets (i, h, e) où i est l'identité d'un nœud, h la valeur de son HertBeat et e son état (vivant, suspecté ou déconnecté). Ce message est diffusé par un nœud initiateur (au début du calcul) à ses voisins en embarquant son identité, son heartbeat et son état dans la variable *path*. En recevant un tel message, les autres nœuds corrigent leurs vues sur la vivacité des autres nœuds. Ils mettent à jour le message en se rajoutant à la liste. Ils peuvent aussi corriger les informations erronées dans ce message et suppriment les plus

```
1   init () {
2     Participant_i = Neighbors_i;
3     Susp_temp_i = Susp_final_i = ∅;
4     H_i[i] = 0;
5     R_i = 0;
6     For all (q ∈ neighbors_i) {
7       H_i[q]=0;
8       TO1_i[q]=TO2;
9   }}
10  hb_reception(b, h) {
11    if (TO1_i[b] = = NULL)   TO1_i[b]=TO2;
12    if (b ∈ Participant_i) {
13      H_i[b] = h;
14      TO1_i[b] = TO1_i[b]+( current_Time() - deadline_i[b])/2;   //decrement TO
15    } else {
16        if ((b ∈ susp_temp_i)or(b ∈ susp_final_i)) {
17          update(1,b);
18          TO1_i[b] = TO1_i[b]+1;   //increment TO
19        } else {   // i don't know b
20          Participant_i= Participant_i ∪ {b};
21          TO1_i[b] = TO2;
22      }}
23    deadline_i[b] = current_time() + TO1_i[b];
24  }
25  update(a, b) {
26    Participant_i = Participant_i - { b };
27    susp_temp_i = susp_temp_i - { b };
28    susp_final_i = susp_final_i - { b };
29    if (a= =1) Participant_i = Participant_i∪{ b };
30     else     susp_temp_i = susp_temp_i∪{ b };
31  }
```

FIGURE 7.5 – FDAN : Les procédures.

anciennes informations.

7.5.1 Structures de données

Pour son bon fonctionnement, $AFDAN$ associe pour chaque nœud i les structures de données suivantes :

— $Neigh_i$: Il s'agit d'un ensemble d'identificateurs de nœuds. Cet ensemble contient la liste des voisins du nœud i. Ce sont les voisins MAC du nœud i à un instant donné. Comme cet ensemble peut changer à cause de la mobilité, des déconnexions et re-connexions, cet ensemble est calculé par la couche MAC puis mis à jour à chaque fois qu'un nœud envoie un heartbeat. Dans le contexte des $MANET$, la couche MAC est munie de mécanismes qui lui permettent de déterminer les voisins directs d'un nœud

donné i et ce en échangeant des messages spécifiques appelés $Hello$.

— $Part_i$: Il s'agit d'un ensemble d'identificateurs de nœuds. Cet ensemble contient la liste de nœuds connus par le nœud i. Initialement, elle contient tous les voisins directs du nœud i.

— H_i : Cette structure est un tableau dynamique d'entiers. Chaque élément de ce tableau $H_i[j]$ donne la valeur de battement de cœur d'un nœud donné j telle qu'elle a été reçue par le nœud i, sachant que $j \in Part_j$. Les valeurs de ce tableau sont initialisées à 0 au lancement du protocole.

— E_j : Il s'agit d'un tableau d'états. Chaque élément $E_i[j]$ désigne l'état du nœud j tel que le voit le nœud i, tel que $i \in Part_j$. $E_i[j]$ peut prendre les valeurs suivantes :

-Active : Si j est vu par i comme vivant (actif ou correct).

-Tsusp : Si j est suspecté temporairement de défaillance par le nœud i. En d'autres termes, j est mis dans la liste temporaire de suspects gérée par le nœud i.

-Fsusp : Si j est suspecté réellement de défaillance par i. Ceci signifie que j est mis dans la liste finale de suspects de i.

-Disc : Si j est déconnecté.

— STO : C'est une constante, dénotant une valeur de seuil d'un délai d'attente. Cette valeur est la même quel que soit le nœud que nous considérons dans le système. En d'autres termes, elle ne varie pas d'un nœud à un autre et elle ne varie pas durant l'exécution du protocole. Nous expliquons, plus loin, à comment et quand elle est utilisée.

— $FTO_i[j]$: Il s'agit d'un tableau dynamique, tel que chaque élément $FTO_i[j]$ désigne la valeur du premier timeout associé à un nœud j par le nœud i, tel que $j \in Neigh_i$. Initialement, on affecte à $FTO_i[j]$ la valeur (constante) prédéfinie STO.

— $Ddl_i[j]$: Tableau de dates, tel que chaque élément de ce tableau $Ddl_i[j]$ désigne l'instant du dernier délai où le nœud i devrait recevoir le battement du cœur du nœud j, sachant que $j \in Neigh_i$. Si cet instant est dépassé, il doit suspecter (temporairement ou finalement) ce nœud.

7.5.2 Messages

$AFDAN$ se base sur l'échange d'information à travers l'envoi et réception de messages dans la perspective de surveiller la vivacité des autres nœuds du sys-

tème. Ceci permet de distinguer entre les nœuds actifs (vivants) et ceux suspectés de défaillance.

Pour cet effet, nous utilisons dans $AFDAN$ divers types de messages :

— $Alive(i,\ H_i[i])$: Il s'agit d'un message de battement de cœur (heartbeat). Chaque nœud i diffuse ce message à tous ses voisins afin de les informer qu'il est dans un état actif (vivant) et aussi pour transmettre la valeur courante (actuelle) de son heartbeat ($H_i[i]$).

— $Corr(path)$: Ce message est envoyé par un nœud i dans la perspective de corriger une fausse suspicions et de mettre à jour les listes de suspects gérées par les autres nœuds. $Path$ est une liste de triplets (j, h, e) où j est l'identificateur d'un nœud que connaît déjà i ; $h = H_i[j]$ (dernière valeur de battement de cœur de j connue par i) et $e = E_j[i]$ (l'état de j selon i).

— $Disc(i,\ H_i[i])$: Ce message est diffusé par le nœud i à ses voisins afin de les informer qu'il va se déconnecter. C'est le mécanisme de connectivité qui lui permet de savoir à quel moment il doit envoyer ce message.

— $Rec(i,\ H_i[i])$: Ce message est diffusé par le nœud i à ses voisins afin de les informer qu'il vient tout juste de réintégrer (re-connexion) le réseau.

Comparativement à $FDAN$, presque les mêmes structures de données sont utilisées dans $AFDAN$, à l'exception de la structure *path* qui a été introduite dans le message de correction.

7.5.3 Description détaillée du protocole

Le protocole $AFDAN$ est constitué de deux composants : détection de pannes et le gestionnaire des déconnexions.

Le composant de détection de pannes

Le module de détection de pannes consiste en cinq tâches concurrentes exécutées par chaque nœud i. Les tâches sont décrites ci-dessous.

F.Tâche1. Envoi du heartbeat : Le nœud i incrémente la valeur de son battement de cœur ($H_i[i]$) puis diffuse périodiquement (toutes les β unités de temps) un message de vivacité ($Alive(i,\ H_i[i])$) à ses voisins directs ($Neigh_i$). Ceci est décrit à travers les lignes 2 et 3 de l'algorithme défini sur la figure 7.6.

F.Tâche2. Réception du heartbeat : Les traitements effectués par un nœud i lors de l'exécution de cette tâche sont détaillés dans l'algorithme de la

```
    // F.Task 1
1   Every T units of time {
2     Hi[i] ++;
3     Broadcast Alive(i, Hi[i]) to Neighi;
4     foreach ( (entry j ∈ FTOi[j]) and (j ∉ Neighi) )
5       delete Ddli[j];
6   }
```

FIGURE 7.6 – AFDAN : Composant détection de pannes - Tâche 1.

```
    // F.Task 2
1   When i receives Alive(j, h) from j {
2     if (FTOi[j] = = NULL)  FTOi[j]=STO;
3     Hi[j] = h;
4     if (Ei[j]= ='Active') FTOi[j] = FTOi[j]+( current_Time() - Ddli[j])/2;
5     else {
6       if (Ei[j] = = 'Tsusp') or (Ei[j] = = 'Fsusp'))
7             FTOi[j] = FTOi[j]+1;   //increment TO
8       else {   // i don't know j
9             Parti= Parti ∪ {j};
10            FTOi[j] = STO;
11    } }
12    Ei[j]='Active';
13    Ddli[j] = current_time() + FTOi[j];}
14  }
```

FIGURE 7.7 – AFDAN : Composant détection de pannes - Tâche 2.

figure 7.7. En effet, à chaque fois que le message $Alive(j,\ h)$ (où h est valeur du heartbeat locale envoyée par j, c'est à dire $h = H_j[j]$) est reçu par le nœud i de la part du nœud j, on affecte au battement de cœur associé à ce dernier par i ($H_i[j]$) la valeur reçue (h) comme indiqué sur la ligne 3 de la figure 7.7.

Si le message est reçu pour la première fois [2], alors le nœud j est rajouté à la liste de nœuds participants connus par i ($Part_i$) et le délai $Ddl_i[j]$ associé par i à j est créé et initialisé à la première valeur du timeout ($FTO_j[i] = STO$). Ces traitements sont décrits sur lignes 8 à 10 de la figure 7.7.

Dans le cas où i connaît j, le heartbeat reçu de la part du nœud j par le nœud i, est traité par le protocole selon les cas ci-dessous :

— Si $E_i[j] = Active$ (le nœud j est considéré correct par i), alors la valeur du premier timeout $FTO_i[j]$ associé par i à j est décrémenté comme le montre la ligne 4 de la figure 7.7. En effet, comme le nœud j est vu par i comme étant rapide dans ses réponses, le premier délai (timeout) est

2. Ceci pourrait correspondre au démarrage du protocole ou lorsqu'un nouveau nœud vient juste d'intégrer la liste des voisins de i à cause de la mobilité.

```
    // F.Task 3
1   When Ddl_i[j] reached {
2     if (E_i[j] = = 'Tsusp') {
3       E_i[j] = 'Fsusp';
4       delete Ddl_i[j];
5     } else {
6       E_i[j] = 'Tsusp';
7       Ddl_i[j] = current_time() + STO;
8   } }
```

FIGURE 7.8 – AFDAN : Composant détection de pannes - Tâche 3.

ajusté en conséquence pour la prochaine période.

— Si $E_i[j] = Tsusp$ ou $Fsusp$ (ligne 6, figure 7.7), alors la valeur du premier timeout associé à j, $FTO_i[j]$ est incrémenté. Ceci signifie que le nœud i réagit de façon lente et ainsi le premier délai doit être ajusté en conséquence pour la prochaine période (voir ligne 7, figure 7.7). Ainsi, cette réception du message de battement de cœur permet la correction d'une fausse suspicion. De ce fait, l'état du nœud j est mis à jour (ligne 12, figure 7.7).

Ensuite, Dans tous les cas, le protocole met à jour l'état de j comme étant actif (vivant) (voir ligne 12 de la figure 7.7) puis affecte au temporisateur associé à j ($Ddl_i[j]$) la valeur (recalculée) du premier timeout, en attendant de recevoir le nouveau message de vivacité ($Alive$) (voir ligne 13, figure 7.7).

F.Tâche3. Expiration du Timeout : Chaque nœud i gère des temporisateurs pour tous les nœuds voisins. Lorsque le délai est atteint (ligne 1, figure 7.8), le protocole effectue des traitements spécifiques selon la situation :

— Si le nœud j est vu comme actif par le nœud i (ligne 5, figure 7.8), alors j est placé dans la liste temporelle de suspects de i (ligne 6, figure 7.8). Un second délai est accordé à j par i comme dans $FDAN$. Il s'agit d'une extension du délai final en vue de recevoir son message de battement de cœur avant de confirmer la suspicion de défaillance (ligne 7, figure 7.8).

— Si le nœud j est dans la liste de suspects temporelle de i (ligne 2, figure 7.8), qui signifie que l'extension du délai est consommée sans recevoir de signe de vivacité de la part de j. Par conséquent, ce dernier est retiré de la liste temporelle de suspects puis mis dans la liste finale de suspects (ligne 3, figure 7.8) et le temporisateur $Ddl_i[j]$ est désactivé (ligne 4, figure 7.8).

L'extension du délai permet au nœud i de gérer avec une bonne précision les fausses suspicions. La valeur de STO doit être considérée avec précaution. Si

l'on considère une petite valeur de STO, le nombre de fausses suspicions risque d'augmenter. Par ailleurs, si on opte pour une grande valeur de STO, très peu de modifications dans les états des nœuds sont observées et les listes restent presque identiques. Ainsi, il y a risque que le protocole n'atteigne pas la complétude forte. De plus, le temps de détection d'une panne réelle risque d'augmenter.

Dans $AFDAN$, nous supposons que STO est égal au temps entre l'envoi de deux battements de cœur, c'est à dire β. En effet, dans un cas idéal, si un premier battement de cœur est envoyé à l'instant t et le suivant à l'instant $t+\beta$ et si les délais de transmissions sont identique (Δt), alors ces deux battements de cœur sont reçus à $t+\Delta t$ et $t+\Delta t+\beta$. En d'autres termes, ils sont reçu à un intervalle de β unités de temps. Ceci justifie la valeur choisie pour STO.

F.Tâche4. Initialisation de la correction : Cette tâche est exécutée lors de l'initialisation du système. Nous décrivons son algorithme dans la procédure d'initialisation *init* (lignes 3 à 6) sur la figure 7.10.

La correction d'une fausse suspicions est effectuée à travers la circulation dans le réseau des différentes occurrences du message de correction $Corr(path)$. Ce message permet de collecter les informations les plus récentes concernant les états des nœuds par lesquels est passé ce message. Le message $Corr$ est propagé dans les différentes parties du réseau afin de corriger les éventuelles fausses suspicions.

Au lancement du protocole, un nœud particulier k, appelé initiateur, génère la première instance du message de correction $Corr(path)$ puis l'envoie à ses voisins avec $path = (k, H_k[k], E_k[k])$. En d'autres termes, le message ne contient que l'état du nœud initiateur k.

Le choix du nœud initiateur de cette phase est arbitraire. En réalité, plusieurs possibilités sont envisageable mais vu la complexité de l'environnement des MANETs aucune proposition n'apporte de solution efficace. Ainsi, nous avons voulu laisser ce choix ouvert. En effet, on peut par exemple choisir le nœud qui a le degré le plus fort (le plus grand nombre de voisins) ou celui ayant la plus petite identité... Par la suite, différentes instances du message initial $Corr$ sont générées puis propagées dans les différentes parties du réseau.

Cependant, comme nous allons le décrire avec un peu plus de détails dans $F.T$â$che5$ (algorithme décrit dans la figure 7.9), nous introduisons des mécanismes pour contrôler la propagation de ce message dans le réseau et éviter de l'inonder. Parmi ces mécanismes qu'on a pris en considération est de restreindre le nombre de voisins à qui est destiné le message $Corr$ à deux seulement. Ceci

```
    // F.Task 5
1   When i receives Corr(path) from j {
2     prop_i = false;
3     L_i = ∅;
4     for all ((q, h, s) ∈ path) {
5       if (q ∈ Neigh_i) {
6         L_i = L_i ∪ {q}
7         if |L_i|= = 1 c1_i = q;
8         if |L_i|= = 2 c2_i = q;
9     } }
10    CL_i = Neigh_i - L_i;
11    if (|CL_i| > 0) c2_i = (select any q ∈ CL_i);
12    if (|CL_i| > 1) c1_i = (select any q ∈ CL_i and ≠ c1_i);
13    path=update_path (path);
14    if (prop_i) {
15      path = path . (i, H_i[i], 'Active'); //concatenation
16      Send Corr (path) to c1_i and c2_i;
17  } }
```

FIGURE 7.9 – AFDAN : Composant détection de pannes - Tâche 5.

permet de limiter le trafic échangé dans le réseau.

F.Tâche5. Correction des fausses suspicions : La gestion de la correction des fausses suspicions est assurée par les étapes intermédiaires du message $Corr(path)$. A chaque fois que le nœud i reçoit un message $Corr$ de la part de j, il opère de la manière suivante afin de mettre à jour l'information sur les nœuds qui apparaissent dans $path$. Cette mise à jour est définie par les procédures $update\_path$ et $update$ appelées respectivement sur la ligne 13 de la figure 7.9 et la ligne 15 de la figure 7.10. Ces procédures sont définies sur les lignes 9 à 36 de la figure 7.10.

— Pour $b \notin Part_i$ et $(b, H_j[b], E_j[b]) \in path$: le nœud i ajoute le nœud b à ses connaissances comme l'indique la ligne 20 de la figure 7.10. Le nœud i doit copier localement les informations (heartbeat et état) reçues (dans l'algorithme définie dans la figure 7.10, ceci correspond à la mise de la variable u à 1 sur la ligne 20, ce qui permet de savoir qu'il faut mettre à jour l'information locale dans la ligne 32).

— Pour $b \in Part_i$ et $(b, H_j[b], E_j[b]) \in path$: le nœud i vérifie quel est l'état le plus récent (entre l'information locale et l'information reçue). Ces traitements sont définis dans la procédure $update$ dans la figure 7.10. Si $H_i[b] > H_j[b]$ alors i possède l'état de b le plus récent, ainsi il met à jour le triplet de b dans $path$. Dans l'algorithme, nous avons affecté la valeur -1 à la variable u (ligne 24) pour savoir que la mise à jour concerne $path$

(ligne 33).

Si $H_i[b] < H_j[b]$ (lignes 22 et 29) alors i doit mettre à jour localement l'état de b comme suit : $H_i[b]$ et $E_i[b]$ on leur affecte la valeur de battement de cœur et l'état de b contenus dans le triplet du nœud b dans *path* embarquée par le message $Corr$ envoyé par j (ligne 32). Nous avons utilisé la variable u comme de le cas précédent (ligne 22) en lui affectant cette fois-ci la valeur 1. En d'autres termes, dans cette situation, le nœud i met à jour les informations concernant b selon la vue de j ($H_i[b] = H_j[b]$ et $E_i[b] = E_j[b]$).

Le nœud i ne propage pas le message $Corr$, s'il n'a aucune incidence sur la décision du nœud i. On entend par incidence, le fait qu'aucun état n'est mis à jour localement et dans la structure *path*. Dans l'algorithme, la variable u indique s'il y a eu influence ou non du message $Corr$. En effet, quand $u = 0$ aucun traitement n'est effectué (voir la procédure *update* dans la figure 7.10). Comme nous l'avons précisé ci-dessus, la variable u est à 1, si on doit mettre à jour localement l'information et à -1, si on doit mettre à jour l'information sur *path*. Ainsi, pour décider de propager ou non le message $Corr$, nous utilisons une variable booléenne $prop_i$ qui est initialisée à $faux$ (ligne 2, figure 7.9) ce qui indique par défaut qu'il ne faut pas propager *path*. La variable $prop_i$ est mise à $vrai$ uniquement dans le cas où il y a eu mise à jour localement ou dans *path* des informations, c'est à dire $u \neq 0$. Dans ce dernier cas, le message $Corr$ est envoyé à deux voisins de i (lignes 14 à 16 de la figure 7.9).

L'arrêt est justifié par le fait que la non influence du message $Corr$ est due certainement à la réception par i de $Corr$ plusieurs fois dans une courte période ce qui fait que toutes les informations sont à jour (chez i et dans le message $Corr$). Ce qui signifie que plusieurs instances du message $Corr$ circulent dans le réseau sans aucun effet sur la correction. De ce fait, et afin de réduire le coût lié à l'accroissement du trafic, chaque nœud i arrête la circulation d'une instance du message $Corr$ si elle n'a aucun effet.

Pour optimiser et afin d'arriver à un consensus entre la justesse (précision) et le trafic, nous pouvons proposer de rajouter un compteur $Cdes_i$ au niveau de chaque nœud i. Ce compteur est réinitialisé toutes les $Tdes$ unités de temps à une certaine valeur maximale $CdMax_i$. A chaque réception du message $Corr$ qui n'a aucun effet sur i, $Cdes_i$ est décrémenté. Lorsque $Cdes_i$ atteint la valeur 0, l'instance de $Corr$ est supprimée par i. Seulement, il est très complexe de fixer

les valeurs optimales pour $CdMax$ et $Tdes$. Dans notre cas, nous pouvons dire que nous avons pris le cas le plus simple avec $CdMax = 1$ et $Tdes = \infty$.

