UNIVERSITE DES SCINECES ET DE TECHNOLOGIE -MOHAMMED BOUDIAF- ORAN

N° attribué par la bibliothèque
|_|_|_|_|_|_|_|_|_|_|

THÈSE

pour obtenir le grade de

Docteur de l'Université de thèse

Spécialité :

préparée au laboratoire

dans le cadre de l'École Doctorale

présentée et soutenue publiquement
par

REBBAH Mohammed

le juin 2014

Titre:
Tolérance aux fautes dans les grilles de calcul

Directeur de thèse: **X**
Co-directeur de thèse: **Y**

Jury

M. Z,	Président du jury
M. U,	Rapporteur
M. V,	Rapporteur
M. W,	Examinateur
M. X,	Directeur de thèse
M. Y,	Co-directeur de thèse

// *Remerciements*

Un grand merci à mes directeurs de thèse, le Professeur Adbelkader BENYETTOU, merci d'avoir fait confiance en nous "les gars de Mascara" et merci de votre disponibilité et de vos encouragements, et le Professeur Yahya SLIMANI, merci d'avoir cru en moi dès le début et d'avoir pris le temps de m'expliquer, de m'apprendre c'est quoi la recherche... J'ai beaucoup appris à tes côtés. Merci de m'avoir fait découvrir les plaisirs du travail bien fait. Je tiens également à te remercier pour ton écoute et pour tous les conseils que tu m'as donnés sur bien des sujets. Merci de me donner l'exemple d'un homme de science, Merci pour tout ...

J'exprime également ma profonde gratitude à Madame le Professeur BENAMRANE Nacera pour avoir accepté de présider mon jury de thèse. Je remercie tout particulièrement Monsieur le Professeurs YAGOUBI Belabbas, Doyen à la Faculté des Sciences, Université d'Oran, Madame FIZAZI Hadria, Professeur à l'Université des Sciences et de la Technologie d'Oran, Mohamed BOUDIAF et Monsieur AMAR BENSABER Djamel, Maître de Conférences et Directeur des études à l'ENSI Sidi Belabbess pour l'honneur qu'ils me font en participant à mon Jury de thèse.

J'adresse aussi mes plus sincères remerciements à Monsieur le Professeur Lionel BRUNIE, responsable de l'équipe DRIM au sein du laboratoire LIRIS à INSA Lyon, pour m'avoir reçu au sein de son équipe, pour son soutien moral ainsi que pour la très bonne ambiance que j'ai toujours trouvée au sein l'équipe DRIM. Je remercie approfondiment GHERAISSA Mabrouka pour son aide durant tous mes stage à Lyon.

Je remercie également mes amis de Mascara, DEBAKLA Mohammed, MOKHTARI Chakir et SMAIL Omar, ainsi que BENDOUKHA Hayat. Avec eux j'ai passé de très bons moments depuis le Magister et pendant le parcours de thèse.

J'ai eu le plaisir d'encadrer les étudiants à l'Université de Mascara, ceux qui ont travaillé sur les grilles ou sur les sujets de tolérance aux fautes, je remercie KHALDI Miloud, BOURASI Med Fouad, AMRANI Asmaa, NESNAS Fatima Zohra, BAKBAK Amina, BEGUENINE Fethi, SMAIL

Sofiane, LAROUCI Abdellah, SETAOUTI Samir, BENSMIRA Mohammed Ramzi ...

J'adresse également mes plus sincères remerciements à ma famille, mon père et ma mère, mes frères, mes sœurs, mon cher neveu "ABDELHAK" et ma très chère nièce ARBA.

Enfin, merci à celle qui me donne l'envie, la joie et le soutien d'évoluer. Merci à toi Ma femme pour tout ... Amina, Imene, Youness et Yasmine seront fières de nous pour tout ce qu'on fait ensemble pour eux.

Mohammed REBBAH

0.1 molakhass

Résumé

L'objectif de cette thèse est de proposer des techniques de tolérance aux fautes pour les grilles de calcul. Elles généralisent la problématique soulevée par le développement de trois techniques de tolérance aux fautes, leurs implémentations et leurs évaluations sous une Desktop Grid ou par des simulations. Comme première contribution, nous proposons une amélioration du modèle G/S/M par un modèle hiérarchique dynamique et son implémentation, sous forme d'un service de grille, sous le middleware Globus. Puis, la proposition d'un modèle hiérarchique tolérant aux fautes pour une grille mobile. La deuxième contribution, qui est à la fois majeure et originale, concerne la proposition de deux modèles décentralisés de tolérance aux fautes pour les grilles. Ces modèles se basent sur un nouveau type de graphe que nous avons défini, à savoir les graphes colorés dynamiques. Grâce à ces graphes, nous arrivons à définir, de manière abstraite et formelle, les caractéristiques fondamentales des grilles de calcul, comme la dynamicité, l'hétérogénéité et le passage à l'échelle. Nous utilisons un jeu de couleurs, pour colorier les sommets de ces graphes, en tenant compte des caractéristiques physiques et logiques des nœuds d'une grille. Ces couleurs (qui sont susceptibles de changer au cours du temps), nous ont permis d'une part, de représenter la dynamicité des grilles et de mettre en place des techniques de tolérance aux fautes, d'autre part. Dans la troisième contribution, nous proposons un modèle fiable de tolérance aux fautes basé sur une migration périodique des tâches vers le substitut le plus fiable.
Mots clés : Grille de calcul, Tolérance aux fautes, Graphe dynamique coloré, Performance, Fiabilité Globus.