Il existe une probabilité minime pour que toutes les instances du message $Corr$ soient détruites ou perdues. En effet, le message est à chaque réception dupliqué et envoyé vers deux voisins (ligne 16 de la figure 7.9). Mais vu les caractéristiques de l'environnement ad hoc, il y a toujours un petit risque d'extinction du message $Corr$ qui peut se produire au début lorsqu'il y a peu d'instances de ce message qui circulent dans le réseau.

Pour résoudre ce problème, plusieurs possibilités sont envisageables. Nous pouvons proposer la régénération automatique de ce message par un nœud donné. Par exemple, celui qui ne l'a jamais reçu ou qui ne l'a pas reçu pendant une longue période peut générer une nouvelle instance de ce message de la même manière que le nœud initiateur dans la tâche précédente (lignes 4 et 5 de la figure 7.10). Le temps d'attente de ce message nécessite la gestion d'un nouveau timeout dont le seuil est difficile à fixer. Seulement, cette solution risque de faire augmenter considérablement le trafic et par conséquent, influe négativement sur la consommation de l'énergie et de la bande passante. Pour cet effet, nous avons opté pour la solution d'éviter de gérer le régénération du message $Corr$.

Il est à noter que si la correction n'est pas effectuée en raison de la rareté ou la disparition des instances du message $Corr$ circulant dans le réseau, la précision de $AFDAN$ va diminuer et le protocole se comporte dans le pire des cas légèrement mieux qu'un simple protocole de détection de pannes basé heartbeat.

Par ailleurs, quand i décide de propager l'instance du message $Corr(path)$ reçu, il sélectionne comme deux de ses voisins. Il prend en considération la nouvelle structure *path* après mise à jour (voir procédure $update\_path(p)$ définie sur les lignes 9 à 17 de la figure 7.10 et appelée à la ligne 13 de la figure 7.9).

Il faut noter que l'ordre d'insertion des tuples (les triplets : identité, nombre de heartbeats et état) dans la structure *path* est important. En effet, il désigne l'historique du message. A savoir le premier tuple inséré concerne le nœud le plus ancien par lequel est passé le message. Par contre, le dernier tuple inséré concerne le nœud le plus récent par lequel est passé le message.

Exemple 1 *Par exemple, soit le nœud* 1 *l'initiateur du processus de correction. Initialement, il transmet le message* $Corr(1,\ H_1[1],\ E_1[1])$ *à deux de ses voisins qu'il sélectionne aléatoirement : par exemple,* 3 *et* 5*. Lorsque le nœud* 3 *reçoit le message, il rajoute un tuple qui le concerne dans la structure path comme*

suit : $Corr($ $(1,\ H_1[1],\ E_1[1]),\ (3,\ H_3[3],\ E_3[3])$ $)$. *Par la suite, il sélectionne aléatoirement deux autres nœuds parmi ses voisins qui ne sont pas cités dans path. Il leur transmet une nouvelle instance du message* $Corr$. *Si* $\{1, 5, 8\}$ *sont les voisins de* 3*, ce dernier peut envoyer le message* $Corr$ *uniquement aux nœuds* 5 *et* 8*, car* 1 *existe dans path reçu par* 3.

De cette manière, nous visons à propager le messages dans le réseau en minimisant le trafic et en même temps prévenir la situation de famine.

Cependant, si tous les voisins de i sont cités dans *path*, alors il sélectionne deux parmi les voisins cités en début de la structure *path*. C'est à dire parmi les plus anciens ayant reçu le message de correction.

Par la suite, en vue de réduire la quantité de données embarquée dans le message $Corr$, le nœud i vérifie si la structure *path* reçue de la part de j contient son état (ligne 10, figure 7.10). Ensuite, il supprime tous les tuples qui ont été insérées avant son propre tuple dans *path* (lignes 11 et 12 de la figure 7.10). En effet, ces tuples ont déjà été traités lorsque i a reçu le message la dernière fois, et il existe une forte probabilité que cette information soit déjà connue par ses voisins actuels. Par conséquent, il n'est pas nécessaire d'encombrer le message avec des données qui pourraient être inutile pour le processus.

Enfin, le nœud i supprime son propre tuple, s'il existe déjà dans *path* (ligne 13, figure 7.10) et l'insère à la fin de la liste (ligne 15, figure 7.9), car il est le dernier nœud ayant reçu cette instance du message $Corr$. Puis il envoie la nouvelle instance $Corr(path)$ aux voisins sélectionnés comme l'indique la ligne 16 de la figure 7.9.

Composant Gestion des Déconnexions

La gestion des déconnexions est constituée des deux tâches décrites ci-dessous et qui sont exécutées par chaque nœud i.

D.Tâche1. Mise à jour de la Connectivité : Cette tâche est décrite par l'algorithme défini sur la figure 7.11.

Dès qu'un nœud i est sur le point de se déconnecter, il incrémente son battement de cœur local (ligne 2) et change son état local à $Disc$ (ligne 3) puis diffuse dans son voisinage un message de déconnexion $disc(i,\ H_i[i])$ (ligne 4).

Quand un nœud se reconnecte, il incrémente encore son battement de cœur local (ligne 7) et remet son état local à actif (ligne 8) puis diffuse un message

```
1   init () {
2     H_i[i] = 0;  Part_i = Neigh_i;
3     if (i = = initiator) {
4       Path = (i, H_i[i], E_i[i]);
5       Broadcast  Corr (path) to  Neigh_i;
6     }
7     for all (q ∈ Neigh_i) { H_i[q]=0; FTO_i[q]= STO; }
8   }
9   update_path(p) {
10    if (∃ (x, y, z) ∈ p / x = = i) {
11      for all ( (q, h, s) ∈ p / (q, h, s) appear before (i, y, z) )
12        delete (q, h, s) from p;
13      delete (i, y, z) from p;
14    }
15    for all ( (q, h, s) ∈ p )    update (q, h, s);
16    return p;
17  }
18  update(b, h, t) {
19    if (b ∈ Part_i) u=0;
20    else {    u=1;  Part_i = Part_i ∪ {b}; }
21    if ((u = = 0) and (E_i[b] ≠ t)) {
22      if (H_i[b] < h)       u = 1;
23      else {
24        if (H_i[b] > h)     u = -1;
25        else if (E_i[b]= = 'Fsusp') u = -1;
26             else if (t= = 'Fsusp') u = 1;
27      }
28    } else {
29      if (H_i[b] < h) H_i[b] = h;
30      else h = H_i[b];
31    }
32    if (u= =1)  { E_i[b] = t; H_i[b] = h; }
33    if (u= =-1) { t = E_i[b]; h = H_i[b]; }
34    if ((E_i[b]= = 'Disc') or (E_i[b]= = 'Disc')) delete Ddl_i[b];
35    if (u ≠ 0) prop_i=true;
36  }
```

FIGURE 7.10 – AFDAN : Les procédures.

de re-connexion $Rec(i,\ H_i[i])$ dans son voisinage pour les informer de son retour (ligne 9).

D.Tâche2. Gestion du Message de Connectivité : Lors de la réception par le nœud i d'un message de re-connexion ou de déconnexion envoyé par le nœud j, il effectue ce qui suit comme l'indique la figure 7.12.

— Si i reçoit le message $Disc(j,\ H_j[j])$, alors il met à jour la valeur locale du battement de cœur de j par celle qui a été reçue (ligne 4). Il le considère comme étant déconnecté en lui affectant son nouvel état (ligne 5). Il bloque le temporisateur associé à j en mettant le délai d'attente à l'infini

```
    // D.Task 1
1   When i decides to disconnect {   // voluntary or due to connectivity mechanism
2     H_i[i]++;
3     E_i[i] = 'Disc';
4     Broadcast Disc (i, H_i[i]) to Neigh_i;
5   }
6   When i decides to reconnect to the network {
7     H_i[i]++
8     E_i[i] = 'Active';
9     Broadcast Rec (i, H_i[i]) to Neigh_i;
10  }
```

FIGURE 7.11 – AFDAN : Composant gestion des déconnexions - Tâche 1.

(lignes 2 et 3).

— Si i reçoit le message $Rec(j,\ H_j[j])$, alors il met à jour la valeur locale du battement de cœur de j par celle qui a été reçue (ligne 10). Il remet son état à actif (ligne 11). Un temporisateur est par la suite créé et associé à j par i et la valeur du timeout est réinitialisée (ligne 8).

```
    // D.Task 2
1   When i receives Disc (j, h) from j {
2     FTO_i[j] = ∞;
3     Ddl_i[j] = ∞;
4     H_i[j] = h;
5     E_i[j] = 'Disc';
6   }
7   When i receives Rec (j, h) from j {
8     FTO_i[j]= STO;
9     Ddl_i[j]= current_time () + FTO_i[j];
10    H_i[j] = h;
11    E_i[j] = 'Active';
12  }
```

FIGURE 7.12 – AFDAN : Composant gestion des déconnexions - Tâche 2.

7.6 Protocole FDRAM

Nous avons constaté que la durée de vie d'un message correction dans le réseau est longue dans le protocole AFDAN. De plus, la quantité de données est volumineuse et cause ainsi la surcharge du réseau. Ces deux points ne sont pas en adéquation avec les réseaux ad hoc où la bande passante est limitée. Afin de remédier à ce problème, nous avons proposé une amélioration à travers le protocole $FDRAM$ [9].

Ainsi, la conception de ce Protocole se base sur les mêmes hypothèses et structures de données que celles du protocole $AFDAN$. Par contre, nous proposons de modifier la gestion de la correction pour les raisons citées ci-dessus. En effet l'envoi des messages de correction par un nœud i se fait uniquement lors du changement d'état d'un nœud q, en embarquant dans ce message l'identité de q, son heartbeat $H_i[q]$ et son état $E_i[q]$(suspect, vivant ou déconnecté).

Ainsi, le message de correction $Corr$ est généré par n'importe quel nœud et ne contient pas toute une liste de nœuds et sa propagation s'arrête rapidement.

7.6.1 Structures de données

$FDRAM$ est un protocole distribué. Chaque nœud i utilise ses propres structures de données. Les structures de données sont identiques à celles utilisées par le protocole $AFDAN$ et qui sont définies dans la section 7.5.1. Afin d'éviter les répétitions, nous citons, ci-dessous, sans détails les structures de données du protocole $FDRAM$.

- $Neigh_i$: Il s'agit d'un ensemble d'identificateurs des nœuds voisins du nœud i.
- $Part_i$: Il s'agit d'un ensemble d'identificateurs de nœuds que connait le nœud i.
- H_i : Cette structure est un tableau d'entiers pour sauvegarder les valeurs des battements de cœur (heartbeats).
- E_i : C'est un tableau où chaque élément désigne l'état d'un nœud j tel qu'il est vu par le nœud i et peut prendre une des valeurs suivantes : *Active*, Tsusp, Fsusp ou Disc.
- STO : Il s'agit d'une constante qui désigne une valeur seuil du temporisateur.
- FTO_j : C'est un tableau de timeouts.
- Ddl_j : Il s'agit d'un tableau de dates limites (deadline en anglais).

7.6.2 Les messages

$FDRAM$ est un protocole qui se base sur la technique de battement de cœur. Ainsi, la détection des nœuds défaillant se base sur l'échange de messages. Pour cet effet, on utilise (comme dans $AFDAN$) quatre différents types de messages.

Comparativement à $AFDAN$ et $FDAN$, les mêmes messages sont utilisés

dans $FDRAM$. En effet, les trois premiers sont similaires dans la forme et la manière d'utiliser que ceux de $AFDAN$.

Parmi les différences dans $FDRAM$ qu'on peut signaler, c'est au niveau du quatrième message (C'est à dire le message de correction $Corr$) . Le contenu ainsi que la gestion de ce message ont changé relativement aux deux autres protocoles.

Ci dessous les quatre messages utilisés par notre protocoles :

— $Alive(i,\ H_i[i])$: Il s'agit du message de battement de cœur qui permet à un nœud donné de prouver sa vivacité.

— $Disc(i,\ H_i[i])$: Ce message est diffusé par un nœud i à ses voisins directs avant de se déconnecter.

— $Rec(i,\ H_i[i])$: Ce message est diffusé par un nœud i vers ses voisins dès qu'il se re-connecte au réseau.

— $Corr(i,\ (q,h,e)\ )$: Il s'agit du message de correction qui est envoyé par un nœud i à un autre nœud j pour l'informer sur le nouvel état d'un autre nœud q. Il embarque dans ce message le triplet $(q,h\ ,\ e)$ où q est l'identificateur d'un nœud connu par i ; $h = H_i[q]$ est la valeur du battement de cœur de q telle qu'elle est connue par i et $e = E_i[q]$ est l'état de q selon i. Ce message permet de corriger les fausses suspicions et de mettre à jour les listes de suspects que gèrent les autres nœuds. Il permet aussi de propager l'information sur la défaillance dans tout le réseau. Ainsi, l'information concernant un nœud (suspicion ou correction de fausse suspicion) circule rapidement.

Il faut noter que les déconnexions et re-connexions d'un nœud sont gérés dans $FDRAM$ de la même manière que dans $AFDAN$. Cela est reflété par l'envoi respectivement des messages $Disc$ et Rec aux voisins directs. Ceci permet de faire la distinction entre les nœuds déconnectés de ceux qui peuvent être défaillants.

7.6.3 Description détaillée du protocole

Le protocole $FDRAM$ est constitué de deux composants : le détecteur de pannes et le gestionnaire des déconnexions.

Le composant de détection de pannes

Le détecteur de pannes est constitué de quatre tâches concurrentes exécutées par chaque nœud i impliqué dans le calcul distribué.

```
    // F.Task 2
1   When i receives Alive(j, h) from j {
2     if (FTO_i[j] = = NULL)   FTO_i[j]=STO;
3     H_i[j] = h;
4     if (E_i[j]= ='Active')  FTO_i[j] = FTO_i[j]+( current_Time() - Ddl_i[j])/2;
5     else {
6       if (E_i[j] = = 'Tsusp') or (E_i[j] = = 'Fsusp')) {
7         FTO_i[j] = FTO_i[j]+1;
8         if (E_i[j] = = 'Fsusp') Broadcast Corr (j, H_i[j], E_i[j]) to  Neigh_i;
9       } else {   // i don't know j
10            Part_i= Part_i ∪ {j};
11            FTO_i[j] = STO;
12    } }
13    E_i[j]='Active';
14    Ddl_i[j] = current_time() + FTO_i[j];}
15  }
```

FIGURE 7.13 – FDRAM : Composant gestion des défaillances - Tâche 2.

Dans l'ensemble, nous avons gardé le même fonctionnement que dans $AFDAN$ à l'exception des instructions supplémentaires dans les tâche 2 et 3 pour gérer les messages de correction $Corr$. La tâche 4 introduit une procédure différente pour un meilleur traitement des fausses suspicions.

En effet, dans la perspective de trouver un compromis entre la précision et la complétude tout en respectant les contraintes de mobilité et particulièrement la consommation des ressources, nous proposons dans $FDRAM$ un mécanisme différent afin de traiter les fausses suspicions.

F.Tâche1. Envoi du battement de cœur : Cette tâche se déroule de la même manière que la tâche $F.Tâche1$ du protocole $AFDAN$ et comme le précise l'algoritthme de la figure 7.6. En effet, un nœud i diffuse un message de vivacité $Alive(i,\ H_i[i])$ vers tous les $(Neigh_i)$ périodiquement après avoir incrémenté sa valeur de battement de cœur, $H_i[i]$.

F.Tâche2. Réception du battement de cœur : L'algorithme associé à cette tâche est décrit dans la figure 7.13. A chaque fois que le nœud i reçoit de la part du nœud j le message $Alive(j,\ H_j[j])$, le battement de cœur correspondant à j sauvegardé chez i lui est affecté la valeur reçue (ligne 3). Si ce message est reçu pour la première fois (c'est à dire, de la part d'un nœud qui est inconnu pour i) alors j est rajouté à l'ensemble des nœuds participants connus par i (ligne 10) et le timer associé $(FTO_i[j])$ est créé avec le délai mis à la première valeur comme l'indique la ligne 11.

Pour les prochains messages de vivacité envoyés par j et reçus par i (C'est à

dire que i connaît j), le protocole doit se comporter selon les cas suivants :

— Si j est considéré comme actif par le nœud i ($E_i[j] = Active$), alors la valeur du premier temporisateur associé à i ($FTO_i[j]$) est décrémentée (ligne 4). En effet, le nœud j est vu par i comme étant rapide dans la réponse. Ainsi, il est préférable d'ajuster le délai en conséquence pour la prochaine période.

— Comme le précise la ligne 6, si j est dans la liste temporaire de suspects (c'est à dire $E_i[j] = Tsusp$) ou j est dans la liste finale de suspects ($E_i[j] = Fsusp$) (nous observons ici la détection d'un cas de fausse suspicion) alors la valeur du premier timeout associé à j ($FTO_i[j]$) est incrémentée (ligne 7). Ceci signifie que ce nœud réagit lentement. De ce fait le premier timeout doit être ajusté en conséquence pour la prochaine période afin d'éviter de le suspecter une nouvelle fois.

  La réception tardive du battement de cœur de j permet à i de corriger localement une fausse suspicion (dans le cas où j est dans la liste de suspects finale). De ce fait, l'état du nœud j est mis à jour chez i.

  Par ailleurs, i doit informer les autres nœuds de cette fausse suspicion pour une éventuelle correction. En effet, cette fausse suspicion s'est peut être propagée précédemment dans le réseau à cause du message de correction généré lors de l'expiration du second délai (voir ci-dessous *F.Tâche.3*). Ainsi, i diffuse le message $Corr(i,\ (j, H_i[j], Active)\ )$ à ses voisins (voir ligne 8).

Ensuite, dans toutes les situations, le protocole met à jour l'état de j à $actif$ (ligne 13) et fixe la date du prochain délai associé à j selon la valeur du timeout ($FTO_i[j]$), en attendant de recevoir le prochain message de vivacité (ligne 14).

F.Tâche3. Expiration du délai : Les traitements liés à cette tâche sont décrits dans l'algorithme défini dans la figure 7.14. Chaque nœud i gère des temporisateurs liés à ses voisins. Quand le délai pour un nœud voisin j est atteint sans recevoir de message de vivacité de sa part peut avoir plusieurs significations. On peut l'interpréter généralement par soit le la lenteur du nœud j soit par une réelle défaillance de j.

Dans ce cas, le protocole réalise des traitements spécifiques selon les cas suivants :

— Si le nœud j est vu comme étant actif par i ($E_i[j] = Active$ comme l'indique la ligne 6), alors j est mis dans la liste temporaire de suspects

```
    // F.Task 3
1   When Ddl_i[j] reached {
2     if (E_i[j] = = 'Tsusp') {
3       E_i[j] = 'Fsusp';
4       delete Ddl_i[j];
5       Broadcast Corr (j, H_i[j], E_i[j]) to  Neigh_i; //propagate correction
6     } else {
7       E_i[j] = 'Tsusp';
8       Ddl_i[j] = current_time() + STO;
9   } }
```

FIGURE 7.14 – FDRAM : Composant gestion des défaillances - Tâche 3.

de i (voir ligne 7). Par la suite, i accorde une extension de délai à j en espérant recevoir son battement de cœur avant de confirmer ou infirmer sa suspicion de défaillance (ligne 8). En d'autres termes, on veut détecter le vrai état de j dans un délai n'excédant pas STO unités de temps. En effet, le vrai état de j peut être décelé à travers : la réception de son battement de cœur durant l'extension de la période (le nœud est lent) ou à travers l'expiration du second délai (le nœud est probablement défaillant) ou en recevant une information le concernant via un message de correction (par exemple dans le cas où le nœud s'est éloigné du voisinage de i).

— Si le nœud j est déjà dans la liste temporaire de suspects de i ($E_j[i] = Tsusp$, ligne 2), ce qui signifie que l'extension du délai a été consommée sans recevoir de message de vivacité de la part de j. Ainsi, ce dernier est retiré de la liste temporaire et mis dans la liste finale de suspects (ligne 3) et son temporisateur $Ddl_j[i]$ est désactivé (ligne 4).
Par la suite, le nœud i diffuse dans son voisinage (ligne 5) un message de correction $Corr(i,\ (j, H_i[j], Fsup)\ )$ pour informer ses voisins que le nœud j est suspecté de défaillance.

Comme dans $FDAN$ et $AFDAN$, il faut noter que l'extension du délai permet au nœud j de gérer avec précision les fausses suspicions. La valeur de STO doit être prise avec une grande précaution. Ceci a été discuté dans la section 7.5.3, lors de la description de la troisième tâche du détecteur de pannes ($F.Tâche3$).

F.Tâche4. Réception du message de correction : La gestion de la correction des fausses suspicions est réalisée en grande partie par les traitements faits lors la réception du message $Corr$. Chaque fois qu'un nœud i reçoit un message $Corr(i,\ (b, h, e)\ )$ de j, il prend la décision s'il doit mettre à jour les informations du nœud b ou non. La décision est prise en fonction des valeurs

```
    // F.Task 4
1   When i receives Corr(b, h, t) from j {
2     propi = test(b, h, t);
3     if (propi = = 1) {
4       update(b, h, t);
5       Broadcast Corr(b, Hi[b], Ei[b]) to Neighi; //propagate correction
6     }
7     if (propi = = -1) {
8       Send Corr(b, Hi[b], Ei[b]) to j; //ask j to correct its information
9   } }
```

FIGURE 7.15 – FDRAM : Composant gestion des défaillances - Tâche 4.

reçues (h qui est battement de cœur de b et e son état). Les vérifications sont effectuées à travers l'appel à la fonction *test* sur la ligne 2 de l'algorithme défini dans la figure 7.15. Cette fonction retourne une valeur (sauvegardée dans la variable $prop_i$) qui correspond à la décision qui est prise. Si $prop_i = 1$ alors on met à jour l'information locale. Si $prop_i = -1$ alors on corrige la fausse information transmise par j afin de minimiser la durée de toute fausse suspicion dans le réseau.

Nous décrivons ci-dessous traitements effectués par la fonction *test* qui est définie dans la figure 7.16 :

— Si $b \notin Part_i$ alors le nœud i ajoute le nœud b à ses connaissances puis copie son état et son battement de cœur localement comme l'indique la ligne 8.

— Si $b \in Part_i$ alors le nœud i vérifie quel est l'état qui est le plus récent entre le local et celui reçu de la part de i comme suit :

Si $H_i[b] > h$ (ligne 12) qui signifie que i possède l'état le plus récent de b (du moins par rapport à j) alors le message n'a aucun effet sur i. On a mis la variable u à -1 pour représenter ce cas. Par ailleurs, le nœud i doit corriger l'information détenue par j. Dans cette perspective, il envoie à j le message $Corr(i,\ (b, H_i[b], E_i[b])\ )$ seulement quand $E_i[p] \neq e$ (ligne 9). L'envoi du message est symbolisé par le renvoi de -1 par la fonction *test*, ce qui correspond à affecter cette valeur à la variable $prop_i$, ce qui force le nœud à exécuter les lignes 7 et 8 de l'algorithme présenté dans la figure 7.15.

Si $H_i[b] < h$ (ligne 10) qui signifie que j possède l'état le plus récent de b alors j doit mettre à jour l'état de b localement. Dans l'algorithme, ceci est effectué à travers la mise de la valeur de la variable u à 1 (ligne 10). Cette

```
1   init () {
2     H_i[i] = 0;  Part_i = Neigh_i;
3     for all (q ∈ Neigh_i) {
4       H_i[q]=0;
5       FTO_i[q]=STO;
6   } }
7   test(b, h, t) {
8     if (b ∈ Part_i) u=0;
9     else {  u=1; Part_i = Part_i ∪ {b}; H_i[b]=h; }
10    if ((u = = 0) and (E_i[b] ≠ t)) {
11      if (H_i[b] < h)       u=1;
12      else {
13        if (H_i[b] > h)     u=-1;
14        else if (E_i[b]= = 'Fsusp') u=-1;
15             else if (t= = 'Fsusp') u=1;
16    } }
17    if ((u = = 0) and (H_i[b]< h)) H_i[b]=h;
18    return u;
19  }
20  update(b, h, t) {
21    E_i[b] = t;
22    H_i[b] = h;
23    if ((E_i[b]= = 'Disc') or (E_i[b]= = 'Disc')) delete Ddl_i[b];
24  }
```

FIGURE 7.16 – FDRAM : Les procédures.

variable est renvoyée par la fonction *test* dans la variable $prop_i$ (ligne 2, figure 7.15). Ceci déclenche (ligne 3 7.15)l'appel à la procédure *update* sur la ligne 4 (figure 7.15). Par la suite, il doit répercuter ses changements sur ses voisins à l'exception de i en envoyant le message de correction $Corr(i,\ (b, H_i[b], E_i[b]))$ (ligne 5, figure7.15) seulement quand $E_i[p] \neq e$ (ligne 9, 7.16).

La correction des fausses suspicions dans $FDRAM$ est réalisée à travers la transmission d'instances du message $Corr$ quand c'est nécessaire. Chaque instance de ce message collecte la plus récente information concernant l'état d'un nœud donné et puis la propage dans les autres parties du réseau à travers ses voisins puis leurs voisins et ainsi de suite (saut par saut) tant qu'il a un impact sur les nœuds visités. En effet, si le message $Corr$ n'a aucune incidence sur le nœud i, c'est à dire quand l'état de b dans i est le même que celui qui est dans j (l'émetteur du message $Corr$), alors le nœud i arrête la propagation de ce message. Il s'agit du cas où $prop_i = 0$. Cette variable est mise à 0 grâce à la fonction *test* (ligne 7 de la figure 7.16).

Composant gestion des déconnexions

La gestion des déconnexions consiste en deux tâches qui sont exécutées par chaque nœud i. Il s'agit des mêmes tâches que celles définies dans le protocole $AFDAN$ (voir *D.Tâche1* et *D.Tâche2* de la section 7.5.3) comme le précisent les algorithmes définis dans les figures 7.11 et 7.12.

D.Tâche1. Mise à jour de la connectivité : Lorsqu'un nœud j est sur le point de se déconnecter ou de se reconnecter, il incrémente son battement de cœur local et met à jour son état local et diffuse un message de déconnexion $Disc(i,\ H_i[i])$ ou de re-connexion $Rec(i,\ H_i[i])$ (selon le cas) dans son voisinage.

D.tâche2. Gestion du message de connectivité : Quand le nœud j reçoit un message de déconnexion ou de re-connexion, celui-ci procède comme suit :

Si i reçoit le message $Disc(j,\ H_j[j])$ ou $Rec(j,\ H_j[j])$ de la part de j, alors il met à jour la valeur du battement de cœur local associé à i par celle qu'il vient de recevoir et fait de même pour l'état local de j, $E_j[i]$ à $Disc$ ou $Active$ selon le message reçu. A la fin, il bloque le temporisateur associé à j dans le cas d'une déconnexion

afin de ne plus attendre de message de vivacité de la part i durant sa période de déconnexion. Ceci lui évite de le suspecter pendant cette période.

Pour terminer, il lui crée un temporisateur et initialise le premier timeout ($FTO_j[i] = STO$).

7.7 Discussion

Dans les trois protocoles ($FDAN$, $AFDAN$ et $FDRAM$), la détection est principalement basée sur la technique des battements de cœur (heartbeat). Dans ces trois protocoles, il est demandé à chaque nœud de diffuser périodiquement (tous les β unités de temps) un message de *vivacité* (appelés *Alive* dans $AFDAN$ et $FDRAM$ et *I_Alive* dans $FDAN$) dans son voisinage. Cette approche est plus pratique dans le contexte des MANETs. En effet, elle génère moins de trafic que l'approche pinging, comme nous l'avons expliqué auparavant.

La correction des fausses suspicions est réalisée à travers l'échange des messages de *correction* (appelés *Corr* dans $AFDAN$ et $FDRAM$ et *need_correction* dans $FDAN$). Ces messages servent aussi pour la propagation de l'information (détection et correction de fausses suspicions) dans le réseau.

Contrairement à $FDAN$, , les timeouts de $FDRAM$ et $AFDAN$ associés aux messages de vivacité sont utilisé comme première étape, afin de distinguer les nœuds lents des nœuds rapides ou ordinaires. Dans la seconde étape, d'autres mécanismes sont introduits pour corriger les fausses suspicions et par la même de distinguer les nœuds lents de ceux qui sont défaillants. Par contre, dans $FDAN$, les temporisateurs sont aussi associés au message de correction qui est lui même un battement de cœur particulier.

En effet, comme le système distribué est par définition asynchrone, l'utilisation de temporisateurs suppose théoriquement qu'il y a une sorte de synchronisme (système partiellement synchrone). Dans $FDAN$, ceci affecte négativement la propriété justesse (précision) du protocole, vu que c'est le seul et unique mécanisme utilisé pour suspecter les nœuds défaillants. Effectivement, dans $FDAN$ le message de correction *need_correction* n'est qu'un heartbeat particulier qui est envoyé périodiquement par un nœuds (tous les K heartbeats).

Dans $AFDAN$, divers instances du message $Corr$ circulent dans le réseau afin de collecter puis propager le plus récent état des différents nœuds, sans être dépendant des timeouts. Un tel mécanisme rend possible l'amélioration de la justesse de la procédure de détection comme vont le confirmer par la suite les résultats de simulations.

Par contre, ceci peut induire en retour un trafic plus important ce qui n'est pas en adéquation [3] avec le contexte des MANETs. En conséquence, dans le but de résoudre ce problème, des mécanismes supplémentaires pour contrôler le nombre de messages $Corr$ ainsi que leur propagation dans le réseau.

En effet, une instance du message de correction ($Corr$) est générée initialement par un nœud prédéfini puis envoyée au plus à deux de ses voisins. Ces derniers réalisent le même processus en recevant une instance de ce message et ainsi de suite... Le message circule en continue dans le réseau tant qu'il a un impact sur l'information locale du nœud visité. Dans le cas contraire, l'instance est détruite.

Ces contrôles n'empêchent pas de dire que le coût supplémentaire engendré par les messages de correction peut être non borné dans $AFDAN$. En effet, ces messages sont plus volumineux, même si leurs instances sont moins nombreuses dans le réseau.

Pour palier à ce problème, dans $FDRAM$, le message de correction n'est

3. Ceci peut être préjudiciable dans le contexte des MANETs.

TABLE 7.1 – Les performances théoriques de nos protocoles de détection de pannes.

Protocole	$FDAN$	$AFDAN$	$FDRAM$
Technique	Heartbeat	Heartbeat	Heartbeat
Justesse	Finalement forte	Finalement forte	Finalement forte
Complétude	Forte	Faible	Forte
Classe	$\diamondsuit P$	$\diamondsuit Q$	$\diamondsuit P$
Gestion des déconnexions	Simple	Simple	Simple
Trafic	Peu de messages, Message de correction volumineux	Trop de trafic (plusieurs instances *corr* circulant, Message de correction volumineux	Peu

généré que s'il y a nécessité et il est allégé. En effet, le message de correction ne concerne que l'état d'un seul unique nœud à la différence avec la structure *path* embarquée dans les messages de correction de $AFDAN$.

Par conséquent, $AFDAN$ peut signaler une meilleure précision (en termes de pourcentage de détection), que $FDRAM$. Mais d'autre part, le temps moyen nécessaire pour corriger les fausses suspicions peut être plus long dans $AFDAN$ par rapport à $FDRAM$.

De plus, il n'est pas sûr dans $AFDAN$ que l'information sur la détection de panne soit propagée dans tout le réseau à cause de la gestion de la propagation du message de correction. Ainsi, $AFDAN$ assure une complétude faible, ce qui n'est pas le cas pour $FDRAM$.

Finalement, les déconnexions et re-connexions d'un nœud sont exprimés par l'envoi respectivement des messages de *déconnexion* et de *reconnexion* aux voisins directs. Ceci permet de distinguer entre une déconnexion et une défaillance d'un nœud. Les trois protocoles utilisent le même principe.

Le tableau 7.1 récapitule cette analyse.

7.8 Analyse des Performances

Nous avons utilisé le simulateur de réseaux NS2 [88] pour tester et analyser nos protocoles dans la perspective de comparer leurs performances. Nous avons

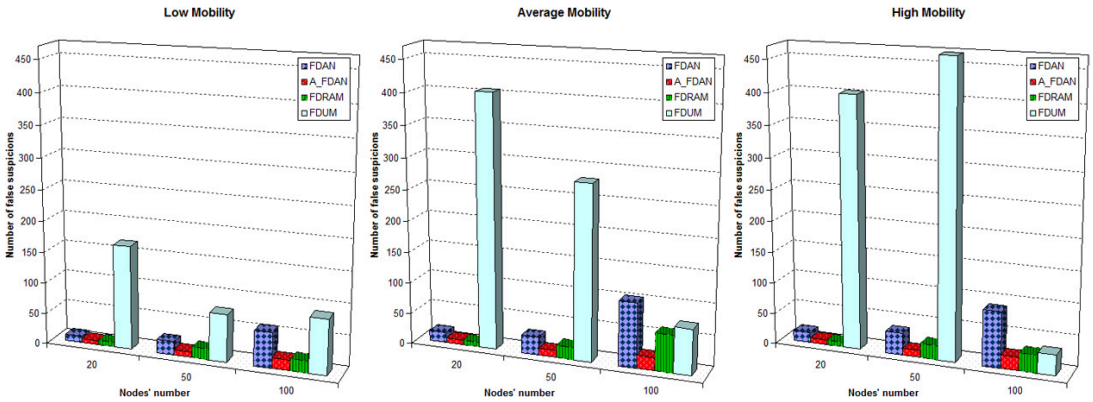

FIGURE 7.17 – Nombre de fausses suspicions par rapport au nombre de nœuds.

implémenté nos trois protocoles $FDAN$ [7], $AFDAN$ [8] et $FDRAM$ [9]. Nous avons aussi implémenté le protocole $FDUM$ [46]. Pour toutes les simulations effectuées, nous avons opté pour le modèle de mobilité Random Way point Model (RWM) [89] qui se rapproche le plus de la réalité. En outre, nous avons considéré les paramètres suivants :

- La surface de déploiement des nœuds a été fixée à : 1000x1000 m^2. Nous avons aussi fixé la portée du signal radio de chaque nœud à 250 m.
- La durée de chaque test est de 100 *secondes* et le nombre de pannes simulées est une (1).

Par ailleurs, nous avons varié les paramètres suivants :

- Nous varions le nombre de nœuds mobiles comme suit : 20, 50 et 100. La variation de ce paramètre nous permet de juger la scalabilité (passage à l'échelle).
- Dans le but de voir l'effet des changement de topologie sur les protocoles de détection de pannes, la mobilité est variée entre : $faible$, $moyenne$ et $forte$. Ce que nous considérons comme mobilité est la variation distances entre les nœuds. En effet, la mobilité n'est pas uniquement le changement de vitesse mais plutôt le changement de position dans le temps relativement aux autres nœuds ce qui engendrera un changement dans la topologie et des déconnexions et re-connexions.

Les résultats des simulations effectués sur les quatre protocoles sont décrits ci-dessous :

Le nombre moyen de fausses suspicions : Cette mesure dénote le nombre moyen de fausses suspicions commis par un nœud. Elle permet d'évaluer

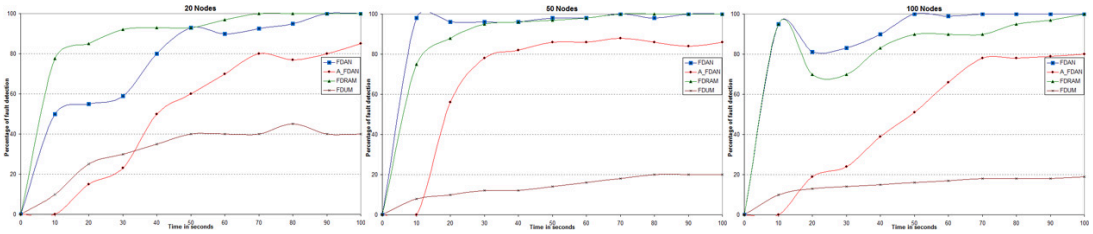

FIGURE 7.18 – Le pourcentage des nœuds ayant détecté la panne.

la précision (justesse) des protocoles.

Dans la figure 7.17 nous représentons le nombre moyen de fausses suspicions en fonction du nombre de nœuds tout en faisant varier la mobilité. Nous constatons que $AFDAN$ donne les meilleures performances. Cependant, ceux de $FDRAM$ sont presque égales. En effet, dans $AFDAN$ et $FDRAM$ la correction se fait plus fréquemment que dans $FDAN$ et $FDUM$. En outre, dans $FDUM$, le protocole ne gère qu'un seul niveau de la liste de suspects. De plus, les déconnexions ne sont pas traitées et peuvent être perçues comme des défaillances.

Enfin, la mobilité semble avoir un grand impact sur les performances $FDUM$, contrairement aux autres protocoles.

Le pourcentage des nœuds ayant détecté la panne : Ce paramètre est calculé puis remis à zéro toutes les secondes pendant le temps de simulation. Il permet d'évaluer la complétude du protocole.

Ce paramètre est évalué en faisant varier le nombre de nœuds et la mobilité tout en simulant la défaillance d'un nœud à la dixième ($10^{ème}$) seconde.

Comme le montre la figure 7.18, les résultats montrent que pour $FDAN$ et $FDRAM$; plus le nombre de nœuds est élevé, la complétude de la détection est rapide et ce, contrairement à $FDUM$. En outre, nous constatons que les pourcentages peuvent diminuer avec le temps, cela est dû à la non fiabilité des protocoles de détection de défaillance, en général.

Le trafic moyen par nœud : Cette mesure représente la quantité moyenne de données envoyées par nœud durant le temps de simulation. Elle permet de déterminer quels sont les protocoles qui génèrent le moins de trafic, et donc consommant le moins d'énergie et de bande passante.

Nous avons réalisé ces mesures en faisant varier la mobilité et le nombre de nœuds. Les résultats sont présentés dans la figure 7.19.

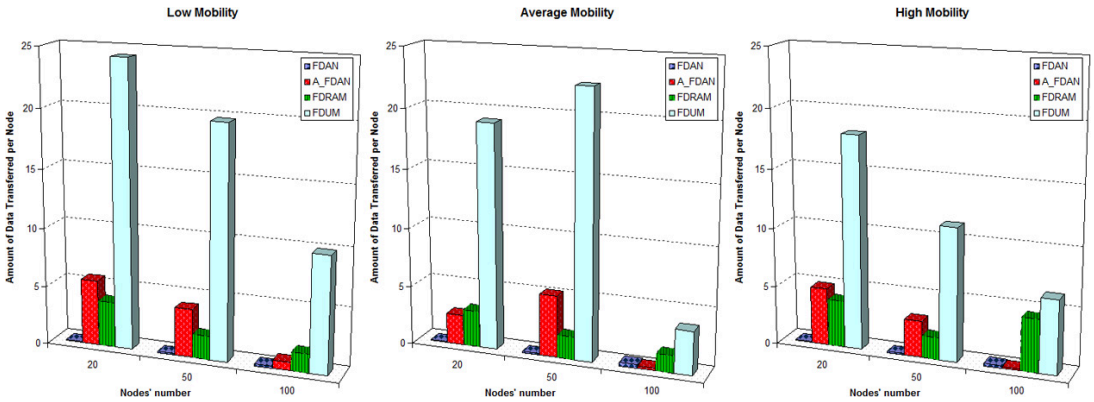

FIGURE 7.19 – Le trafic moyen généré durant la simulation.

Nous constatons que $FDUM$ rapporte les pires résultats en raison de la taille des listes échangées dans les messages générés par la technique adoptée en l'occurrence *Requête/Réponse*.

En ce qui concerne $AFDAN$, nous remarquons que la progression du trafic n'est pas linéaire. En effet, le trafic dans ce protocole dépend principalement de la structure *path* embarquée dans les messages de correction. Outre la durée de vie de ce dernier qui dépend de son incidence sur les nœuds destinataires, qui peuvent augmenter l'overhead de ce message.

D'un autre côté, $FDAN$ surpasse nettement les autres protocoles pour cette mesure. Ceci est expliqué par la quantité de données transférée par ce protocole pendant le processus de correction est très faible.

Par ailleurs, le protocole de $FDRAM$ semble rendre des résultats meilleurs que $AFDAN$ lorsque le nombre de nœuds n'est pas élevé. Cependant, lorsque celui-ci augmente (autour de 100 nœuds), $AFDAN$ surpasse légèrement $FDRAM$.