Abstract

The objective of this thesis is to provide fault tolerance techniques for grid computing. They generalize the issues raised by the development of three fault-tolerance techniques, their implementations and evaluations in a Desktop Grid or by simulations. As a first contribution, we propose an improved G/S/M model by a dynamic hierarchical model and its implementation as a grid service over the Globus middleware. Then, a fault tolerant hierarchical model for mobile grid is proposed. The second contribution, which is both a major and original, it concerns the proposal of two decentralized models of fault tolerance for grids. These models are based on a new type of graph that we have defined, namely dynamic colored graphs. With these graphs, we can define an abstract and formal model, the basic characteristics of grid computing, such as dynamicity, heterogeneity and scalability. We use a set of colors to color the vertices of these graphs, taking into account the physical and logical nodes of a grid. These colors (which are likely to change over time), enabled us firstly to represent dynamicity of grid computing and implement fault tolerance techniques. In the third contribution, we propose a reliable fault tolerant model based on a periodic migration of jobs to the most reliable substitute.

Keywords : Grid Computing, Fault tolerance, Dynamic Colored Graph, Performance, Reliability, Globus.

Table des matières

0.1 molakhass iv
Résumé v
Abstract vi
Table des matières vii
Table des figures xiii
Liste des tableaux xv

Introduction générale **1**
1 Problématique 1
2 Objectifs de la thèse 2
3 Contributions 2
4 Structure du manuscrit de thèse 4

1 Grille de calcul : Etat de l'art **7**
1 Introduction 7
2 Systèmes distribués 7
2.1 Avantages de la distribution 8
2.2 Défis de la distribution 9
3 Modèle client/serveur 9
4 Système Pair à pair 9
4.1 Caractéristiques 10
4.2 Avantages et inconvénients du modèle P2P 10
5 Grappes de calcul (Cluster) 10
5.1 Définition et caractéristiques d'un cluster 11
5.2 Catégories de cluster 11
5.2.1 Clusters de haute disponibilité 11
5.2.2 Clusters de haute performance 12
5.3 Exemples de clusters 12
5.3.1 Beowulf 12
5.3.2 MOSIX 12
6 Grille de calcul 13
6.1 Définition 13
6.2 Caractéritiques d'une grille 14
6.3 Architecture d'une grille de calcul 14
6.4 Topologies d'une grille de calcul 15
6.5 Cadres d'utilisation de grille 16

6.6 Grille de données 17
6.7 Normes et grilles 18
6.8 Outils d'expérimentations des grilles 18
6.8.1 Middlewares 19
6.8.2 Simulateurs de grille 20
7 Grille mobile 20
8 Grille pervasive 21
8.1 Définition 21
8.2 Architecture générale d'une grille pervasive 21
9 Conclusion 21

2 Tolérance aux fautes dans les grilles de calcul 23
1 Introduction 23
2 Sûreté de fonctionnement 23
3 Taxonomie des fautes 25
4 Tolérance aux fautes dans les systèmes distribués 26
4.1 Etapes de la tolérance aux fautes 26
4.2 Techniques de détection des fautes 27
4.2.1 Cas des systèmes synchrones/asynchrones 27
4.2.2 Messages "Ping/Pong" 27
4.2.3 Echanges de messages de vie (*heartbeats*) 27
4.3 Techniques de tolérance aux fautes 28
4.3.1 Tolérance aux fautes par duplication 28
4.3.2 Technique *Rollback-Recovery* 30
5 Tolérance aux fautes dans les grilles de calcul 31
5.1 Modèle de fautes pour les grilles de calcul 31
5.1.1 Fautes franches 33
5.1.2 Fautes par omission 33
5.1.3 Fautes byzantines 33
5.1.4 Fautes des middlewares 34
5.2 Analyse des défaillances dans les grilles de calcul 34
5.3 Détection de fautes dans les grilles de calcul 35
5.3.1 Limites des méthodes actuelles de détection de fautes 35
5.3.2 Méthodes de détection de fautes scalables 36
5.3.3 Difficultés de détection de fautes 40
5.4 Techniques de tolérance aux fautes dans les grilles de calcul . . 40
5.4.1 Checkpoint et recouvrement 41
5.4.2 Réplication des ressources d'une grille 42
5.4.3 Tolérance aux fautes par ordonnancement 43
5.4.4 Tolérance aux fautes pour les applications de type "Diviser pour régner" 44
6 Conclusion 45

3 Modèles hiérarchiques de tolérance aux fautes 47
1 Introduction 47
2 Modèle hiérarchique dynamique 47

2.1 Modèle proposé 48
2.2 Notion de famille 49
2.3 Types de fautes traitées 50
2.4 Détection de fautes 51
2.5 Tolérance aux fautes 51
2.5.1 Méthode de distribution 51
2.5.2 Méthode de remplacement 52
2.6 Restructuration de l'arborescence 54
2.7 Scénarios d'exécution 54
2.7.1 Niveau N_1 : gestionnaire des feuilles 55
2.7.2 Niveau intermédiare N_i 55
2.7.3 Niveau Racine 56
2.8 Discussion 56
3 Modèle multi-arborescent pour une grille mobile tolérante aux fautes 56
3.1 Rôle des dispositifs mobiles dans une grille mobile 57
3.1.1 Dispositifs mobiles comme consommateurs de ressources 57
3.1.2 Dispositifs mobiles comme fournisseurs de ressources 58
3.2 Modèle de fautes dans un environnement mobile 59
3.3 Modes de fonctionnement des dispositifs mobiles 59
3.4 Architecture proposée pour une grille mobile 60
3.4.1 Fonctionnement du modèle 60
3.4.2 Acteurs du modèle de grille mobile 61
3.4.3 Niveaux du modèle 62
3.5 Modèle hiérarchique associé à une grille mobile : le modèle $G/I/S$ 62
3.5.1 Détection d'un nouveau DM 63
3.5.2 Déconnexion d'un DM 63
3.5.3 Déconnexion d'un INT 64
3.6 Tolérance aux fautes 65
3.6.1 Tolérance au niveau interlocuteur 65
3.6.2 Tolérance au niveau gestionnaire de groupe 66
3.7 Discussion 66
4 Conclusion 68

4 Modèles de tolérance aux fautes basés sur les graphes colorés dynamiques 69
1 Introduction 69
2 Définitions 70
2.1 Terminologie 70
2.2 Graphe dynamique 70
2.2.1 Notion de base temporelle 70
2.2.2 Processus d'évolution 72
2.3 Coloration des graphes 72
2.3.1 Définition 72
2.3.2 Coloration des sommets 73