Enfin, la mobilité ne semble pas avoir d'incidence sur la quantité de trafic induit, quel que soit le protocole considéré.

Le temps moyen pour corriger une fausse suspicion : Cette mesure est liée à la précision du protocole en permettant d'évaluer le temps nécessaire pour un nœud afin de corriger une fausse suspicion.

En effet, un protocole qui est capable de corriger toute fausse suspicion est considéré comme *finalement fort* pour la propriété de justesse.

Dans les graphiques présentés dans la figure 7.20, nous avons réalisé cette mesure pour chaque type de mobilité tout en faisant varier le nombre de

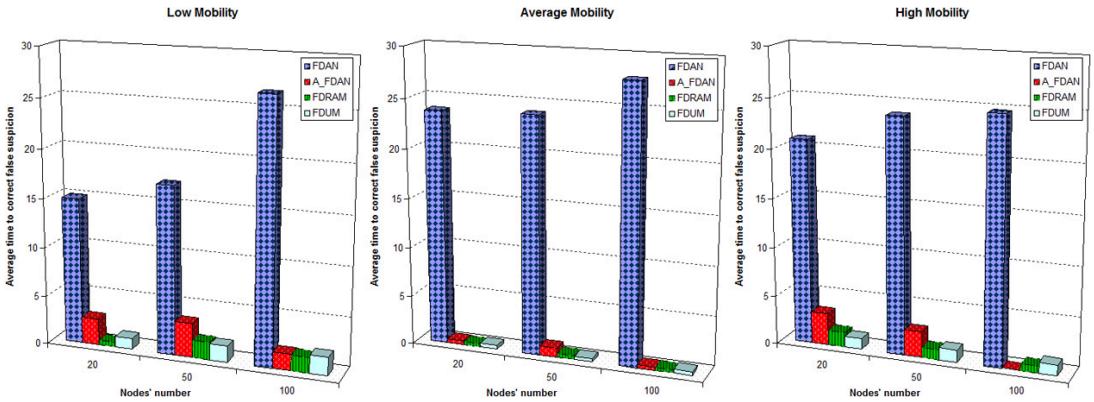

FIGURE 7.20 – Le temps moyen pour corriger une fausse suspicion.

nœuds.

Comme nous pouvons le constater, les protocoles $FDUM$, $AFDAN$ et $FDRAM$ surpassent clairement le protocole $FDAN$. Ceci est dû aux mécanismes de correction asynchrones introduits dans ces protocoles, qui font qu'il soit possible de corriger les fausses suspicions assez rapidement.

Cependant, le protocole $FDRAM$ semble surpasser légèrement le protocole $AFDAN$ car dans $FDRAM$ le message de correction est envoyé instantanément une fois une fausse suspicion est détectée, tandis que, dans $AFDAN$ il faut attendre le passage du message de correction pour propager les informations.

À d'autres égards, nous pouvons voir que la mobilité a un petit effet pour cette mesure.

Les résultats de simulation obtenus dépendent de la technique utilisée par chaque protocole. Cependant, nous remarquons que le protocole $FDRAM$ fournit un meilleur compromis entre la précision, la complétude et la surcharge de trafic que les trois autres protocoles.

En effet, $FDRAM$ gère mieux les messages de correction $Corr$. Au lieu de répercuter l'état des nœuds déjà visités dans le message $Corr$ comme le fait $AFDAN$, il envoie uniquement les informations concernant le nœud objet d'une mise à jour de son état (suspicion ou la correction d'une fausse suspicion). Les simulations montrent que $FDRAM$ garantit une forte complétude et une précision éventuellement forte, tandis que $AFDAN$ assure une faible complétude et une précision finalement forte.

Même si dans $FDRAM$ nous avons un peu plus de fausses suspicions que

dans $AFDAN$ (moins précis), il est à noter qu'il est plus rapide à réagir et à les corriger et à détecter l'apparition d'une vraie défaillance. Par ailleurs, le trafic induit est mieux contrôlée dans $FDRAM$ alors qu'il semble non borné dans $AFDAN$. Enfin, l'augmentation du nombre de nœuds ainsi que la variation de la mobilité ne menacent pas considérablement les performances de $FDRAM$.

7.9 Conclusion

Nous avons présenté dans ce chapitre trois nouveaux protocoles de détection de pannes pour les applications distribuées qui s'exécutent dans un environnement mobile ad hoc. Les protocoles partagent certains principes comme l'utilisation de la technique de battement de cœur qui génère moins de messages et les deux niveaux de listes de suspects qui permet de réduire le nombre de fausses suspicions et de ce fait fait augmenter la précision. Les trois protocoles se basent essentiellement sur la bonne gestion des temporisateurs qui permettent de s'assurer de la vivacité d'un nœud ou de le suspecter de défaillance.

La différence entre les trois protocoles est la gestion du message de correction. Dans le premier protocole, en l'occurrence $FDAN$, le message de correction est un battement de cœur particulier qui est envoyé régulièrement par chaque nœud à ses voisins. Dans $AFDAN$, il s'agit d'un message initié par un nœud particulier, dont les instances circulent dans le réseau en transmettant une structure *path* qui donne des informations sur les états des nœuds qu'a visité ce message. Dans le dernier protocole ($FDRAM$), le message de correction est généré uniquement lorsqu'il y a nécessité (modification de l'état d'un nœud).

Les résultats des simulations ont montré que les trois protocoles donnent de bonnes performances. On peut dire que le protocole $FDRAM$ est globalement meilleur particulièrement en terme de rapidité de correction de fausses suspicion. ceci nous permet d'éviter de lancer des processus de recouvrement inutiles.

Dans le chapitre suivant, nous abordons deux autres protocoles qui complètent notre outil de tolérance aux pannes. Il s'agit du protocole de calcul de points de contrôle (chekpointing) et le protocole de recouvrement par retour arrière (rollback recovery).

Chapitre 8

Protocoles de Checkpointing et Recouvrement

8.1 Introduction

Nous avons cité dans le chapitre 5 une liste non exhaustive des travaux dans ce domaine. A notre connaissance, peu de travaux ont été définis pour s'exécuter dans l'environnement mobile ad hoc. Parmi ces travaux, la majorité sont conçus pour les MANETs hiérarchiques (avec clustérisation). Par conséquent, il est utile de concevoir un protocole dédié aux MANETs plats. Ainsi, des défis supplémentaires sont à relever :

- Le protocole de calcul de points de contrôle (checkpointing) doit générer un état global du système cohérent et qui soit le plus récent possible afin de perdre le minimum possible de calcul en cas de défaillance.
- Le protocole doit être indépendant de toute structure (par exemple le cluster), topologie ou protocole de routage.
- Le protocole doit être indépendant de l'application sous-jacente.
- Le protocole doit fonctionner quel que soit le nombre considéré de nœuds.

Pour répondre à ces enjeux, nous avons proposé dans [10][11] un nouveau protocole de calcul de points de contrôle (checkpointing), appelé $2PACA$ *(Two Phases Algorithm of checkpointing for Ad hoc mobile networks)*. Ce protocole est dédié pour les applications qui se déroulent sur réseaux MANETs plats.

Dans ce chapitre, nous nous intéressons à la conception et la description détaillée du protocole de calcul de points de contrôle ($2PACA$), la mise en place

de la mémoire stable ainsi que le protocole de recouvrement. Une analyse des performances faite suite à des simulations exhaustives effectuées sous la plateforme de simulation réseaux NS [88] est présentée. Nous allons aussi fournir des résultats théoriques qui établissent sa solidité et son exhaustivité.

8.2 Concepts de Base

8.2.1 Hypothèses

Nous considérons, durant le déroulement du calcul de points de contrôle :

— Les canaux de communication sont fiables et aucun message ne peut être perdu ou corrompu (altéré) durant la phase de coordination.

— Il n'y a pas de partitionnement dans le réseau durant la phase de coordination.

8.2.2 Idée de Base

De façon plus concrète, notre protocole consiste en un algorithme de calcul de points de contrôle hybride qui se déroule en deux phases : *la phase Quasi-Synchronous et la phase de Coordination.*

La première phase de l'algorithme permet à chaque nœud de progresser dans le calcul de points de contrôle de façon indépendante. Ainsi, il prend un *un point de contrôle spontané* (*spontaneous checkpoint*) et le sauvegarde localement (voir actions 1.1 et 1.2 de la figure 6.1) Cette solution est plus approprié pour l'environnement mobile et plus particulièrement dans le contexte des MANETs. En effet, il limite la quantité de données à enregistrer dans la mémoire stable, ce qui diminue le nombre de messages envoyée et ainsi permet de réduire la consommation de l'énergie et de la bande passante.

La seconde phase est dédiée à coordination (synchronisation). Un nœud, appelé *coordinateur*, initie cette opération qui consiste à enregistrer un état global cohérent et l'enregistre dans la mémoire stable. Le coordinateur fait circuler la requête de checkpointing à travers ses voisins directs. Ces derniers la propagent, à leur tour, vers leurs voisins. Ainsi de suite, jusqu'à ce que la requête couvre tous les nœuds du réseau. A la fin de cette phase, les points de contrôle locaux les plus récents (un point de contrôle par nœud qui est déjà sauvegardé localement), sont transférés par la suite vers la mémoire stable (SS) pour déterminer

un point de contrôle global cohérent permanent (ligne de recouvrement / the recovery line).

Par ailleurs, le protocole doit ajuster son fonctionnement selon l'environnement et la nature du calcul distribué sous-jacent. Pour ce faire„ deux paramètres doivent être initialisés lors du lancement du protocole de calcul de points de contrôle :

— α : Il représente le nombre maximal de points de contrôle que peut prendre un nœud donné localement avant d'initialiser la coordination.
— T : Il correspond au délai (temps) maximal de la période qu'observe un nœud donné entre deux point de contrôle spontanés consécutives.

Une discussion sur l'intérêt et les effets de ces deux paramètres est présentée plus loin dans ce chapitre.

8.3 Description du protocole 2PACA

Nous décrivons dans cette section le protocole de checkpointing proposé, appelé 2PACA(Two Phase Algorithm for Checkpointing for Ad hoc Networks). Les procedures principales de ce dernier sont décrites dans la figure 8.1. Comme nous l'avons expliqué ci-dessus, le protocole fonctionne en deux phases : la première phase, appelé $quasi-synchrone$ ($quasi-synchronous$), permet à chaque nœud de collecter ses points de contrôle de façon indépendante des autres nœuds et les sauvegarde dans sa mémoire locale (par exemple son disque dur).

La seconde phase consiste à réaliser une coordination entre les nœuds participant au calcul dans la perspective de transférer uniquement les checkpoints utiles vers la *mémoire stable* (*stable storage*). Ainsi, cette phase permet de déterminer un état global cohérent assez récent qui peut être exploité directement par le processus de recouvrement après occurrence d'une panne. Nous donnons ci-dessous les détails de fonctionnement de chacune des deux phases.

8.3.1 Phase 1 (Quasi-Synchrone)

L'algorithme exploite le concept déjà présenté dans [78], qui associe à chaque nœud i deux variables entières : Sn_i et $Next_i$ qui sont initialisées à 0 (ligne 6 de la figure 8.1). De façon plus concrète :

```
1   init () {
2      Const Alpha;
3            Level; // is a min value of combination of energy and connectivity
4      connectivity_i; //energy and signal quality, managed by the physical layer
5      neighbour_i; //List of neighbours of i, managed by the physical layer
6      Sn_i=0 ; Next_i=0 ; Cp_i=0;
7      Coord_i=Null; CP_state_i=false;
8      List_i=∅;
9      Weight_i=0; //Float value between 0 and 1
10     Connect_i=1; //0 disconnected, 1 connected, 2 reconnect tentative
11  }
12  CP_spontaneous() {
13     Next_i++;
14     if (Next_i > Sn_i) {
15        cn =local_CP(Next_i);
16        if ((cn == 0) and (not(CP_state_i))){
17           Coordination(1, NULL);
18        }
19     }
20  }
21  local_CP(n) {
22     k= n % Alpha;
23     C_i[k]=local_state_of_i //take local checkpoint C
24     Sn_i=n;
25     return k;
26  }
27  Coordination(w, sender) {
28     CP_state_i=true;
29     weight_i=w;
30     if ((connectivity_i>Level)and(Connect_i==1)) Coord_i=i;
31     else Coord_i=NULL;
32     for (all p in (neighbour_i - {sender, Coord_i})) {
33        weight_i=weight_i/2;
34        send CP_request(Sn_i,Coord_i,weight_i) to p;
35     }
36     if (Connect_i==1) Save C_i[0] as tentative_checkpoint in SS;
37     else Save C_i[0] as disconnect_checkpoint in SS;
38  }
```

FIGURE 8.1 – Description du fonctionnement de $2PACA$.

— Sn_i est le numéro séquentiel du dernier point de contrôle pris localement par le nœud i.
— $Next_i$ est une variable compteur qui est incrémentée chaque T unités de temps permettant de décider à quel moment doit être pris un point de contrôle *spontané*.

Ces informations sont maintenues localement par chaque nœud. Néanmoins, afin d'assurer une progression cohérente du processus de calcul de points de contrôle, chaque nœud doit embarquer (piggyback) son numéro de séquence (Sn_i) dans les messages de calcul envoyés vers les autres nœuds. Chaque nœud peut cependant ajuster sa progression durant cette phase, en comparant la valeur de son numéro de sequence avec celle des autres nœuds. Il s'agit d'une synchronisation implicite. C'est pour cette raison qu'on a appelé cette phase $quasi - synchrone$.

Durant cette phase, nous avons deux types de points de contrôle locaux :

```
    // Task 1: Spontaneous and forced checkpoints management
1   Every T units of time {
2      CP_spontaneous();
3   }
4   When i sends a computation message m to j{
5      m.Sn = Sn_i;
6      Send m to j;
7   }
8   When i receives a computation message m from j{
9      if (Connect_i==1) {
10        if (m.Sn>Sn_i) local_CP(m.Sn);
11        process(m);
12     }
13     else queue(m);
14  }
```

FIGURE 8.2 – Tâche 1 : Gestion des points de contrôle spontanés et forcés.

Point de contrôle spontané

Chaque T unités de temps, le nœud i incrémente la variable $Next_i$. Si la valeur de $Next_i$ est plus grande que celle de la variable Sn_i, alors il prend un point de contrôle spontané. (lignes 13 à 15, figure 8.1).

Dans le cas contraire, nous pouvons déduire que le nœud i a déjà pris un point de contrôle forcé durant la dernière période T et par conséquent il est inutile de prendre un checkpoint spontané.

Point de contrôle forcé

Le nœud i doit prendre un point de contrôle forcé dans les deux cas suivants :

1. Lorsqu'il reçoit une valeur de numéro de séquence (Sn embarqué dans le message de calcul), qui est plus grande que la valeur locale de son numéro de séquence Sn_i. (comme le montre la figure 8.2, ligne 10).
2. A la réception d'une requête de coordination $CP\_Request$ avec une valeur de Sn plus grande que la valeur locale Sn_i. Cette requête est envoyé par un nœud appelé coordinateur vers les autres nœuds, leur demandant d'achever la première phase par la prise d'un point de contrôle forcé (si nécessaire) et de passer à la phase de coordination. ceci est décrit dans la figure 8.4, lignes 2 et 3.

Durant cette phase (quasi-synchrone), chaque nœud i prend un checkpoint et le sauvegarde localement. (voir ligne 23 de la procédure $local\_CP$ dans la figure 8.1)

Le nœud i maintient un temporisateur qui est initialisé à T. Lorsque le délai expire, il prend un *point de contrôle spontané* (lignes 1 et 2 de la tâche 1 dans la figure 8.2) à condition qu'il n'a pas été contraint pendant l'intervalle de temps de prendre un *point de contrôle forcé*. L'exemple que montre la figure 8.3 décrit les deux types de points de contrôle.

Exemple 2 *Nous avons trois nœuds : MH_1, MH_2 et MH_3 qui échangent des messages. Les points de contrôle spontanés sont pris périodiquement et sont représentés par le symbole '['. Les points de contrôle forcés sont représentés par le symbole '$\lfloor_*$'. Les numéros de séquence correspondant aux checkpoints sont indiqués par Sn. La valeur de $Next$ est incrémentée chaque T unités de temps. Lorsque, MH_1 et MH_3 reçoivent respectivement les messages m_1, m_2 et m_3, ils ne doivent pas prendre de point de contrôle forcé. Le message m_4 envoyé par MH_2, dans lequel est embarqué le numéro de séquence $(m_4.Sn = 5) > (Sn_1 = 4)$, force MH_1 à prendre un point de contrôle forcé avant de traiter le message. Durant la période suivante (Lorsque $Next_1 = 5$), MH_1 est déchargé de la prise du point de contrôle spontané.*

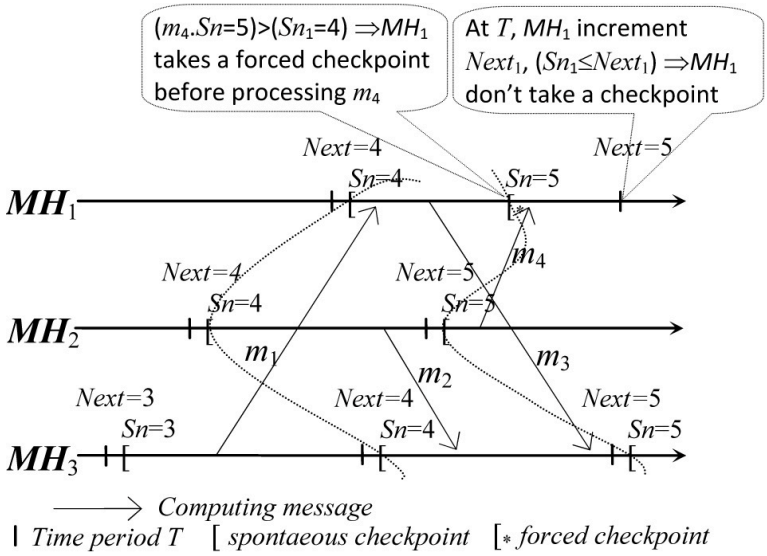

FIGURE 8.3 – Points de contrôle spontanés et points de contrôle forcés.

8.3.2 Phase 2 (Coordination)

Cette phase consiste, premièrement, à collecter tous les checkpoints ayant le même numéro de séquence (Sn_i) pour déterminer un *point de contrôle global cohérent et permanent*. Par la suite, ces points de contrôle sont sauvegardés en mémoire stable qu'on note SS (pour le mot anglais *stable storage*).

Cette phase est initiée lorsqu'un nœud a déjà effectué un nombre prédéfini de points de contrôle locaux. Plus concrètement, pendant que les points de contrôle sont pris, chaque nœud i vérifie si son numéro de séquence a atteint une valeur égale à un multiple du seuil prédéfini noté α. (voir ligne 15, figure 8.1 qui appelle la procédure $local\_CP$ définie dans la ligne 21 de la figure 8.1). En d'autres termes, chaque nœud vérifie s'il a pris α checkpoints locaux depuis la dernière coordination. Lorsque ce seuil est atteint pour un un nœud i, il initie la seconde phase appelée phase de coordination. (comme indiqué dans la ligne 17 de la figure 8.1).

A vrai dire, le premier nœud qui atteint ce seuil est déclaré l'initiateur de la phase de coordination. Par conséquent, ce dernier diffuse, d'abord, une requête de coordination (ligne 34, procédure *coordination*, figure 8.1) vers tous ses voisins directs. Les nœuds qui reçoivent une telle requête et qui ne sont pas en phase de coordination la diffusent à leur tour vers leurs voisins directs. Ainsi de suite, jusqu'à sa propagation vers tous les nœuds du réseau (où le partitionnement

n'est pas autorisé lors de cette phase). Lors de la réception de la requête de coordination, chaque nœud sauvegarde son dernier point de contrôle local pris comme un étant un *checkpoint tentative* dans la mémoire stable SS (ligne 36, figure 8.1).

Par ailleurs, si le numéro de séquence du dernier checkpoint du nœud est inférieur à celui qui a été embarqué dans la requête de coordination alors le nœud est contraint de prendre un *point de contrôle forcé* (exécution de la procédure $local\_CP$, ligne 3 de la figure 8.4).

D'un autre coté, nous appelons *point de contrôle permanent*, une tentative de checkpoint qui a été validée par l'initiateur une fois la phase de coordination phase a atteint son étape finale.

Le chackpoint permanent est le seul qui est pris en considération dans l'état global cohérent qui est utilisé lors du recouvrement arrière suite à la panne d'un nœud.

Pour assurer la terminaison de cette phase de coordination, nous adoptons le mécanisme défini par Cao et Singhal dans [67]. Ce mécanisme consiste à ce qui suit. Une variable réelle, notée *weight*, est embarquée (is piggybacked) dans les requêtes de coordination ainsi que dans les réponses aux requêtes. Sa valeur est initialisée à 1 par le nœud initiateur comme le montre la figure 8.1, lignes 17 et 29. Avant de propager la requête de coordination, chaque nœud divise la valeur de *weight* par 2 pour chacun des destinataires (lignes 33 et 34, fig. 8.1). Dans la réponse à la requête de coordination, la valeur restante de *weight* est envoyée vers l'initiateur (ligne 10 de la figure 8.4). Quand l'initiateur reçoit les réponses, il additionne les *weights* reçus (ligne 19, figure 8.4). Si le total est égal à 1 (ligne 21 de la figure 8.4), alors l'initiateur termine la phase de coordination en convertissant son propre point de contrôle tentative enregistré dans la SS en un *point de contrôle permanent*, puis demande aux autres nœuds à faire de même, en diffusant dans le réseau un *commit message*.

Ainsi, tous les nœuds mettent à jour leurs checkpoint tentative comme point de contrôle permanent (voir ligne 32 de la figure 8.4) dans la mémoire stable (SS).

Notons que la SS est généralement implémentée de manière répartie dans le contexte des MANETs. Par conséquent, l'initiateur est au courant seulement de la localisation de ses propres données dans la SS.

Exemple 3 *La figure 8.5 montre trois nœuds qui coopèrent dans le but de calcu-*

```
      // Task 2: Coordination request management
1     When i receives a CP_request(sn, initiator, w) from j {
2        if (not(CP_state_i) and (Sn_i<CP_request.sn)) {
3           local_CP(CP_request.sn);
4           Coord_i=CP_request.initiator;
5           if (Coord_i==NULL){
6              Coord_i=i;
7              Coordination(1, NULL);
8           }else {
9              Coordination(CP_request.w, j);
10             send reply(weight_i) to Coord_i;
11             Weight_i=0;
12          }
13       }else {
14             if (Coord_i=NULL) Coord_i= CP_request.Initiator;
15             send reply(CP_request.w) to CP_request.Initiator;
16       }
17    }
18    When i receives reply(w) from j {
19       Weight_i=Weight_i+reply.w;
20       List_i=List_i U {j};
21       if (Weight_i==1) {
22          for (all p in List_i) send commit() to p;
23          Weight_i=0;
24          List_i=∅;
25          Save tentative_checkpoint as permanent_checkpoint in SS;
26          Cp_i++;
27          CP_state_i=false;
28       }
29    }
30    When i receives commit() from j {
31       if (CP_state_i) {
32          Save tentative_checkpoint as permanent_checkpoint in SS;
33          Cp_i++;
34          CP_state_i=false;
35       }
36    }
```

FIGURE 8.4 – Tâche 2 : Gestion de la requête de coordination.

ler un point de contrôle global cohérent ayant comme numéro de séquence $Sn = 5$ *(*$\alpha = 5$*).* MH_2 *est l'initiateur car il est le premier a atteindre la valeur* 5*. Il sauvegarde son point de contrôle local comme étant une tentative dans la mémoire stable (SS). Par la suite, il envoie la requête* $CP\_request$ *à ses voisins* MH_1 *et* MH_3*. Lorsque ces deux derniers reçoivent cette requête, ils prennent un point de contrôle forcé, puis les sauvegardent comme checkpoints tentatives.*

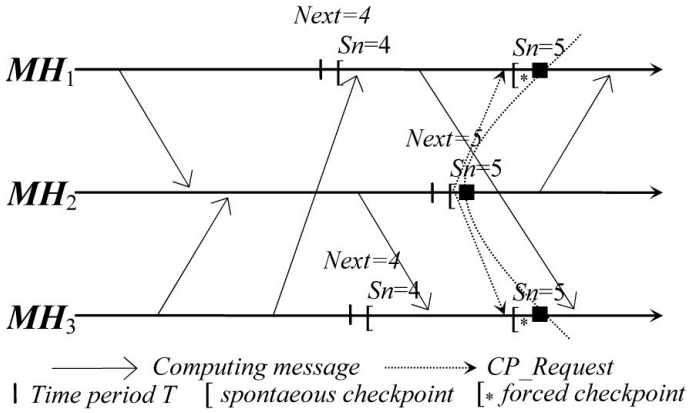

FIGURE 8.5 – Collecte d'un point de contrôle global cohérent.

8.4 Discussion

Nous discutons dans cette section quelques questions liées à l'implémentation de notre protocole. Particulièrement, nous nous concentrons sur l'interprétation du rendement de notre protocole vis à vis de certains aspects liés au contexte des MANETs

8.4.1 Initiatives concurrentes

Nous discutons dans cette section comment faire face au problème des initiatives de coordinations concurrentes. En d'autres termes, lorsque plusieurs nœuds lancent en parallèle la requête de coordination en tant qu'initiateurs. Ces différents initiateurs peuvent avoir des valeurs potentiellement différentes de Sn. Dans notre protocole, ce cas ne risque pas de se produire. En effet, lorsque des nœuds initient la phase de coordination, ils doivent avoir nécessairement la même valeur de Sn. De plus, une fois un nœud entre dans la seconde phase, il ne prend en considération que la première requête reçue et ignore les autres. En outre, si un nœud est déjà en phase de coordination, il doit ignorer toutes les autres requêtes de coordination. Ainsi, il répond à toute requête supplémentaire de coordination sans la propager vers ses propres voisins (ligne 15, figure 8.4. Par conséquent, les initiatives concurrentes ne sont pas un problème dans notre protocole et la ligne de recouvrement demeure cohérente. De plus, ils vont plutôt aider à terminer la phase de coordination de façon plus rapide. En effet, la circulation de plusieurs requêtes dans la réseau émanant de sources différentes font que ces requêtes arrivent à tous les nœuds du réseau plus rapidement que lorsque la requête émane

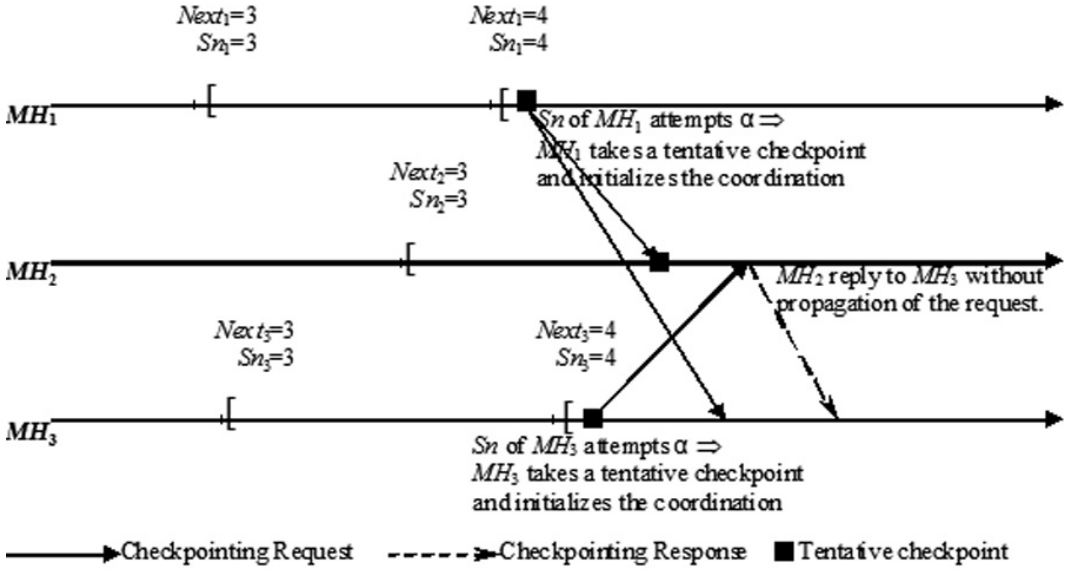

FIGURE 8.6 – Cas d'initiatives multiples.

d'une seule source.

Exemple 4 *Dans cet exemple (figure 8.6), nous avons posé* $\alpha = 4$. MH_1 *le premier à atteindre cette valeur. Ainsi, il devient automatiquement initiateur de la phase de coordination et diffuse la requête de coordination à ses voisins. Si* MH_3 *arrive à cette valeur avant de de recevoir la requête de la part de* MH_1*, il devient à son tour initiateur avec la même valeur de Sn. Lors de l'arrivée de la requête de* MH_1 *chez* MH_2*, ce dernier la traite et en attendant il reçoit celle de* $MH3$*. Dans ce cas,* $MH2$ *répond à* $MH3$ *(à cause de la variable weight) et ignore cette requête de coordination (Il ne la diffuse pas à ses voisins).*

8.4.2 Gestion de la mémoire locale

Chaque nœud enregistre au maximum α checkpoints localement. Ceci est réalisé à travers le vecteur C_i. Ainsi, chaque état local représentant le point de contrôle est sauvegardé à la position $k \in [0,\ \alpha]$ dans C_i, où k est égal à Sn_i *modulo* α. En effet, après chaque phase de coordination, le nouveau point de contrôle (spontané ou forcé) calculé écrase l'ancien de la phase précédente. Ainsi, un ramasse miette (garbage collector) est implémenté implicitement et de manière très simple.

8.4.3 Gestion des contraintes de mobilité

Nous discutons dans cette section les mécanismes utilisés dans notre protocole permettant de faire face aux contraintes liées à l'environnement mobile. Nous citons à titre d'exemple les points suivants :

- L'utilisation des mémoires locales durant la première phase réduit le surcoût engendré par les messages de contrôle permettant de transférer le point de contrôle vers la mémoire stable. Par conséquent, l'énergie et la bande passante sont utilisées de manière rationnelle et ainsi mieux gérées de façon plus efficace.
- Pour éviter la saturation de la mémoire locale, un ramasse miettes très simple est lancé (comme expliqué plus haut) à la fin de la phase de coordination afin d'écraser les anciens points de contrôle locaux.
- Les requêtes de coordination sont propagées à travers les voisins directs. Ceci évite au protocole d'être dépendant du protocole de routage. De plus, envoyer un message routé d'un nœud i vers un nœud j est plus coûteux que de le diffuser vers les voisins sans routage. Nous avons aussi évité la diffusion générale (le broadcast) de ce message à cause des initiatives multiples qui risquent d'inonder le réseau avec les requêtes de coordination.

Un autre défi à relever lors de la gestion des contraintes de l'environnement ad hoc mobile est les déconnexions des nœuds qui peuvent être volontaires ou involontaires. Pour cet effet, un traitement spécifique est prévu. Les mesures prises lors de la déconnexion ou la re-connexion d'un nœud sont décrites dans la figure 8.7. De plus, certaines vérifications et contrôles sont donnés dans la figure 8.1.

Vu que les déconnexions sont un phénomène commun dans les MANETs, nous avons prévu au niveau de chaque nœud i une variable notée $connect_i$. cette variable peut prendre trois valeurs possibles :

- 0 : désigne qu'un nœud a subi une déconnexion volontaire ou involontaire (batterie et/ou signal faible).
- 1 : désigne qu'un nœud est connecté et il participe activement au calcul distribué.
- 2 : désigne qu'un nœud est en train d'initialiser une re-connexion mais qui n'a toujours pas rejoint le calcul distribué (connecté mais inactif)

Il faut noter qu'un mécanisme de connectivité est supposé être implémenté au niveau de chaque nœud. Ce qui permet à un nœud d'avoir le temps nécessaire

```
      // Task 3: Disconnection management
1     When i needs to disconnect { //voluntary disconnection or signal problem
2        Connect_i=0;
3        if (CP_state_i) Save tentative_checkpoint as disconnect_checkpoint in SS;
4        else CP_spontaneous();
5     }
6     When i needs to reconnect {//voluntary disconnection or signal problem
7        Connect_i=2;
8        for (all p in neighbour_i) send reconnect() to p;
9     }
10    When i receives reconnect() from j {
11       if(Connect_i==1) send reply_connect(Sn_i,Cp_i,Coord_i,CP_state_i) to j
12    }
13    When i receives reply_connect(sn,ckpt,c,cs) from j {
14       if (Connect_i==2){
15          if (reply_connect.cs) {
16             if((Cp_i<reply_connect.ckpt)or not(CP_state_i)) {
17                Coord_i=reply.c;
18                Sn_i=reply.sn;
19                send reply(0) to Coord_i;
20             }
21             Save disconnect_checkpoint as tentative_checkpoint in SS;
22          }else {
23             if(CP_state_i) {
24                Coord_i=reply.c;
25                Save disconnect_checkpoint as permanent_checkpoint in SS;
26             }elsif(Cp_i==reply.ckpt) delete disconnect_checkpoint in SS;
27              else {
28                 Sn_i=reply.sn;
29                 Save disconnect_checkpoint as permanent_checkpoint in SS;
30              }
31          }
32          Connect_i=1;
33          for (all queued message m) {
34             if (m.Sn>Sn_i) local_CP(m.Sn);
35             process m;
36          }
37       }
38    }
```

FIGURE 8.7 – Gestion des déconnexions et re-connexions.

pour réagir aux déconnexions volontaires (niveau de batterie faible ou puissance du signal mauvaise). Donc, quand un nœud risque de se déconnecter, il affecte à la variable $Connect_i$ la valeur 0 puis vérifie s'il est en phase de coordination phase, ou non (selon la valeur de la variable CP_state_i).

Si le nœud est en phase de coordination et a déjà pris et enregistré son *point de contrôle local* comme une tentative dans la mémoire stable, alors il doit le convertir en *point de contrôle de déconnexion* (*disconnecting checkpoint*). Dans le cas contraire, il prend un point de contrôle spontané qu'il sauvegarde localement et aussi dans la mémoire stable comme étant un *point de contrôle de déconnexion*.

Dans le cas où le nœud se trouve toujours dans la première phase, il prend un point de contrôle forcé et le sauvegarde dans la SS comme un *point de contrôle de déconnexion*.

En outre, il est inconcevable qu'un nœud qui risque de se déconnecter assure la coordination de la seconde phase. Pour cet effet, il peut juste lancer la seconde phase sans jouer le rôle de coordinateur. Ainsi, il envoie la requête de coordination avec l'identificateur (*id*) du coordinateur égal à $NULL$. Lorsque ses voisins reçoivent la requête, il deviennent coordinateurs.

Par ailleurs, dans notre algorithme, la connectivité est définie par la variable $connectivity_i$ et sa valeur minimale (le seuil) est définie par la constante $Level$. Ces valeurs sont définies par le système.

Lorsqu'un nœud se reconnecte, il ne peut pas savoir ce qui s'est passé durant son absence dans le système et particulièrement ce qui est en relation avec l'application distribuée. Par conséquent, un traitement spécifique est nécessaire durant sa re-connexion.

Ainsi, un nœud qui vient de se reconnecter doit informer ses voisins en envoyant un *message de reconnexion* noté *reconnect*. A ce moment, il est considéré dans un état "*enattente de reconnexion* et ce, en assignant la valeur 2 à la variable $Connect_i$.

Durant cet état, un nœud ne doit pas traiter les messages de calcul reçus pour éviter tout cas d'incohérence. Cependant, il les enregistre dans liste d'attente pour les traiter par la suite dans un ordre FIFO (premier arrivé, premier traité).

Chacun des voisins du nœud qui vient de reconnecter et lors de la réception du message *reconnect*, il doit l'informer sur l'état actuel de l'application distribuée en envoyant un message *reply_connect*. Dans ce message, on embarque le

numéro de séquence du dernier point de contrôle local pris par ce voisin, ainsi que le numéro de séquence de son dernier point de contrôle permanent enregistré dans la SS. Outre, il informe le nœud qui vient de se reconnecter si la tâche de checkpointing est en phase de coordination ou non et l'identité du nœud coordinateur si c'est le cas.

Lors de la réception du message *reply_connect*, le nœud qui vient de se reconnecter met à jour ses informations locales selon l'état courant de l'application distribuée. Les points de contrôle de déconnexion (enregistrés dans la SS juste avant la déconnexion) sont traités selon les cas suivants :

- Si les nœuds voisins sont en phase de coordination, il doit convertir le point de contrôle de déconnexion en point de contrôle tentative.
- S'il s'est déconnecté en pleine phase de coordination et après sa re-connexion il trouve ses voisins ne sont pas en phase de coordination, il convertit son point de contrôle de déconnexion en point de contrôle permanent.
- S'il s'est déconnecté alors qu'il était dans la première phase et aucune coordination n'a été initialisée durant sa déconnexion, alors il supprime son point de contrôle de déconnexion dans la SS.
- S'il s'est déconnecté lors de la première phase et ses voisins ont terminé une ou plusieurs coordinations durant son absence, il convertit son point de contrôle de déconnexion déjà enregistré dans la SS en point de contrôle permanent.

Après la mise à jour de son état local, le nœud qui vient de se re-connecter affecte au paramètre $Connect_i$ la valeur 1 puis traite tous les messages de calcul mis en attente.

8.4.4 Mémoire Stable

L'implémentation de la mémoire stable (en anglais appelé stable storage et qu'on note ici SS) est un grand défi à relever dans le contexte des MANETs. Dans notre solution, nous avons opté pour l'utilisation de la technique de réplication de données pour assurer la disponibilité des informations sur les points de contrôle. En effet, la réplication a toujours été une solution de tolérance aux pannes pour assurer la disponibilité dans les systèmes répartis. Nous proposons la solution proposée par Moussaoui et al. dans [90] qui consiste à créer une copie de la donnée originale à chaque k sauts (hops) tout en évitant de générer les mêmes copies dans deux nœuds voisins. Pour assurer une mise à jour cohérente, un

algorithme de réplication pessimiste avec un protocole d'invalidation sont pris en considération [91].

Les performances de cette solution sont donnés dans [90] et [91]. Par ailleurs, la réplication peut être coûteuse mais elle reste la solution la plus indiquée pour assurer la disponibilité dans un environnement distribué.

8.5 Recouvrement

Dans le cas où une panne d'un nœud survient, le *mécanisme de détection*,qui est supposé s'exécuter en parallèle (comme nous l'avons expliqué dans les deux chapitres précédents), informe les autres nœuds du système de l'occurrence de cette défaillance.

Une fois informés, tous les nœuds arrêtent leur calcul en attendant que le processus de recouvrement soit lancé. Par la suite, le module de recouvrement est initialisé afin de faire retourner en arrière tous les nœuds impliqués dans l'application distribuée vers le dernier point de contrôle permanent.

Le protocole de recouvrement est très simple grâce à la phase de coordination. Cette phase fournit un état global cohérent (correspondant à la ligne de recouvrement).

Ainsi, le nœud qui remplace le nœud défaillant se charge de coordonner l'opération de recouvrement. Il commence par télécharger son point de contrôle de sa mémoire stable (SS) et retourne vers ce point dans son calcul.

Le nœud coordinateur doit informer tous les autres nœuds en diffusant dans le réseau un message appelé *rollback* dans le but que chaque nœud qui reçoit un tel message retourne en arrière.

Ainsi, lorsqu'un nœud actif reçoit le message *rollback*, il fait un retour arrière vers le point de contrôle local correspondant au checkpoint permanent enregistré sur la mémoire stable. Pour cet effet, nous utilisons une variable booléenne R_i au niveau de chaque nœud i. Cette variable est initialisée à $faux$ à la fin de chaque coordination. Lors de la réception d'un message qui met à jour la variable Sn_i locale, la valeur de R_i est alors mise à $vrai$. Ceci permet aux nœuds n'ayant pas reçu de messages depuis la dernière coordination de ne pas retourner en arrière. Ainsi, nous réduisons la perte en quantité de calcul.

La figure 8.8 illustre le recouvrement par retour arrière (rollback-recovery).