2.3.3 Coloration des arêtes 73
3 Graphe coloré dynamique 73
4 Modèle de tolérance aux fautes basé sur les niveaux de performance . 74
4.1 Environnement du système de grille 74
4.2 Modélisation de grille par un graphe coloré dynamique 75
4.3 Règle de coloration des sommets 75
4.4 Transmission du vecteur d'état 76
4.5 Tolérance aux fautes 76
4.5.1 Calcul des substituts 77
4.5.2 Sélection des substituts 77
4.5.3 Types de dynamicité 78
4.6 Discussion 78
5 Modèle de degré de tolérance 79
5.1 Règle de coloration 79
5.2 Calcul du degré de tolérance 80
5.3 Protocole de stabilisation 80
5.4 Formulation mathématique du modèle 81
5.4.1 Etat des sommets du graphe 81
5.4.2 Etats du graphe 82
5.5 Tolérance aux fautes 83
5.5.1 Type de fautes traitées 84
5.5.2 Modèle de tolérance aux fautes par distribution sur les voisins collaborateurs 84
5.5.3 Modèle de tolérance aux fautes par désignation d'un remplaçant 85
5.6 Discussion 85
6 Conclusion 86

5 Modèle fiable de tolérance aux fautes 87
1 Introduction 87
2 Migration des tâches pour la tolérance aux fautes 88
3 Analyse de fiabilité pour un service de grille 90
3.1 Modèle abstrait de RMS 90
3.2 Modèle de grille 91
3.3 Modélisation d'une grille de calcul par un graphe 92
3.4 Le temps médian de réparation (MDTTR) 93
3.5 Fiabilité des services de grille 94
4 Technique de tolérance aux fautes 97
4.1 Détection de fautes 98
4.2 Stratégie de tolérance aux fautes 98
4.3 Tolérance aux fautes d'un nœud de travail 99
4.4 Tolérance aux fautes du nœud 100
5 Illustration numérique 100
6 Conclusion 102

6 Expérimentations **105**
1 Introduction . . . 105
2 Environnement d'évaluation . . . 105
3 Modèle hiérarchique dynamique . . . 106
3.1 Implémentation . . . 106
3.2 Architecture de service de DHM-FTGrid . . . 106
3.3 Architecture de la grille de test . . . 107
3.4 Expérimentations . . . 108
4 Modèle multi-arborescent pour une grille mobile tolérante aux fautes 109
4.1 Modèle de simulation . . . 109
4.2 Résultats et interprétations . . . 111
4.2.1 Détection des fautes des subordonnés . . . 111
4.2.2 Tolérance aux fautes des INT . . . 111
5 Modèle des substituts . . . 112
5.1 Evaluation et performances . . . 112
5.2 Paramètres observés . . . 112
5.3 Résultats relatifs au modèle basé sur un graphe coloré dynamique . . . 113
5.3.1 Classes des substituts . . . 114
5.3.2 Taux de dynamicité . . . 114
5.3.3 Transfert du vecteur d'état . . . 114
5.4 Résultats relatifs aux performances de la grille . . . 116
5.4.1 Politiques de comparaison . . . 119
5.4.2 Résultats de simulation . . . 119
6 Modèle de degré de tolérance . . . 123
6.1 Influence du degré de tolérance . . . 124
6.1.1 Phase de coloration . . . 124
6.1.2 Phase de stabilisation . . . 125
6.2 Tolérance aux fautes . . . 125
7 Modèle fiable de tolérance aux fautes . . . 126
7.1 Métriques de performance . . . 127
7.2 Algorithmes de référence . . . 127
7.3 Résultats expérimentaux . . . 128
8 Conclusion . . . 130

Conclusion et Perspectives **131**

Publications de l'auteur **133**

Bibliographie **135**

Table des figures

1.1 Architecture d'une grille de calcul [62] 15
1.2 Outils d'expérimentations sur les grilles [30]. 19

2.1 Arbre de la sûreté de fonctionnement [116] 25
2.2 Duplication active [199] 28
2.3 Duplication passive 29
2.4 Duplication hybride 30
2.5 Probabilité de dé faillance dans les environnements parallèles ré, où λ=0.0005 [91] 32
2.6 Exemple de détection de fautes par les invariants du graphe [182] . . 39

3.1 Modèle hiérarchique dynamique de grille 49
3.2 Famille du nœud $N_{i,k,L}$ 50
3.3 Tolérance aux fautes en appliquant la méthode de distribution 52
3.4 Remplacement d'un nœud défaillant qui appartient à un niveau intermédiaire dans l'arborescence 53
3.5 Modes de fonctionnement des dispositifs mobiles [152] 60
3.6 Environnement d'une grille mobile 61
3.7 Architecture d'une grille mobile 61
3.8 Architecture hiérarchique du modèle d'une grille mobile 63
3.9 Schéma décrivant la détection d'un nouveau dispositif 64
3.10 Schéma de déconnexion et de déplacement d'un dispositif mobile . . . 64
3.11 Tolérance aux fautes locale (au niveau interlocuteur) 65
3.12 Tolérance aux fautes vers le niveau supérieur (au niveau interlocuteur) 66
3.13 Tolérance aux fautes des interlocuteurs (au niveau gestionnaire de groupe) 67
3.14 Tolérance aux fautes des dispositifs mobiles (au niveau gestionnaire de groupe) 67

4.1 Exemples de graphes colorés dynamiques 74
4.2 Graphe coloré dynamique avec 5 attributs 76
4.3 Graphe coloré dynamique (25 sommets, 45 arcs avec α=3). 79
4.4 Graphe après coloration 80
4.5 Graphe après stabilisation 80
4.6 Différents états, en terme de couleurs, des des sommets du graphe . . 81
4.7 Graphe coloré dynamique (8 sommets, 18 arcs et α=2). $Etat_t(G) = VB$. 82
4.8 Différents états du graphe 83