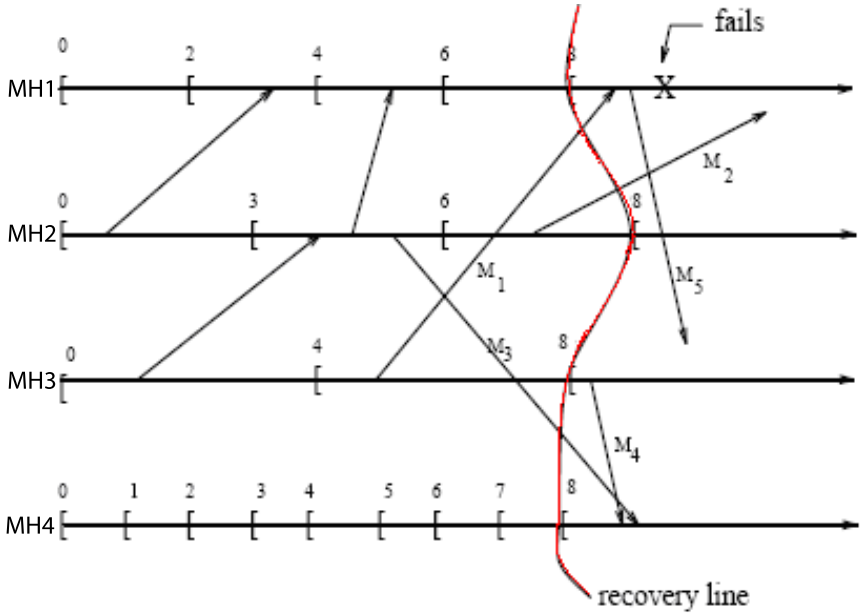

FIGURE 8.8 – Exemple de recouvrement par retour arrière.

8.6 Analyse théorique

Nous discutons dans cette section quelques aspects théoriques qui doivent être satisfaits par notre protocole.

8.6.1 La Cohérence

Le protocole de calcul de points de contrôle doit retourner *un état global cohérent*. Dans notre protocole, la *ligne de recouvrement* est constituée d'une collection de points de contrôle ayant le même *numéro de séquence* (Sn) Ces points de contrôle sont collectés à l'issue de la première phase ($phase\ quasi-synchrone$).

Theoreme 5 *L'algorithme 2PACA enregistre toujours en mémoire stable un état global cohérent, G.*

Proof. Nous démontrons le théorème ci-dessus par l'absurde. Supposons qu'il existe une situation où l'algorithme enregistre un état global du système G qui n'est pas cohérent.
$\Longrightarrow$ Il existe un message M envoyé par un nœud i vers un nœud j qui est orphelin dans G. $\Longrightarrow$ Il existe un message M, tel que l'événement d'envoi de M par un

nœud i, qu'on note $send(M)$, n'appartient pas à G et l'événement de réception de M par un nœud j, qu'on note $receive(M)$ appartient à G.
$\Longrightarrow$ Soit CSS la valeur de Sn correspondant au point de contrôle global G enregistré dans la SS.

Soit Cbs le numéro de séquence du dernier point de contrôle pris par le nœud i avant l'envoi de M.
$\Rightarrow CSS \leq Cbs$ ...(1)

Soit Cbr le numéro de séquence du dernier point de contrôle pris par le nœud j avant la réception de M.
$\Rightarrow CSS > Cbr$ ...(2)
$\Longrightarrow$ A partir de (1) et (2), nous avons par transitivité $Cbs > Cbr$ ...(3)

D'un autre coté, i embarque dans le message M son Sn_i qui est égal à Cbs (en d'autres termes, $M.Sn = Cbs$). Lorsque j reçoit M, il compare son numéro de séquence $Sn_j = Cbr$ avec celui reçu dans M $M.Sn$.
Nous avons deux cas possibles qui peuvent survenir :
(i) Si $M.Sn > Sn_j$, alors il prend (selon l'algorithme) un point de contrôle local avec un numéro de séquence égal à $M.Sn = Cbs$ et l'événement de réception de M n'est pas enregistré dans ce checkpoint. Il s'agit du dernier point de contrôle local avant le traitement du message M.
$\Rightarrow Cbs = Cbr$, nous avons une contradiction avec (3)
(ii)Si $M.Sn \leq Sn_j$, alors il traite M et ne prend pas un point de contrôle local.
$\Rightarrow$. Dans le prochain checkpoint local, l'événement de réception est enregistré.
$\Rightarrow$ Le numéro de séquence du dernier point de contrôle local avant le traitement de M qu'on a noté ci-dessus $Cbr = Sn_j$. $\Rightarrow Cbr \geq M.Sn = Cbs$
$\Rightarrow Cbr \geq Cbs$. Nous avons une contradiction avec (3).

Nous déduisons que l'algorithme 2PACA enregistre toujours un point de contrôle global cohérent dans la mémoire stable (SS). ■

8.6.2 La terminaison de la phase de coordination

Un autre point très important dans un algorithme distribué est la *terminaison*. Cette propriété est assurée grâce au mécanisme utilisé dans la phase de coordination (seconde phase) et ce à travers la variable *weight*. Nous avons expliqué auparavant comment fonctionne ce mécanisme afin d'achever la phase de coor-

dination du protocole correctement. Les preuves de la terminaison de la coordination en utilisant ce mécanisme peuvent être consultées dans [67]

8.6.3 Performances théoriques

Les performances d'un protocole de calcul de points de contrôle basé recouvrement peuvent être évaluées selon les critères suivants :

1. Messages supplémentaires (de contrôle) dues au checkpointing,
2. Nombre de checkpoints dans la SS,
3. Mobilité,
4. Latence,
5. Passage à l'échelle (Scalabilité).

Le tableau 8.1 montre les performances théoriques du protocole $2PACA$:

TABLE 8.1 – Les performances théoriques de $2PACA$.

Critère	**Observation**
Messages supplémentaires	Messages de coordination : $O(n)$ Information embarquée : $O(1)$
Nombre de Checkpoints dans SS	≤ 2 (tentative/déconnexion, permanent)
Coût du recouvrement	Processus de recouvrement : simple Pas d'effet domino.
Latence	Moyenne
Nombre de nœuds	inconnu / variable
Gestion de la mobilité	Communication par voisinage gestion des déconnexions

8.7 Analyse des Performances

Dans cette section, nous présentons les résultats des simulations qui évaluent les performances de notre protocole. Pour cet effet, nous avons réalisé deux séries différentes de tests. La première série consiste à tester le paramètre α. La seconde permet de comparer notre protocole avec un protocole de calcul de points de contrôle coordonné similaire à celui de Chandy et Lamport [30] que nous avons adapté à l'environnement mobile ad hoc en propageant les requêtes par voisinage.

Pour simuler notre protocole, nous utilisons le simulateur de réseaux NS2 [1] [88]. Pour cet effet, nous avons considéré les paramètres suivants :

— La surface du réseau est fixée à 1000x1000 mètres.

— La portée du signal pour un nœud donné est fixée à 250 mètres.

— Nous considérons le protocole de routage AODV [2] [93] Ceci est motivé par le fait que ce dernier est le protocole le plus célèbre dans le contexte des MANETs et il est déjà implémenté sous le simulateur NS. De plus, pour le déroulement des simulations sous NS, il est obligatoire de définir le protocole de routage.

Durant les simulations que nous avons effectuées, nous avons varié les métriques suivantes :

— Nombre de nœuds : 5, 20, 50, 100.

— Types de mobilité : Forte, Moyenne, Faible.Nous définissons la mobilité comme étant la variation moyenne des distance entre les nœuds (deux à deux).En effet, il est à rappeler qu'en réalité la mobilité n'est pas la variation de la vitesse relative d'un nœud mais c'est le mouvement relatif entre les nœuds.

Cependant, la durée en moyenne de chaque simulation est de 600 secondes.Par ailleurs, pour certains tests, nous exécutons les simulations pour 100, 200, 500 et 1000 secondes.Pour les déconnexions, nous avons pris en considération trois types de scénarios : sans déconnexion, le cinquième (1/5) des nœuds se déconnectent toutes les 100 secondes, la moitié (1/2) des nœuds se déconnectent toutes les 100 secondes.

Nous avons mesuré ce qui suit :

i) La latence.

ii) Nombre de checkpoints par nœud.

iii) Nombre de messages de contrôle.

iv) Sur-coût en messages par nœud

Chaque point des graphes obtenus est le résultat de 5 valeurs obtenues de 5 tests différents. Où les cinq tests correspondent à cinq exécutions de l'algorithme avec cinq scénarios différents. Ces scénarios ont en commun le même nombre de nœuds et le même type de mobilité. Ainsi, à chaque combinaison de type de

1. Une étude comparative des simulateurs de réseaux peut être trouvée dans [92]
2. Par ailleurs, 2PACA ne dépend pas d'un protocole de routage spécifique.

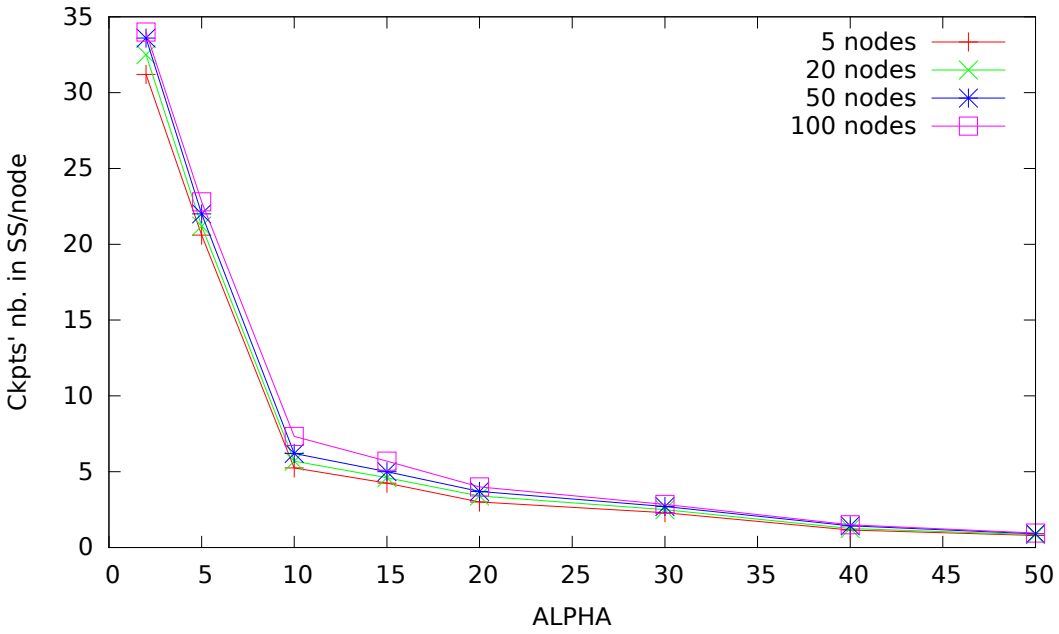

FIGURE 8.9 – Nombre de checkpoints dans SS par nœud par rapport à α

mobilité et nombre de nœuds correspond cinq scénarios différents.Nous avons ignoré les valeurs extrêmes (le maximum et le minimum). Nous avons par la suite calculé la moyenne des trois valeurs restantes.

8.7.1 Première série de simulations

Dans la première série de simulations effectuées, nous essayons de déterminer l'impact de la variation du paramètre α sur les performances de notre protocole de calcul de points de contrôle. Les résultats sont montrés par les figure 8.9 et 8.10. De la figure 8.9, nous remarquons que la valeur de α n'est pas affectée par la variation du nombre de nœuds dans le systèmes. Cependant, pour une très grande valeur de α, nous observons un nombre minime de coordinations. Par conséquent, l'état global cohérent calculé risque de ne pas être assez récent. Ainsi, ceci peut mener à un effet domino lorsque la valeur de α tend vers l'infini (∞). Par ailleurs, une très petite valeur de α affecte négativement le nombre de messages de contrôle. En se référant aux figures 8.9 et8.10, nous considérons α=20 comme valeur optimale pour la suite de nos simulations.Pour cette valeur, on observe une faible congestion du réseau par les messages de contrôle et suffisamment de points de contrôle globaux pour calculer la ligne de recouvrement la plus récente.

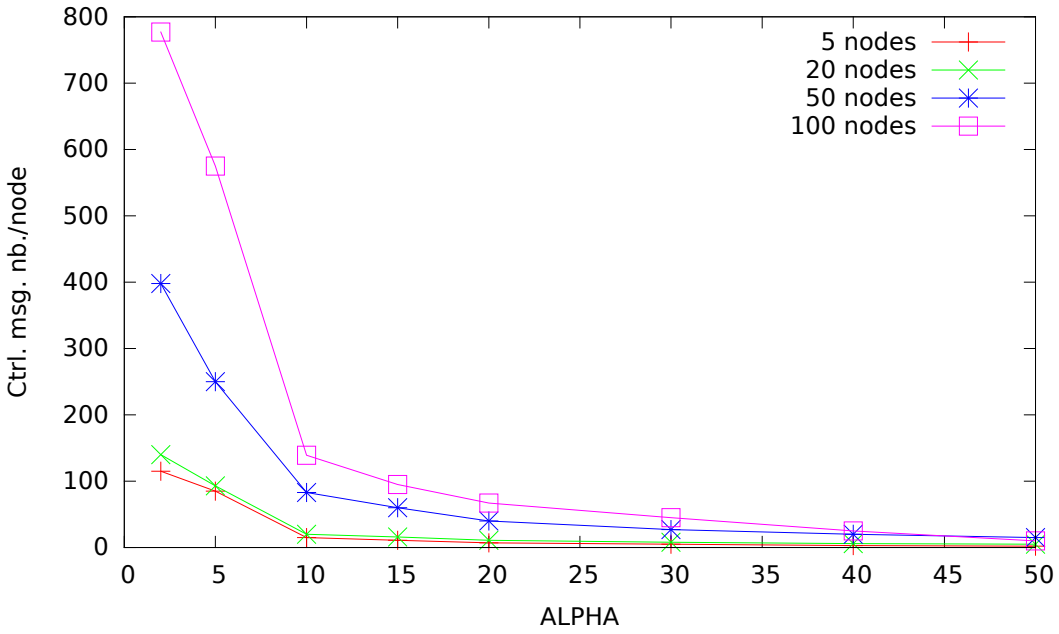

FIGURE 8.10 – Nombre de messages de contrôle par nœud par rapport à α

Ainsi, nous pouvons affirmer que notre protocole peut faire face à toute situation et tout type d'application distribuée en ajustant la valeur de α. Par contre, pour les applications qui ne sont pas sensibles au temps, le protocole devrait envisager une faible valeur de α pour se comporter comme un protocole coordonné. Pour les réseaux où la panne est rare, le protocole devrait envisager une valeur élevée de α, réduisant ainsi les effets secondaires du protocole sur l'application distribuée.

8.7.2 Seconde série de simulations

Dans la seconde séries de tests, nous comparons les performances de notre protocole avec celle d'un protocole de checkpointing coordonné.

Pour analyser les performances de notre protocole de calcul des points de contrôle, nous évaluons la *latence* et la *surcharge en messages de contrôle*. A partir de 20 nœuds, la latence due au calcul des points de contrôle commence à décroître (voir figure 8.11). Cela est dû à la concentration des nœuds sur la surface. De ce fait, les nœuds ont plus de voisins à travers lesquels ils propagent plus vite les requêtes de coordination dans le réseau. Cela se traduit par la réalisation de la tâche de checkpointing plus rapidement. En outre, l'augmentation

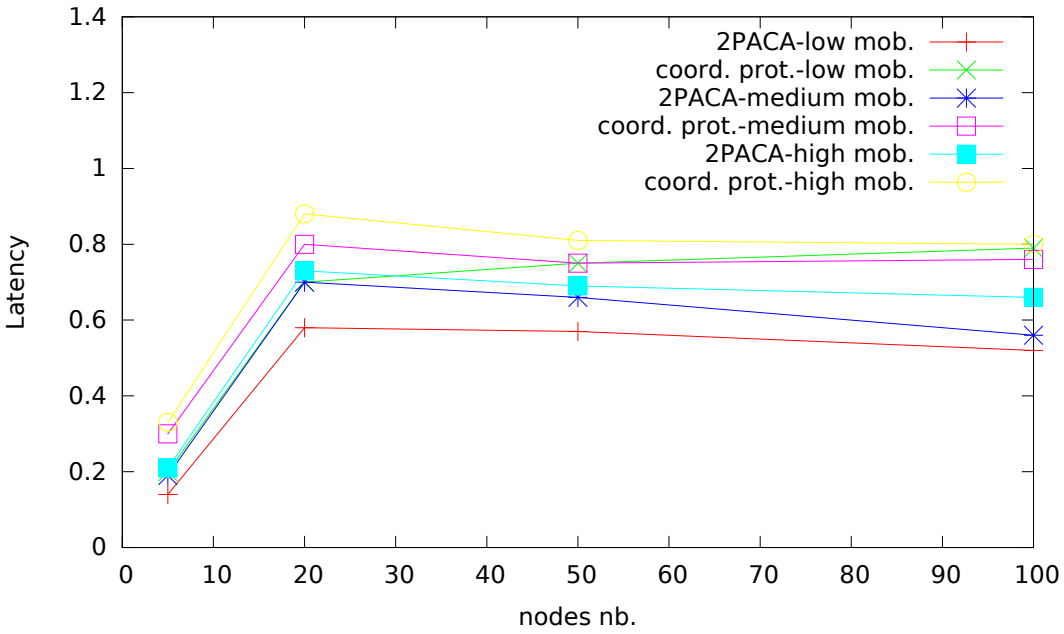

FIGURE 8.11 – Latence par rapport au nombre de nœuds.

du nombre de nœuds intensifie le nombre de coordinations simultanées, ce qui entraîne une propagation rapide des requêtes de coordination. Il faut noter que la mobilité n'a presque aucune influence sur la latence. Ceci est due au mécanisme qu'on adopté pour la gestion de la mobilité.

Nous remarquons que notre protocole donne de meilleurs résultats pour ce paramètre en comparaison avec le protocole coordonné. Ceci est dû à l'utilisation de la phase quasi-synchrone et les coordinations simultanés dans notre protocole.

Selon la figure 8.12, le nombre de messages de contrôle échangés augmente proportionnellement au nombre de nœuds. Cependant, le mode de propagation adopté dans notre protocole empêche une augmentation exponentielle des messages de contrôle. Il faut également noter que la mobilité a un impact modéré sur cette mesure.

Par ailleurs, les deux protocoles offrent des performances similaires. Cela est dû principalement au même mécanisme utilisé pour transmettre les messages de coordination dans le réseau, à savoir la propagation par voisinage.

Dans la figure 8.13, nous présentons l'impact des déconnexions sur les performances de notre protocole.Il est montré que les déconnexions ont un effet minime sur le nombre de messages enregistrés dans la SS qui est principalement dû au mécanisme de déconnexion proposée (voir section 8.4).

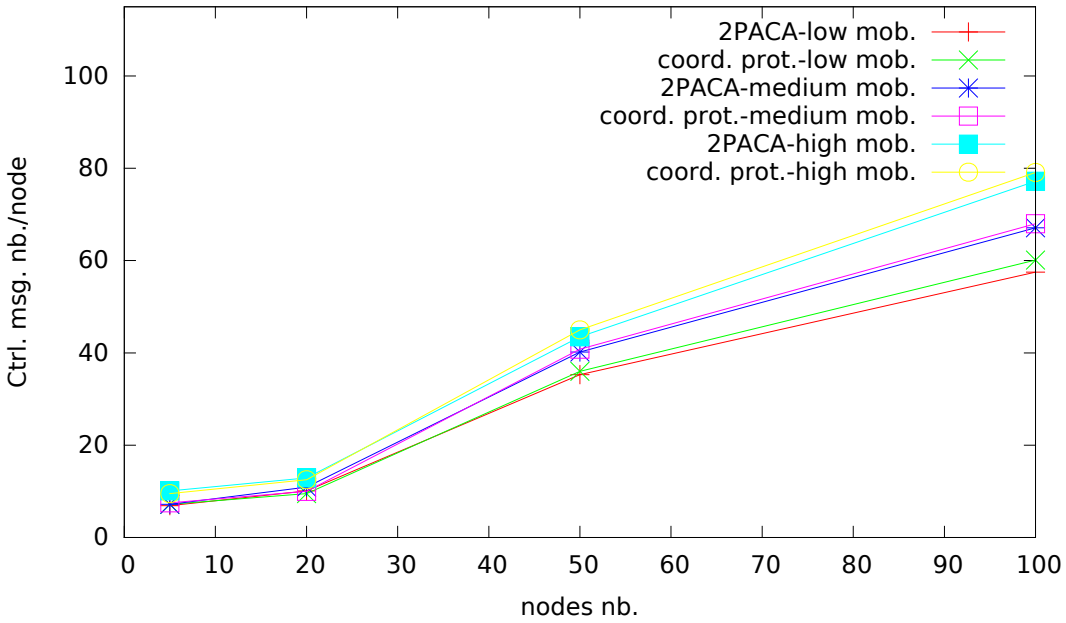

FIGURE 8.12 – Messages de contrôle par nœud par rapport au nombre de nœuds.

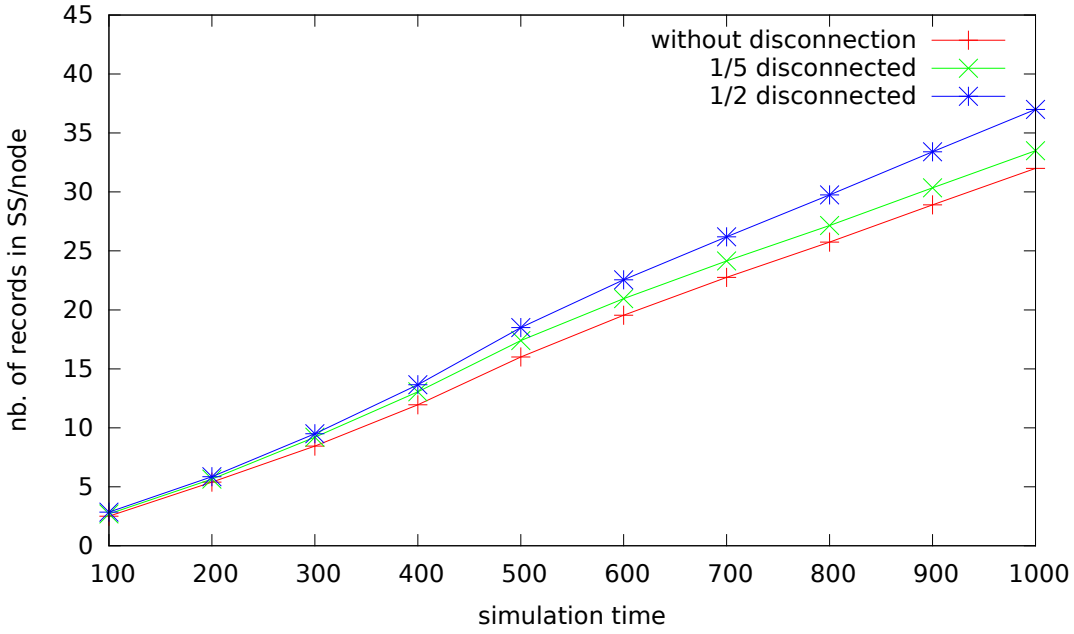

FIGURE 8.13 – Effet des déconnexions sur le nombre d'enregistrements dans SS.

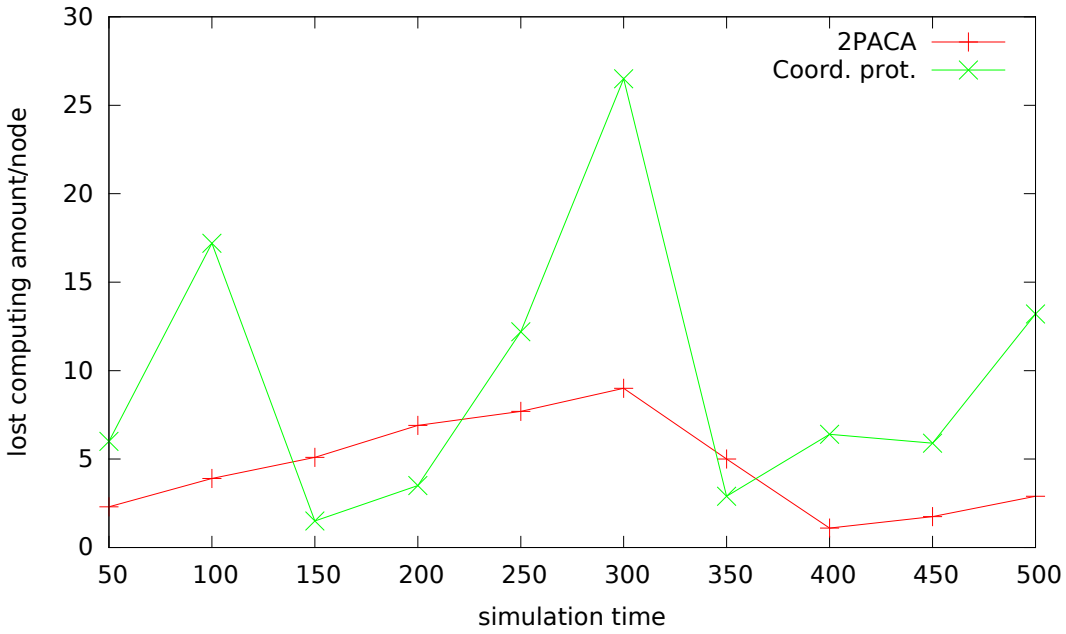

FIGURE 8.14 – Quantité moyenne de calcul perdu par rapport au temps.

Il faut noter que la majorité des protocoles de calcul de points de contrôle ne met pas en œuvre un mécanisme de gestion de la déconnexion. Par conséquent, il n'y a donc aucun moyen de comparer notre protocole avec les autres.

Dans les figures 8.14 et 8.15, nous évaluons les performances globales de notre protocole de checkpointing par rapport à la tâche de recouvrement par retour arrière en cas de pannes. Ainsi, nous avons mesuré la quantité de calcul perdu en cas de recouvrement.Cette perte est estimée en temps d'exécution.

Nous constatons que la perte est insignifiante et le processus de recouvrement est grandement amélioré. En effet, notre protocole de checkpointing permet de fournir des lignes de recouvrement assez récentes en cas de défaillance.

Comme l'évaluation de ce paramètre dépend de l'intervalle de temps durant lequel les simulations sont effectuées, nous avons examiné différentes simulations avec différentes plages de temps pour réaliser une évaluation précise des performances de notre protocole. Par conséquent, nous constatons que pour notre protocole, la perte en calcul ne dépasse jamais un seuil donné. Cependant, pour le protocole coordonné, la perte peut être énorme en cas de reprise. Ceci est expliqué par le fait que la phase de coordination se produit de façon aléatoire.

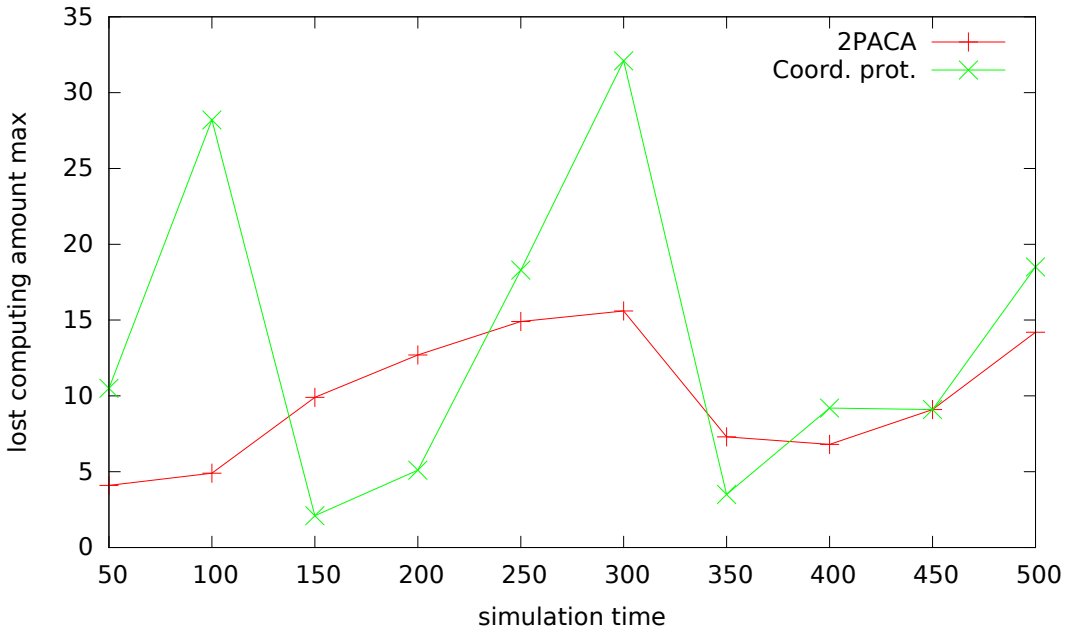

FIGURE 8.15 – Quantité maximale de calcul perdu par rapport au temps.

8.8 Conclusion

Dans ce chapitre, nous avons traité le problème du recouvrement dans les applications distribuées afin d'assurer la tolérance aux pannes. Dans ce contexte, nous avons présenté un nouveau protocole de calcul des points de contrôle hybride dédié aux MANETs plats. Pour gérer les contraintes des MANETs, notre protocole, appelé $2PACA$, combine deux phases pour effectuer la tâche de checkpointing. Une première tâche où les points de contrôle sont pris au niveau local de manière asynchrone et une seconde phase durant laquelle la coordination est lancée afin de déterminer la ligne de recouvrement cohérente du système.

Nous avons évalué les performances de $2PACA$ à travers des simulations. Une analyse théorique a été aussi présenté dans ce chapitre. Les résultats obtenus montrent que le protocole donne un point de contrôle global cohérent et que les performances ne sont pas affectées par la mobilité ou l'augmentation de la taille du réseau (passage à l'échelle/scalabilité)

Ce chapitre termine la description des différents protocoles constituant notre outil de tolérance aux pannes. Dans le chapitre suivant, nous donnons la conclusion générale ainsi qu'une présentation des travaux futurs.

Chapitre 9

Conclusions

Ce chapitre résume brièvement les travaux entrepris dans le cadre de cette thèse à travers le bilan ainsi que les principaux résultats obtenus et donne un aperçu sur quelques perspectives qui vont guider les travaux futurs.

9.1 Bilan

Nous avons commencé cette thèse par une étude du contexte et de l'environnement de travail. Nous avons présenté dans les chapitres 2 et 3 le domaine des systèmes distribués et la tolérance aux pannes ainsi que l'environnement mobile. Nous avons par la suite étalé notre étude de l'état de l'art par le survol des techniques de détection de pannes, de calcul de points de contrôle et de recouvrement arrière. Quelques travaux connexes ont été présentés et discutés.

Dans le cadre de notre thèse, nous avons proposé plusieurs protocoles de tolérance aux pannes dans les réseaux mobiles ad hoc (MANETs). Nous avons aussi proposé une synchronisation entre les différents protocoles pour permettre à l'outil de bien fonctionner et permettre à une application distribuée de se dérouler normalement.

Nous avons aussi présenté les résultats expérimentaux. Les résultats obtenus ont été très intéressants. dans le domaine de la détection de pannes, nous avons développé un protocole de détection des pannes appelé FDAN [7]. D'autres améliorations ont été apportées à ce protocole. Par la suite, les protocoles AFDAN [8] et FDRAM [9] ont été conçus. Ces protocoles prennent en considération : La non connaissance au préalable du réseau et des autres nœuds participant au calcul distribué, la limitation des ressources (processeur, mémoire, bande passante,

énergie, ...), les connexions, déconnexions et arrivée de nouveaux nœuds, ...

Par ailleurs, nous avons montré qu'avec l'utilisation des deux niveaux de listes et des techniques de corrections qu'on arrive à améliorer la justesse (réduction des fausses suspicions) de façon sensible.

Pour le calcul des points de contrôle (checkpointing), le protocole 2PACA [10][11] a été développé pour l'environnement ad hoc. Il définit aussi un protocole de reprise et se base sur la réplication pour sauvegarder les points de contrôle [11]. Le protocole de calcul de points de contrôle et de recouvrement, nous avons réussi à hybrider deux techniques différentes de checkpointing. Ceci nous a permis de tirer profit de chacune d'elle et parallèlement éviter les inconvénients. Nous avons réussi à gérer les contraintes de mobilité et prouvé que l'état global obtenu est toujours cohérent. La sauvegarde des points de contrôle en mémoire stable est une autre problématique dans les réseaux ad hoc. Nous avons proposé d'utiliser la réplication.

Tous les protocoles développés ont été implémentés à travers le simulateur de réseaux *NS*. L'analyse des performances a donné de bons résultats. Selon les résultats de simulation des protocoles de détection de pannes, nous avons réussi à améliorer sensiblement la justesse. Ceci, nous évite de lancer inutilement le recouvrement. Pour le protocole de checkpointing, les résultats obtenus ont montré que le protocole génère un coût minimal ...

9.2 Perspectives

Plusieurs travaux peuvent être réalisés ou explorés dans le futur. Les principaux axes sont la modélisation formelle afin de valider et analyser les performances de l'outil de tolérance aux pannes et les éventuelles améliorations qu'on peut apporter aux différents protocoles développés.

9.2.1 Analyse formelle

Nous proposons de réaliser une validation et une analyse de performances formelle de nos protocoles et ce à travers une modélisation de l'outil de tolérance aux pannes [94]. Nous pouvons nous baser sur les réseaux d'automates stochastiques (SAN).

La modélisation permet de mieux exprimer le fonctionnement et la synchro-

nisation entre les différents protocoles. La modélisation est une étape cruciale dans le développement des protocoles afin de rendre possible par la suite la validation de certaines propriétés, vérifier la cohérence et évaluer les performances de façon formelle. Ainsi, le processus requiert un modèle formel capable de capturer les propriétés comportementales et de préserver les propriétés. De ce fait, le choix du modèle approprié est primordial. Dans ce cadre, nous avons prospecté plusieurs pistes : Réseaux de Pétri, Chaînes de Markov, Réseaux Stochastiques, automates...

Nous nous sommes intéressés particulièrement aux SAN (Synchronized Automat Networks) qui sont détaillés dans les travaux de Plateau et Fourneau [95], Plateau et Atif [96], Mokdad et Ben-Othman[97][98], Clement et al. [99], Greve et al. [100] , ...

Ce choix est justifié par le fait que les SAN (Stochastic Automata Networks) sont les plus adaptés à notre cas qui nécessitent des synchronisations à différents niveaux.

Effectivement, les protocoles que nous avons développés nécessitent un nombre très important de synchronisations inter et intra protocoles. Ainsi, une technique de validation basée sur les SAN a été proposée [94] pour cet ensemble de protocoles.

Les SAN semblent être plus efficaces que les réseaux de files d'attente ou les réseaux de Petri stochastiques pour modéliser des systèmes avec un grand nombre d'états et de synchronisations complexes. D'autre part, les réseaux de files d'attente donnent une représentation très compacte des systèmes avec des conflits de ressources entre des clients indépendants. Les méthodes d'analyse et des algorithmes bien connus peuvent être utilisés pour obtenir soit des résultats analytiques ou numériques. Mais les réseaux de files d'attente sont inefficaces lorsque les contraintes de synchronisation complexes doivent être prises en compte.

Aussi, comme perspectives, nous proposons d'utiliser des outils comme PEPS [101] afin d'évaluer les performances de nos protocoles. Ces outils permettent de résoudre numériquement les SAN de façon optimale. Nous proposons aussi de comparer les résultats obtenus avec les résultats de simulations obtenus à l'aide de NS.

9.2.2 Amélioration du protocole de détection de pannes

Il est intéressent de rendre le protocole de détection de pannes complètement asynchrone. En effet, nous pouvons envisager d'éliminer complètement l'effet des temporisateurs ainsi que leur gestion. Pour cet effet, un nœud peut attendre la réception d'un certain nombre de battements de cœur pour placer certains nœuds dans une des deux listes de suspects. Ce nombre de battement de cœur qu'on appelle seuil β qui doit être défini après étude. Mais vu que le système est asynchrone, ce nombre risque de prendre une assez longue durée pour qu'il soit atteint et ceci influe négativement sur le temps nécessaire pour détecter une panne. Pour résoudre ce problème, nous pouvons proposer de rajouter un délai d'attente τ et le nœud prend la décision soit lorsque le délai est atteint ou lorsque le nombre de battements de cœur reçu a atteint le seuil β défini.

D'autres améliorations peuvent être étudiées comme l'exploration de solutions pour les aspects liés au partitionnement du réseau ou l'utilisation des agents mobiles qui peuvent s'avérer comme solution pour améliorer la détection des pannes et mieux propager l'information dans tout le réseau.

9.2.3 Amélioration du protocole de checkpointing et recouvrement

Des améliorations restent possibles pour les protocoles proposés. Par exemple, améliorer le recouvrement pour que les nœuds qui ne sont pas dépendants causalement (de façon directe ou indirecte) avec le nœud défaillant ne soient pas obligés de faire un retour arrière.

Nous envisageons aussi à étudier le cas du partitionnement du réseau lors de la coordination. et pour d'autres alternatives pour mettre en œuvre la mémoire stable SS.

Bibliographie

[1] O. Babaoğlu and K. Marzullo, "Distributed systems (2nd ed.)," S. Mullender, Ed. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1993, pp. 55–96. [Online]. Available : http ://dl.acm.org/citation.cfm ?id=302430.302434

[2] A. S. Tanenbaum and M. van Steen, *Distributed systems - principles and paradigms (2. ed.)*. Pearson Education, 2007.

[3] N. Badache, "Ordre causal et tolérance aux défaillances en environnement mobile," Thèse de doctorat, USTHB, Algérie, 1998.

[4] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[5] J.-M. Hélary, A. Mostéfaoui, and M. Raynal, "Points de contrôle cohérents dans les systèmes répartis : concepts et protocoles," *TSI. Technique et science informatiques*, vol. 17, no. 10, pp. 1223–1245, 1998.

[6] H. Benkaouha, "Points de contrôle dans les systèmes répartis en environnement mobile," Thèse de magistère, USTHB, Algérie, 2003.

[7] H. Benkaouha, A. Abdelli, K. Bouyahia, and Y. Kaloune, "FDAN : Failure Detection protocol for mobile Ad hoc Networks," in *FGIT-FGCN (1)*, ser. Communications in Computer and Information Science, T.-H. Kim, A. C.-C. Chang, M. Li, C. Rong, C. Z. Patrikakis, and D. Slezak, Eds., vol. 119. Springer, 2010, pp. 85–94.

[8] H. Benkaouha, A. Abdelkrim, N. Badache, J. Ben-Othman, and L. Mokdad, "AFDAN : Accurate Failure Detection protocol for mANets," in *International Wireless Communications*

and Mobile Computing Conference, IWCMC 2015, Dubrovnik, Croatia, August 24-28, 2015. **IEEE, 2015, pp. 733–738. [Online]. Available : http ://dx.doi.org/10.1109/IWCMC.2015.7289174**

[9] H. Benkaouha, A. Abdelkrim, N. Badache, J. Ben-Othman, and L. Mokdad, "Towards improving failure detection in mobile ad hoc networks," in *2015 IEEE Global Communications Conference, GLOBECOM 2015, San Diego, CA, USA, December 6-10, 2015.* IEEE, 2015, pp. 1–6. [Online]. Available : http ://dx.doi.org/10.1109/GLOCOM.2014.7417004

[10] H. Benkaouha, L. Mokdad, and A. Abdelli, "2PACA : Two Phases Algorithm of Checkpointing for Ad hoc mobile networks," in *IWCMC*, R. Saracco, K. B. Letaief, M. Gerla, S. Palazzo, and L. Atzori, Eds. IEEE, 2013, pp. 1359–1364.

[11] H. Benkaouha, N. Badache, A. Abdelli, L. Mokdad, and J. Ben-Othman, "A novel checkpointing protocol for flat manets," *International Journal of Autonomous and Adaptive Communications Systems (IJAACS), Special Issue on : "FCST-12" Selected Topics in Computer Science and Technology*, 2016, In Press.

[12] H. Benkaouha, "Tolérance aux fautes dans le calcul distribué : Synthèse," Laboratoire LSI, USTHB, Algérie, Rapport Technique Interne LSI-TR-09-06, Octobre 2006.