5.1 Architecture en couches d'un RMS 92
5.2 Modèle de grille 93
5.3 Temps de répartition d'un nœud défaillant selon le MDTTR comme période de temps 98
5.4 Influence du volume de données échangé sur la fiabilité du service de grille 102
5.5 Influence du temps d'exécution du processus sur la fiabilité du service de grille 102

6.1 Architecture du service DHM-FTGrid 107
6.2 Archtecture de la grille de test 8/4/2 107
6.3 Tolérance aux fautes par distribution et remplacement 108
6.4 Différents niveaux de tolérance 109
6.5 Architecture de Simobgrid 109
6.6 Environnement de simulation 110
6.7 Nombre des différents substituts (moins performants, identiques et plus performants) selon le nombre de nœuds 114
6.8 Variation des type de substituts pour une dynamicité discontinue et continue 115
6.9 Régions pour différentes valeurs du paramètre h (saut) 115
6.10 Temps de transfert du vecteur d'état pour m=10, h=1, 5, 10, 15 et 20 116
6.11 Temps de transfert du vecteur d'état pour m=40, h=1, 5, 10, 15 et 20 117
6.12 Temps de transfert du vecteur d'état pour h=10, m=10, 15, 20, 25 et 30 117
6.13 Temps de transfert du vecteur d'état pour h=10, m=20, 40, 60, 80 and 100 118
6.14 Cas 1 : Temps de réponse moyen (1000 nœuds, 10000 jobs) 120
6.15 Cas 1 : Temps d'attente moyen (1000 nœuds, 10000 jobs) 121
6.16 Cas 1 : Temps de réponse moyen (4000 nœuds, 60000 jobs) 121
6.17 Cas 1 : Temps d'attente moyen (4000 nœuds, 60000 jobs) 121
6.18 Cas 2 : Temps de réponse moyen (4000 nœuds, 60000 jobs) 122
6.19 Cas 2 : Temps d'attente moyen (4000 nœuds, 60000 jobs) 122
6.20 Cas 3 : Coût de migration des données (500 Mo) 123
6.21 Cas 3 : Coût de migration des données (10 Go) 123
6.22 Tolérance aux fautes des nœuds par deux méthodes (Remplacement et Distribution) 126
6.23 Le temps de réponse moyen 128
6.24 Evaluation du temps d'attente moyen 129
6.25 Fraction des tâches migrées 129

Liste des tableaux

4.1 Quelques éléments de la théoire des graphes 71
4.2 Différents états d'un graphe avec les phases d'apparition associées . . 82

5.1 Temps de réparation en fonction de la cause de la panne [15] 94
5.2 Liste des symboles 95
5.3 Attributs des nœuds et des liens 101
5.4 Attributs des tâches 101

6.1 Différents services de Globus 106
6.2 Résultats relatifs à la détection des fautes des subordonnés 111
6.3 Tolérance aux fautes des interlocuteurs 112
6.4 Propriétés des nœuds de la grille (S : Statique, D : Dynamique) . . . 113
6.5 Paramètres de la grille de calcul 118
6.6 Coloration des sommets par trois méthodes de calcul du degré de tolérance. 124
6.7 Phase de stabilisation par trois méthodes (M1, M2 et M3). 125
6.8 Paramètres de configuration de la grille de calcul 127

Introduction générale

1 Problématique

L'augmentation constante des besoins en terme de puissance de calcul informatique a toujours été un défi auquel la communauté informatique a été (et est toujours) confrontée. Même si les évolutions technologiques de ces dernières années ont permis de mettre en place des machines de plus en plus puissantes, ces dernières restent très insuffisantes par rapport aux besoins en terme de calcul. Partant de cette constatation, les solutions se sont orientées vers une approche qui consiste à regrouper le plus grand nombre de machines (ou systèmes) distribuées pour fournir une puissance de calcul très élevée. C'est cette approche qui est à l'origine des grilles de calcul ou *"Grid Computing"*. L'idée centrale est de pouvoir mutualiser les ressources de machines individuelles, pour les mettre à la disposotion des utilisateurs. Une grille de calcul est une infrastructure particulièrement complexe, car elle est composée de plusieurs milliers de machines très hétérogènes et réparties géographiquement sur une très large échelle. Cette complexité est accentuée par le très fort taux de dynamicité qui caractérise les grilles, dans le sens où de nouvelles ressources peuvent être ajoutées ou retirées à tout moment, de la grille soit à la suite de pannes ou d'arrêts volontaires. Du point de vue utilisation, ces différents niveaux de complexité sont rendus transparents aux utilisateurs grâce à l'existence de middlewares dédiés aux grilles, tels que Globus, gLite, etc.
Du point de vue exploitation, la gestion des ressources d'une grille nécessite de mettre en place des techniques de tolérance aux fautes pour assurer une certaine sûreté de fonctionnement, ce qui permet aux utilisateurs d'avoir un niveau de confiance dans les grilles qu'ils utilisent. Si le problème de tolérance aux fautes a été très largement étudié pour les systèmes informatiques classiques (centralisés, parallèles, distribués), il prend une autre dimension, en terme de complexité, dans le cas des grilles. Ce niveau de complexité supplémentaire vient de trois facteurs essentiels : (i) le fort taux d'hétérogénéité des ressources ; (ii) leur très large distribution ; (iii) le fort taux de dynamicité des ressources. Comme il est très difficile (voir impossible) de concevoir un système qui puisse fonctionner sans présence de fautes matérielles et logicielles, il est donc nécessaire de mettre en place un système de tolérance aux fautes qui puisse offrir un certain nombre de propriétés liées à la sûreté de fonctionnement des systèmes. Un tel système revêt une importance particulière dans le cas des grilles, dans le sens où les ressources d'une grille sont le résultat d'une mutualisation. C'est dans ce cadre général que se situe cette thèse, qui se propose d'étudier le problème

de tolérance aux fautes dans les grilles et de définir un certain nombre de techniques pour le résoudre.

2 Objectifs de la thèse

Nos travaux de recherche portent sur la proposition et la mise en oeuvre de techniques de tolérance aux fautes, qui tiennent compte des caractéristiques intrinsèques des grilles de calcul. A travers ces travaux de recherche, nous essayons d'atteindre les objectifs suivants :

- Notre premier objectif concerne la mise en place d'un mécanisme de tolérance aux fautes sous Globus. Pour cela, nous avons développé et mis en œuvre un service générique de tolérance aux fautes, sous Globus, dédié aux grilles de calcul, appelé *G-Tolere*. Ce service a été ensuite exploité pour définir un modèle hiérarchique dynamique de tolérance aux fautes, appelé *DHM-FTGrid*, toujours sous Globus.
- Le deuxième objectif de cette thèse, est d'intégrer des mécanismes de tolérance aux fautes dans les grilles mobiles. Pour cela, nous avons proposé un modèle multi-arborescent pour une grille mobile, appelé *FTPGRID*.
- A travers le troisième objectif, nous cherchons à définir un modèle d'abstraction des grilles de calcul, qui puisse tenir compte des trois propriétés de base des grilles, à savoir l'hétérogénéité, la dynamicité et le passage à l'échelle. La majorité des modèles actuels ne regroupent pas ces trois propriétés dans un seul modèle. C'est ainsi que nous avons défini un nouveau type de graphe, que nous avons appelé *Graphe Coloré Dynamique*, où les ressources sont modélisées par des sommets et les arcs représentent les connexions entre ces ressources. Grâce à un jeu de couleurs, nous arrivons à modéliser l'hétérogénéité des ressources (couleurs différentes), ce qui permettra par la suite de choisir les ressources les plus appropriées en cas de panne et ce en fonction de leurs couleurs. La structure de graphe nous permet également de prendre en charge la dynamicité d'une grille, et ce par l'ajout ou la suppression de sommets et d'arcs. Partant de cette structure de graphe coloré dynamique, nous avons élaboré un certain nombre de techniques de tolérance aux fautes.
- Le quatrième objectif cherche dans la fiabilité des substituts d'un nœud défaillant. Nous avons proposé une technique de tolérance aux fautes basée sur une migration périodique des tâches du nœud défaillant vers les substituts les plus fiables.

3 Contributions

Les principales contributions de cette thèse, peuvent se résumer comme suit à travers quatre axes :

1. **Modèles hiérarchiques de tolérance aux fautes sous Globus :** A travers un service générique de tolérance aux fautes dédié aux grilles de calcul, que nous avons appelé *G-Tolere*. Ce service permet de prendre en charge trois types

de fautes : les fautes de type arrêt, les fautes de déconnexion et les fautes liées aux qualités de service (QoS). Ce service a été développé sous Globus autour de services de base tels que GRAM, MDS, GSI et GridFTP. Les composants de ce service permettent de détecter et de tolérer les fautes dans une grille selon le modèle hiérarchique G/S/M, proposé par Yagoubi et Slimani [221]. Nous avons ensuite amélioré ce service par un modèle de grille hiérarchique dynamique, transformant toute grille en une arborescence n-aires, contenant un nombre de niveaux variables. Sur ce modèle, nous avons défini et mis en place un mécanisme de tolérance aux fautes basé sur deux techniques : la distribution et le remplacement.

2. **Modèles de tolérance aux fautes pour des grilles spécifiques :** Dans cet axe, nous nous sommes intéressés à l'introduction de la notion de mobilité dans les grilles de calcul. Grâce à cette mobilité, nous permettons d'élargir le champ d'utilisation des grilles (accès à partir d'un smartphone par exemple), mais nous nous confrontons à de nouveaux problèmes liés à ces dispositifs mobiles. Parmi ces problèmes, nous pouvons citer la connectivité intermittente, l'hétérogénéité des dispositifs mobiles, la sécurité, etc. De plus, un environnement de calcul mobile est caractérisé par une disponibilité limitée de ressources, une mobilité élevée, une largeur de bande passante très réduite et des déconnexions fréquentes. Toutes ces caractéristiques ont un impact direct sur les applications, dans la mesure où elles peuvent subir des défaillances. Il est donc nécessaire de définir un service de tolérance aux fautes pour ce type d'environnements. Ainsi, nous avons défini un modèle multi-arborescent pour une grille mobile, composé de trois niveaux hiérarchiques : (i) les racines qui représentent les nœuds de la grille de calcul et qui vont jouer le rôle de gestionnaire de groupes mobiles ; (ii) au deuxième niveau, nous trouvons les interlocuteurs, servant d'intermédiaires entre les gestionnaires et les autres dispositifs mobiles du plus bas niveau ; (iii) ce niveau est composé des dispositifs mobiles qui sont chargés d'exécuter les jobs. Notre modèle prend en compte les contraintes liées à la mobilité de ces dispositifs, notamment la forte occurrence des fautes. Pour cela, nous avons soutenu notre modèle par des mécanismes de tolérance aux fautes des dispositifs mobiles.

3. **Modèle décentralisé de tolérance aux fautes basé sur les graphes colorés dynamiques :** Dans cet axe, nous avons proposé de modéliser une grille de calcul par un graphe coloré dynamique, où l'ensemble des sommets représente les nœuds de la grille et chaque arc représente les communications entre les entités associées aux nœuds. Nous colorons les sommets du graphe selon un certain ensemble de propriétés, qui reflètent leurs états. Nous proposons deux modèles de tolérance aux fautes basés sur les graphes colorés dynamiques. Dans le premier modèle, nous colorons les sommets en fonction de trois classes de paramètres ; chaque sommet classifie les nœuds de la grille en fonction du niveau de performance, en trois classes (identiques, plus performants et moins performants). La technique de tolérance aux fautes proposée cherche les meilleurs substituts suivant ce niveau de performance. Ensuite, dans le deuxième modèle, les sommets du graphe sont colorés par trois couleurs de base (Rouge, Vert, Bleu) en tenant compte d'un seuil de tolérance préalablement défini. Ainsi, nous obtenons trois catégories de sommets : (i)

des sommets dits *instables* (sommets dont le nombre de voisins susceptibles de les remplacer est inférieur au seuil de tolérance ; (ii) des sommets dits *stables* (sommets dont le nombre de voisins susceptibles de les remplacer est égal au seuil de tolérance) ; et, (iii) des sommets dits *hyperstables* (sommets dont le nombre de voisins susceptibles de les remplacer est supérieur au seuil de tolérance). Nous avons tiré profit des potentialités de notre modèle pour tolérer les fautes de type arrêt et de type déconnexion des nœuds de la grille par une première technique décentralisée basée sur la distribution des jobs sur les voisins, et une deuxième technique basée sur l'élection d'un remplaçant chargé de prendre en charge toute la tolérance des nœuds défaillants.