[13] F. C. Gärtner, "Fundamentals of fault-tolerant distributed computing in asynchronous environments," *ACM Comput. Surv.*, vol. 31, no. 1, pp. 1–26, Mar. 1999. [Online]. Available : http ://doi.acm.org/10.1145/311531.311532

[14] A. Mostéfaoui, "Conception et expérimentation d'algorithmes pour la coopération répartie," Thèse de doctorat, Université de Rennes 1, France, 1994.

[15] C.-M. Lin and C.-R. Dow, "Efficient checkpoint-based failure recovery techniques in mobile computing systems," *J. Inf. Sci. Eng.*, vol. 17, no. 4, pp. 549–573, 2001.

[16] J. C. Laprie, "Sûreté de fonctionnement des systèmes : concepts de base et terminologie," *Revue de l'Electricité et de l'Electronique*, no. 11, pp. 95–105, December 2004.

[17] J. Arlat, Y. Crouzet, Y. Deswarte, J.-C. Fabre, J.-C. Laprie, and D. Powell, "Fault tolerance in distributed systems," in *Encyclopedia of Computer Science and Information Systems*, I.-W. E. Les Editions Vuibert, J.Akoka, Ed., 2006, ch. Part 1 : The Technological Dimension of Information Systems, Section 2., pp. 241–270.

[18] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, pp. 56–78, Feb. 1991. [Online]. Available : http ://doi.acm.org/10.1145/102792.102801

[19] V. Hadzilacos and S. Toueg, "Distributed systems (2nd ed.)," S. Mullender, Ed. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1993, ch. Fault-tolerant Broadcasts and Related Problems, pp. 97–145. [Online]. Available : http ://dl.acm.org/citation.cfm ?id=302430.302435

[20] M. C. Pease, R. E. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, 1980. [Online]. Available : http ://doi.acm.org/10.1145/322186.322188

[21] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982. [Online]. Available : http ://doi.acm.org/10.1145/357172.357176

[22] R. D. Schlichting and F. B. Schneider, "Fail-stop processors : An approach to designing fault-tolerant computing systems," *ACM Trans. Comput. Syst.*, vol. 1, no. 3, pp. 222–238, 1983. [Online]. Available : http ://doi.acm.org/10.1145/357369.357371

[23] F. B. Schneider, "Byzantine generals in action : Implementing fail-stop processors," *ACM Trans. Comput. Syst.*, vol. 2, no. 2, pp. 145–154, 1984. [Online]. Available : http ://doi.acm.org/10.1145/190.357399

[24] D. A. Rennels, "Fault-tolerant computing," in *Encyclopedia of Computer Science*. Chichester, UK : John Wiley and Sons Ltd., pp. 698–702. [Online]. Available : http ://dl.acm.org/citation.cfm ?id=1074100.1074394

[25] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso, "Database replication techniques : A three parameter classification," in *SRDS*, 2000, pp. 206–215. [Online]. Available : http ://www.computer.org/proceedings/srds/0543/05430206abs.htm

[26] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[27] G. Barigazzi and L. Strigini, "Application-transparent setting of recovery points," in *13th IEEE Symposium on Fault-Tolerant Computing*, June.

[28] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Software Eng.*, vol. 13, no. 1, pp. 23–31, 1987.

[29] P. Leu and B. K. Bhargava, "Concurrent robust checkpointing and recovery in distributed systems," in *Proceedings of the Fourth International Conference on Data Engineering, February 1-5, 1988, Los Angeles, California, USA*, 1988, pp. 154–163. [Online]. Available : http ://dx.doi.org/10.1109/ICDE.1988.105457

[30] K. M. Chandy and L. Lamport, "Distributed snapshots : Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.

[31] T.-H. Lai and T. H. Yang, "On distributed snapshots," *Inf. Process. Lett.*, vol. 25, no. 3, pp. 153–158, 1987.

[32] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *SRDS*, 1992, pp. 39–47.

[33] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985. [Online]. Available : http ://doi.acm.org/10.1145/3149.214121

[34] L. Temal and D. Conan, "Failure, connectivity and disconnection detectors," in *Actes des 1ères journées francophones Mobilité et Ubiquité 2004, UBIMOB'04, 2004, Nice / Sophia-Antipolis, France*, ser. ACM International Conference Proceeding Series, A. Dery-Pinna and A. Giboin, Eds. ACM, 2004, pp. 90–97. [Online]. Available : http ://doi.acm.org/10.1145/1050873.1050896

[35] N. Badache, "La mobilité dans les systèmes répartis," IRISA, Rennes, France, Rapport Technique, Publication Interne 962, Octobre 1995.

[36] B. R. Badrinath, A. Acharya, and T. Imielinski, "Impact of mobility on distributed computations," *Operating Systems Review*, vol. 27, no. 2, pp. 15–20, 1993. [Online]. Available : http ://doi.acm.org/10.1145/155848.155853

[37] T. Imielinski and B. R. Badrinath, "Mobile wireless computing : Challenges in data management," *Commun. ACM*, vol. 37, no. 10, pp. 18–28, 1994. [Online]. Available : http ://doi.acm.org/10.1145/194313.194317

[38] T. Imielinski and B. R. Badrinath, "Querying in highly mobile distributed environments," in *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings.*, L. Yuan, Ed. Morgan Kaufmann, 1992, pp. 41–52. [Online]. Available : http ://www.vldb.org/conf/1992/P041.PDF

[39] H. Labiod, *Réseaux mobiles ad hoc et réseaux de capteurs sans fil*. Lavoisier, 2006.

[40] J. P. Macker and M. S. Corson, "Mobile ad hoc networking and the IETF," *Mobile Computing and Communications Review*, vol. 3, no. 3, pp. 32–33, 1999. [Online]. Available : http ://doi.acm.org/10.1145/329124.329151

[41] G. Cao, "Designing efficient fault-tolerant systems on wireless networks," in *Proc. of the Third IEEE Information Survivability Workshop (ISW-2000)*, 2000.

[42] T. D. Nguyen and D. Conan, "Group membership tolerating failures and disconnections in mobile environments," in *New Technologies for Distributed Systems (NOTERE 2006)*, June 2006, pp. 01–14.

[43] M. Larrea, A. Fernández, and S. Arévalo, "Eventually consistent failure detectors," *J. Parallel Distrib. Comput.*, vol. 65, no. 3, pp. 361–373, 2005.

[44] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *J. ACM*, vol. 43, no. 4, pp. 685–722, 1996. [Online]. Available : http ://doi.acm.org/10.1145/234533.234549

[45] D. Conan, P. Sens, L. Arantes, and M. Bouillaguet, "Failure, disconnection and partition detection in mobile environment," in *Proceedings of The Seventh IEEE International Symposium on Networking Computing and Applications, NCA 2008, July 10-12, 2008, Cambridge, Massachusetts, USA*. IEEE Computer Society, 2008, pp. 119–127. [Online]. Available : http ://dx.doi.org/10.1109/NCA.2008.18

[46] P. Sens, L. Arantes, M. Bouillaguet, V. Simon, and F. Greve, "An unreliable failure detector for unknown and mobile networks," in *Principles of Distributed Systems, 12th International Conference, OPODIS 2008, Luxor, Egypt, December 15-18, 2008. Proceedings*, ser. Lecture Notes in Computer Science, T. P. Baker, A. Bui, and S. Tixeuil, Eds., vol. 5401. Springer, 2008, pp. 555–559. [Online]. Available : http ://dx.doi.org/10.1007/978-3-540-92221-6_39

[47] F. Greve, P. Sens, L. Arantes, and V. Simon, "Eventually strong failure detector with unknown membership," *Comput. J.*, vol. 55, no. 12, pp. 1507–1524, 2012.

[48] L. Arantes, P. Sens, G. Thomas, D. Conan, and L. Lim, "Partition participant detector with dynamic paths in mobile networks," in *NCA*. IEEE Computer Society, 2010, pp. 224–228.

[49] 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, Distributed Algorithms*, vol. 220, pp. 3–30, 1999. [Online]. Available : http ://www.sciencedirect.com/science/article/pii/S0304397598002357

[50] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, "Stable leader election," in *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, ser. Lecture Notes in Computer Science, J. L. Welch, Ed., vol. 2180. Springer, 2001, pp. 108–122. [Online]. Available : http ://dx.doi.org/10.1007/3-540-45414-4_8

[51] M. Raynal and F. Tronel, "Group membership failure detection : a simple protocol and its probabilistic analysis," *Distributed Systems Engineering*, vol. 6, no. 3, pp. 95–102, 1999. [Online]. Available : http ://dx.doi.org/10.1088/0967-1846/6/3/301

[52] A. Mostéfaoui, D. Powell, and M. Raynal, "A hybrid approach for building eventually accurate failure detectors," in *10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2004), 3-5 March 2004, Papeete, Tahiti*. IEEE Computer Society, 2004, pp. 57–65. [Online]. Available : http ://doi.ieeecomputersociety.org/10.1109/PRDC.2004.1276553

[53] M. U. Bhatti and D. Conan, "Détection de partition pour la gestion de groupes en environnement mobile," in *UbiMob*, ser. ACM International Conference Proceeding Series, J. Coutaz and S. Lecomte, Eds. ACM, 2005, pp. 65–72.

[54] R. Friedman and G. Tcharny, "Evaluating failure detection in mobile ad-hoc networks," *Int. J. Pervasive Computing and Communications*, vol. 5, no. 4, pp. 476–496, 2009. [Online]. Available : http ://dx.doi.org/10.1108/17427370911008857

[55] D. Alistarh, S. Gilbert, R. Guerraoui, and C. Travers, "Of choices, failures and asynchrony : The many faces of set agreement," *Algorithmica*, vol. 62, no. 1-2, pp. 595–629, 2012. [Online]. Available : http ://dx.doi.org/10.1007/s00453-011-9581-7

[56] A. Mostéfaoui, M. Raynal, and C. Travers, "Narrowing power vs efficiency in synchronous set agreement : Relationship, algorithms and lower bound," *Theor. Comput. Sci.*, vol. 411, no. 1, pp. 58–69, 2010. [Online]. Available : http ://dx.doi.org/10.1016/j.tcs.2009.09.002

[57] Y. Afek, E. Gafni, S. Rajsbaum, M. Raynal, and C. Travers, "The k-simultaneous consensus problem," *Distributed Computing*, vol. 22, no. 3, pp. 185–195, 2010. [Online]. Available : http ://dx.doi.org/10.1007/s00446-009-0090-8

[58] E. Jiménez, S. Arévalo, and A. Fernández, "Implementing unreliable failure detectors with unknown membership," *Inf. Process. Lett.*, vol. 100, no. 2, pp. 60–63, 2006.

[59] C. Delporte-Gallet, H. Fauconnier, E. Gafni, and P. Kuznetsov, "Impersonal failure detection," *CoRR*, vol. abs/1109.3056, 2011. [Online]. Available : http ://arxiv.org/abs/1109.3056

[60] J. B. Leners, H. Wu, W. Hung, M. K. Aguilera, and M. Walfish, "Detecting failures in distributed systems with the falcon spy network," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, T. Wobber and P. Druschel, Eds. ACM, 2011, pp. 279–294. [Online]. Available : http ://doi.acm.org/10.1145/2043556.2043583

[61] L. Lim and D. Conan, "An eventual alpha partition-participant detector for manets," in *2012 Ninth European Dependable Computing*

Conference, Sibiu, Romania, May 8-11, 2012, C. Constantinescu and M. P. Correia, Eds. IEEE, 2012, pp. 25–36. [Online]. Available : http ://doi.ieeecomputersociety.org/10.1109/EDCC.2012.15

[62] R. Guerraoui, V. Hadzilacos, P. Kuznetsov, and S. Toueg, "The weakest failure detectors to solve quittable consensus and nonblocking atomic commit," *SIAM J. Comput.*, vol. 41, no. 6, pp. 1343–1379, 2012. [Online]. Available : http ://dx.doi.org/10.1137/070698877

[63] A. Mostéfaoui, M. Raynal, and J. Stainer, "Chasing the weakest failure detector for k-set agreement in message-passing systems," in *11th IEEE International Symposium on Network Computing and Applications, NCA 2012, Cambridge, MA, USA, August 23-25, 2012*. IEEE Computer Society, 2012, pp. 44–51. [Online]. Available : http ://dx.doi.org/10.1109/NCA.2012.19

[64] M. Larrea, A. Fernández, and S. Arévalo, "Eventually consistent failure detectors," in *SPAA*, 2001, pp. 326–327. [Online]. Available : http ://doi.acm.org/10.1145/378580.378747

[65] M. K. Aguilera, W. Chen, and S. Toueg, "Heartbeat : A timeout-free failure detector for quiescent reliable communication," in *WDAG*, ser. Lecture Notes in Computer Science, M. Mavronicolas and P. Tsigas, Eds., vol. 1320. Springer, 1997, pp. 126–140.

[66] A. Lafuente, M. Larrea, I. S. Arriola, and R. Cortiñas, "Communication-optimal eventually perfect failure detection in partially synchronous systems," *J. Comput. Syst. Sci.*, vol. 81, no. 2, pp. 383–397, 2015. [Online]. Available : http ://dx.doi.org/10.1016/j.jcss.2014.06.010

[67] G. Cao and M. Singhal, "Mutable checkpoints : A new checkpointing approach for mobile computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 2, pp. 157–172, 2001.

[68] E. N. . M. Elnozahy, L. Alvisi, Y. min Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," 1996.

[69] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. 1, no. 2, pp. 221–232, 1975.

[70] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[71] B. K. Bhargava and S.-R. Lian, "Independent checkpointing and concurrent rollback for recovery in distributed systems - an optimistic approach," in *SRDS*, 1988, pp. 3–12.

[72] Y.-M. Wang, "Reducing message logging overhead for log-based recovery," in *ISCAS*, 1993, pp. 1925–1928.

[73] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. M. R. Kintala, "Checkpointing and its applications," in *FTCS*. IEEE Computer Society, 1995, pp. 22–31.

[74] R. Prakash and M. Singhal, "Low-cost checkpointing and failure recovery in mobile computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 10, pp. 1035–1048, 1996.

[75] S. K. Gupta, R. K. Chauhan, and P. Kumar, "A minimum-process coordinated checkpointing protocol for mobile computing systems," *Int. J. Found. Comput. Sci.*, vol. 19, no. 4, pp. 1015–1038, 2008.

[76] B. Gupta, S. Rahimi, R. A. Rias, and G. Bangalore, "A low-overhead non-block checkpointing algorithm for mobile computing environment," in *GPC*, ser. Lecture Notes in Computer Science, Y.-C. Chung and J. E. Moreira, Eds., vol. 3947. Springer, 2006, pp. 597–608.

[77] R. Tuli and P. Kumar, "Minimum process coordinated checkpointing scheme for ad hoc networks," *CoRR*, vol. abs/1111.2208, 2011.

[78] D. Manivannan and M. Singhal, "A low overhead recovery technique using quasi-synchronous checkpointing," in *ICDCS*, 1996, pp. 100–107.

[79] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A distributed domino-effect free recovery algorithm," in *Symposium on Reliability in Distributed Software and Database Systems*, 1984, pp. 207–215.

[80] A. Acharya and B. R. Badrinath, "Checkpointing distributed applications on mobile computers," in *PDIS*. IEEE Computer Society, 1994, pp. 73–80.

[81] F. Quaglia, B. Ciciani, and R. Baldoni, "A checkpointing-recovery scheme for domino-free distributed systems," in *Fault-Tolerant Parallel and Distributed Systems*. Springer, 1998, pp. 93–107.

[82] M. Ono and H. Higaki, "Consistent checkpoint protocol for wireless ad-hoc networks," in *PDPTA*, H. R. Arabnia, Ed. CSREA Press, 2007, pp. 1041–1046.

[83] G. Cao and M. Singhal, "On the impossibility of min-process non-blocking checkpointing and an efficient checkpointing algorithm for mobile computing systems," in *1998 International Conference on Parallel Processing (ICPP '98), 10-14 August 1998, Minneapolis, Minnesota, USA, Proceedings*. IEEE Computer Society, 1998, pp. 37–44. [Online]. Available : http ://dx.doi.org/10.1109/ICPP.1998.708461

[84] D. Manivannan and M. Singhal, "Quasi-synchronous checkpointing : Models, characterization, and classification," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 7, pp. 703–713, 1999. [Online]. Available : http ://doi.ieeecomputersociety.org/10.1109/71.780865

[85] G. Li and L. Shu, "Design and evaluation of a low-latency checkpointing scheme for mobile computing systems," *Comput. J.*, vol. 49, no. 5, pp. 527–540, 2006. [Online]. Available : http ://dx.doi.org/10.1093/comjnl/bxk004

[86] P. K. Jaggi and A. K. Singh, "Message efficient global snapshot recording using a self stabilizing spanning tree in a manet," *International Journal of Communication Networks and Information Security (IJCNIS)*, vol. 3, no. 3, pp. 247–255, 2011.

[87] A. K. Singh and P. K. Jaggi, "Asynchronous rollback recovery in cluster based multi hop mobile ad hoc networks," *International Journal of Enhanced Research in Management and Computer Applications (IJERMCA)*, vol. 2, no. 4, pp. 46–55, 2013.

[88] "The network simulator - ns2," http ://www.isi.edu/nsnam/ns/, [Online ; accessed 22-December-2013].

[89] D. B. Johnson and D. A. Maltz, "Truly seamless wireless and mobile host networking. protocols for adaptive wireless and mobile networking," *IEEE Personal Commun.*, vol. 3, no. 1, pp. 34–42, 1996. [Online]. Available : http ://dx.doi.org/10.1109/98.486974

[90] S. Moussaoui, M. Guerroumi, and N. Badache, "Data replication in mobile ad hoc networks," in *MSN*, ser. Lecture Notes in Computer Science, J. Cao, I. Stojmenovic, X. Jia, and S. K. Das, Eds., vol. 4325. Springer, 2006, pp. 685–697.

[91] S. Moussaoui, M. Guerroumi, and N. Badache, "Two phase replication approach for manets," *IJAHUC*, vol. 4, no. 5, pp. 292–303, 2009.

[92] A. Rachedi, S. Lohier, S. Cherrier, and I. Salhi, "Wireless network simulators relevance compared to a real testbed in outdoor and indoor environments," *IJAACS*, vol. 5, no. 1, pp. 88–101, 2012.

[93] C. E. Perkins and E. M. Belding-Royer, "Ad-hoc on-demand distance vector routing," in *WMCSA*. IEEE Computer Society, 1999, pp. 90–100.

[94] H. Benkaouha, L. Mokdad, and A. Abdelli, "SAN-based modeling of fault tolerant protocols for manets," in *IEEE International Conference on Communications, ICC 2014, Sydney, Australia, June 10-14, 2014*. IEEE, 2014, pp. 336–341. [Online]. Available : http ://dx.doi.org/10.1109/ICC.2014.6883341

[95] B. Plateau and J.-M. Fourneau, "A methodology for solving markov models of parallel systems," *J. Parallel Distrib. Comput.*, vol. 12, no. 4, pp. 370–387, 1991.

[96] B. Plateau and K. Atif, "Stochastic automata network for modeling parallel systems," *IEEE Trans. Software Eng.*, vol. 17, no. 10, pp. 1093–1108, 1991.

[97] L. Mokdad and J. Ben-Othman, "Stochastic automata networks for modelling scheduling scheme in wimax networks," *Journal of Interconnection Networks*, vol. 10, no. 4, pp. 481–495, 2009.

[98] L. Mokdad and J. Ben-Othman, "Admission control mechanism and performance analysis based on stochastic automata networks formalism," *J. Parallel Distrib. Comput.*, vol. 71, no. 4, pp. 594–602, 2011.

[99] J. Clément, C. Delporte-Gallet, H. Fauconnier, and M. Sighireanu, "Guidelines for the verification of population protocols," in *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*. IEEE Computer Society, 2011, pp. 215–224. [Online]. Available : http ://dx.doi.org/10.1109/ICDCS.2011.36

[100] F. Greve, L. Arantes, and P. Sens, "What model and what conditions to implement unreliable failure detectors in dynamic networks ?" in *Workshop on Theoretical Aspects on Dynamic Distributed Systems, TADDS '11, Rome, Italy, September 19, 2011*, R. Baldoni and A. A. Shvartsman, Eds. ACM, 2011, pp. 13–17. [Online]. Available : http ://doi.acm.org/10.1145/2034640.2034645

[101] "Peps - performance evaluation of parallele programs," http ://www-id.imag.fr/Logiciels/peps/, [Online ; accessed 27-September-2013].