4. **Modèle fiable de tolérance aux fautes :** Le surcoût engendré par la migration des tâches dans la grille de calcul reste toujours un champ de recherche dans ces environnements. Dans cet axe, nous avons proposé une technique fiable de tolérance aux fautes, qui réduit le surcoût de la migration par une migration périodique des tâches ayant comme période le temps médian de réparation (*Median Time To Repair : MDTTR*) du nœud défaillant. Cette migration est faite vers le substitut le plus fiable parmi un ensemble de substituts.

4 Structure du manuscrit de thèse

L'ensemble des travaux de cette thèse sont synthétisés dans ce manuscrit composé de six chapitres, outre une introduction et une conclusion.
Le chapitre 1 présente un état de l'art sur les grilles de calcul, leurs objectifs, leurs architectures, leurs caractéristiques, ainsi que les différents middlewares et simulateurs associés aux grilles. Cet état de l'art est suivi d'un aperçu général sur les grilles mobiles et les grilles pervasives.
Nous nous intéressons, dans le chapitre 2, au problème de la tolérance aux fautes dans les grilles de calcul. Nous mettons, en particulier, l'accent sur la sûreté de fonctionnement, ses attributs et ses méthodes. Puis, nous mettons en évidence les différentes fautes possibles dans les grilles de calcul, les moyens pour les détecter. Finalement, nous détaillons les différentes techniques de tolérance aux fautes dans les grilles de calcul.
Dans le troisième chapitre, nous présentons les deux modèles hiérarchiques de tolérance aux fautes proposés dans cette thèse. Le premier modèle est une intégration des techniques de tolérance aux fautes dans un modèle de grille, qui est une amélioration du modèle G/S/M, par un modèle hiérarchique dynamique. Pour la grille mobile, un modèle multi-arborescent a été proposé. Ce modèle est soutenu par un mécanisme de tolérance aux fautes, qui prend en compte les spécificités liées au mobile, tel que la disponibilité limitée de ressources, la fréquence de mobilité, la limite de la largeur de la bande passante, ainsi que la déconnexion fréquente de nœuds mobiles.
Dans le chapitre 4, nous présentons les graphes colorés dynamiques comme un nouveau type de graphes, capables d'abstraire toutes les caractéristiques des grilles de calcul. Deux modèles décentralisés de tolérance aux fautes sont proposés sur la base de ces graphes. Le premier colore les sommets d'un graphe en fonction d'un ensemble

d'attributs des nœuds de la grille (attributs liés à leurs potentialités et à leurs états). Chaque nœud classe les autres nœuds en trois classes par rapport à son niveau de performance (identiques, plus performants et moins performants). Ces classes sont ensuite exploitées, selon un processus bien précis, pour déterminer les noeuds qui seront choisis pour remplacer un ou plusieurs noeuds défaillants. Le deuxième modèle colore les sommets en fonction de l'ensemble des voisins collaborateurs (noeuds potentiels pour se substituer à des noeuds défaillants). A partir de ces modèles de graphes, nous avons défini deux mécanismes de tolérance aux fautes basés sur la distribution des jobs sur les collaborateurs ou par la désignation d'un nœud remplaçant d'un nœud défaillant.
Dans le cinquième chapitre, nous donnons un état de l'art sur les différentes techniques de migration des tâches dans les grilles de caclul pour la tolérance aux fautes, suivi par une analyse des systèmes de fiabilité dans les grilles de calcul. Nous avons détaillé notre modèle de fiabilité d'un service de grille, qui permettra plus tard de déterminer le substitut fiable pour chaque service de grille. Par la suite, nous présentons notre approche de tolérance aux fautes basé sur une migration périodique des tâches vers le substitut le plus fiable.
Le chapitre 6 présente et discute les différentes expérimentations que nous avons réalisées pour évaluer les mécanismes de tolérance aux fautes proposés dans les chapitres 3, 4 et 5. Pour cela, nous avons implémenté, sous Globus ToolKit Version 4, une Desktop Grid composée de 15 nœuds. Par contre, le modèle de la grille mobile a été expérimenté avec un simulateur, que nous avons développé à ce sujet en Java. En ce qui concerne les modèles du chapitre 4, ils ont été évalués expérimentalement avec un simulateur développé en Java avec la bibliothèque Graphstream, dédiée à l'exploitation de graphes colorés dynamiques et pour les expérimentations des performances du modèle, nous avons utilisé le simulateur SimGrid. Ce dernier est aussi utilisé pour les expérimentations du modèle fiable de tolérance aux fautes.
Finalement, le manuscrit de thèse se termine par une conclusion générale, qui rappelle à la fois la problématique abordée dans cette thèse, ainsi que nos principales contributions tant sur le plan théorique que sur le plan pratique. Cette présentation est suivie par une liste de quelques perspectives de recherche qui nous semblent présenter un intérêt particulier.

Chapitre 1

Grille de calcul : Etat de l'art

1 Introduction

Le calcul distribué, ou système distribué de manière général, consiste à répartir un calcul ou un système sur plusieurs ordinateurs distincts. Nous pouvons le définir comme une collection de sites autonomes connectés à l'aide d'un réseau de communication [139]. Cette définition implique une propriété importante des systèmes distribués, à savoir une distribution transparente pour l'utilisateur et pour les développeurs d'applications. Les Grilles de Calcul *(Grid Computing)* ont émergé comme une plate-forme prometteuse pour le calcul distribué à grande échelle. Le calcul de grille revêt, depuis plus d'une décennie, un grand intérêt en informatique, car il recèle un potentiel de calcul très large. Son objectif ultime est de transformer le réseau mondial des ordinateurs en une ressource immense et unique de capacité de traitement pratiquement illimitée. Les grilles sont constituées de ressources fortement hétérogènes, géographiquement distribuées et qui appartiennent à différentes organisations indépendantes, ayant chacune leurs propres politiques de gestion et de sécurité. Ces ressources sont soumises à une forte dynamicité du point de vue de leur charge, mais également de leur disponibilité. Des interfaces standards et des mécanismes doivent être fournis afin de masquer, à l'utilisateur, la complexité d'un tel système, et de lui offrir un accès transparent à toutes les ressources de la grille. Pour construire une grille, on utilise un ensemble de logiciels appelé middleware. Le middleware unifie l'accès à des ressources informatiques hétérogènes. Il se place entre les systèmes d'exploitation existants et l'utilisateur. Il masque à ce dernier l'hétérogénéité des machines, des systèmes d'exploitation, des protocoles de communication, des environnements de programmation et d'exécution.

2 Systèmes distribués

Un système distribué est constitué d'un ensemble de sites reliés entre eux par un réseau de communication. L'histoire des systèmes distribués est étroitement liée

à l'évolution des réseaux de communication. Ainsi, l'apparition vers le milieu des années 70 du réseau Ethernet, réseau local à haut débit utilisant un réseau à diffusion, marque une étape importante dans l'émergence des systèmes distribués. Les premiers systèmes distribués, utilisant des réseaux locaux, ont été réalisés en interconnectant plusieurs systèmes homogènes (notamment des systèmes Unix). La plupart de ces systèmes étendent simplement le système de fichiers pour offrir un accès transparent aux fichiers locaux ou distants ; d'autres permettent la création et l'exécution de processus à distance [139]. Tanenbaum définit un système distribué comme une *"collection d'ordinateurs indépendants qui paraissent aux utilisateurs du système comme un système unique et cohérent"* [192]. Il y a deux points essentiels dans cette définition. Le premier est l'usage du mot indépendant ; cela veut dire que, du point de vue architecture, les machines sont capables d'opérer indépendamment. Le deuxième point est que cette séparation est cachée à l'utilisateur laissant apparaître une seule machine "virtuelle". Plusieurs facteurs ont contribué au développement des systèmes distribués : (i) les ordinateurs sont devenus plus petits et meilleur marché ; (ii) les technologies de communication ont progressé au point où il est très facile et peu coûteux de connecter un ensemble d'ordinateurs ; (iii) les débits et la sécurité dans les réseaux ont connu des progrès notoires ; (iv) la croissance explosive de l'Internet et du World Wide Web dans le milieu des années quatre-vingt dix a permis d'avoir des systèmes distribués au-delà de leurs domaines d'application traditionnels, tels que l'automatisation industrielle, la défense et les télécommunications, et presque dans tous les domaines, y compris le commerce électronique, les services financiers, la santé, la gouvernance et le divertissement.

2.1 Avantages de la distribution

Les propriétés suivantes rendent les systèmes distribués de plus en plus présents dans l'informatique actuelle et du future [192] :

- **Collaboration et connectivité :** Une motivation importante pour les systèmes distribués est leur capacité à rassembler de larges quantités d'information et des services distribués, tels que les sites de commerce électronique, les encyclopédies. La popularité de la messagerie instantanée et les forums de discussion sur Internet met en évidence une autre motivation pour les systèmes distribués : garder le contact entre personnes, collaborer, etc.
- **Economie :** Les réseaux informatiques qui intègrent les PDA, les ordinateurs portables, les PC et les serveurs offrent souvent un meilleur rapport performance/prix par rapport aux gros ordinateurs centralisés. Par exemple, ils supportent la décentralisation et peuvent partager des périphériques coûteux, comme les serveurs de fichiers de grande capacité et des imprimantes à haute résolution. De même, les composants logiciels et les services peuvent être exécutés sur des sites avec des attributs de qualités de performance réservés jusque là à certaines catégories d'ordinateurs ou applications.
- **Performance et scalabilité :** Les services applicatifs suscitent, avec le temps, plus d'utilisateurs d'où la nécessité d'augmenter les performances de systèmes distribués pour supporter cette charge de travail. L'augmentation significative des performances peut être obtenue en utilisant la puissance de calcul combinée de nœuds (ou sites) de calcul en réseau.

- **Tolérance aux fautes :** Une des conséquences immédiate de l'informatique distribuée est de tolérer les défaillances d'un système. Les éléments d'un système distribué (nœuds, réseaux, services, etc.) sont susceptibles de tomber en panne suite à des défaillances. Ces défaillances doivent être gérées avec transparence et sans affecter le fonctionnement global du système distribué. D'une manière générale, la mise en œuvre de la tolérance aux fautes nécessite l'utilisation de la réplication à travers les nœuds et les réseaux du système. Elle permet de minimiser les points de défaillance unique, ce qui peut améliorer la fiabilité du système face à des défaillances partielles.

2.2 Défis de la distribution

En dépit de l'omniprésence croissante et l'importance des systèmes distribués, les développeurs de logiciels pour les systèmes distribués font face à plusieurs défis [176] :

- **Complexités inhérentes** qui surviennent de défis inhérents au domaine : Par exemple, les composants d'un système distribué résident souvent dans des espaces d'adressage séparés et donc la communication intra-nœuds a besoin de nouveaux mécanismes, politiques et protocoles. De plus, la synchronisation et la coordination est plus compliquée dans un système distribué puisque les composants peuvent être exécutés en parallèle et la communication peut être asynchrone et non-déterministe. Les réseaux qui connectent des composants dans les systèmes distribués introduisent des effets supplémentaires, tel que la latence, l'instabilité, les fautes transitoires, la surcharge, etc. [210].
- **Complexités accidentelles** qui surviennent de limitations des outils logiciels et des techniques de développement telles que les API's non-portables et les débogueurs distribués non compatibles.
- **Méthodes et techniques inadéquates**, les méthodes d'analyse des logiciels et les techniques de conception populaires [45, 63] sont conçues pour un seul-processus. Le développement de systèmes distribués de qualité avec le respect des exigences de performance, tel que vidéo-conférence ou le contrôle du trafic aérien a été laissé aux experts qualifiés d'architectures des logiciels.

3 Modèle client/serveur

Le modèle client/serveur désigne un mode de comportement et de communication entre plusieurs applications qui s'exécutent sur des ordinateurs connectés en réseau. Ce modèle définit un comportement synchrone entre un client et un serveur, dans la mesure où une fois que le client a soumis une requête au serveur, il doit attendre une réponse de ce dernier pour reprendre son exécution.

4 Système Pair à pair

Le premier système Pair à Pair ou Peer-To-Peer(P2P) est apparu en 1999 [211]. Il s'agissait du logiciel Napster [174], logiciel permettant la diffusion de fichiers mu-

sicaux (principalement au format mp3). Ce système possédait une structure centralisée, à savoir que les pairs (ou nœuds) du réseau envoyaient à un serveur central la liste des fichiers qu'ils mettaient à disposition des utilisateurs. Ce système est apparu aux yeux du grand public comme une véritable révolution, puisqu'il s'agissait du premier logiciel massivement utilisé qui se substituait au modèle client/serveur qui caractérisait les applications utilisées sur Internet jusqu'à présent. L'apparition de Napster a donc modifié la hiérarchie existante entre les différents ordinateurs connectés à Internet, en permettant à chacun de devenir un serveur [179].

4.1 Caractéristiques

Les différentes caractéristiques intrinsèques au modèle P2P garantissent aux systèmes un fonctionnement à large échelle [129]. Un très grand nombre de pairs peuvent interagir dans le réseau, de manière à permettre le partage d'une grande quantité de ressources. Ainsi, le comportement global du réseau résulte uniquement des interactions locales entre les pairs. Les pairs sont autonomes et volatiles, ce qui nécessite de gérer leurs connections et déconnexions dynamiquement. La caractéristique principale de ce type de systèmes [129] est l'absence d'une infrastructure fixe, ce qui les rend difficile à contrôler, évaluer dans leur ensemble [5], mais facilite leur passage à l'échelle. La dynamicité des utilisateurs, des nœuds, des routes et des flux échangés [168], ainsi que l'auto-organisation d'un système sont les mots clés utilisés principalement pour le partage de ressources [125].

4.2 Avantages et inconvénients du modèle P2P

Les avantages des systèmes P2P sont nombreux [183]. Ils donnent accès à un grand nombre de ressources et disposent d'une administration transparente. Un système P2P évolue sans machine(s) dédiée(s) pour son administration. Ces systèmes sont conçus de telle sorte qu'aucun des pairs ne soit indispensable au fonctionnement général : le système n'est pas paralysé par la défaillance d'un ou plusieurs pairs [143]. Les qualités de ce système (robustesse, disponibilité, performances, etc.) augmentent avec le nombre d'utilisateurs, et présentent de nombreux avantages (décentralisation, pas de coûts d'infrastructure, etc.). En contrepartie, de par la volatilité des pairs, le système est non fiable. De plus, des problèmes de sécurité peuvent survenir à cause de certains pairs malveillants. Ce système ne peut pas assurer la confidentialité des données échangées et les vitesses de transfert sont aléatoires.

5 Grappes de calcul (Cluster)

De nos jours, les grappes d'ordinateurs personnels (PC) fournissent une alternative aux multiprocesseurs fortement couplés [144]. Le concept de grappe est né en 1995 avec le projet Beowulf sponsorisé par la NASA [170]. Un des buts initiaux du projet était l'utilisation des ordinateurs massivement parallèles pour traiter les grands volumes de données utilisées dans les sciences de la terre et de l'espace. La

première grappe a été construite pour adresser des problèmes liés aux gigantesques volumes de données des calculs sismiques. Ces nouvelles solutions permettent aux utilisateurs de bénéficier de plates-formes fiables et très performantes à des prix relativement faibles : le rapport prix/performance d'une grappe de PC est de 3 à 10 fois inférieur à celui des supercalculateurs traditionnels. Les grappes sont composées d'un ensemble de serveurs (PC) interconnectés entre eux par un réseau. Ils permettent de répondre aux problématiques de haute performance et de haute disponibilité. Elles ont été utilisées avec succès [189] pour, par exemple, les moteurs de recherche sur Internet utilisant des fermes de serveurs à grands volumes (comme Google par exemple). Les applications dans lesquelles sont utilisées les grappes sont typiquement des applications de lectures intensives, ce qui rend plus facile l'exploitation du parallélisme.

5.1 Définition et caractéristiques d'un cluster

On parle de cluster, de grappe de serveurs ou de ferme de calcul pour désigner des techniques consistant à regrouper plusieurs ordinateurs indépendants (appelés nœuds), afin de permettre une gestion globale et de dépasser les limitations d'un ordinateur. Les grappes de serveurs représentent une technologie résidant dans la mise en place de plusieurs ordinateurs en réseau qui vont apparaître comme un seul ordinateur ayant plus de capacités. Cet usage optimisé des ressources permet la répartition des traitements sur différents nœuds. Un des grands avantages d'un cluster est qu'il n'est plus besoin d'acheter une machine multiprocesseurs coûteuse, mais qu'il est désormais possible de se contenter de petits systèmes que l'on peut connecter les uns aux autres selon l'évolution des besoins.

5.2 Catégories de cluster

Il existe différentes catégories de clusters à savoir [222] :

5.2.1 Clusters de haute disponibilité

Les clusters de haute disponibilité sont utilisés pour protéger une ou plusieurs applications sensibles [222]. Pour atteindre cet objectif, l'application et toutes les ressources qui lui sont nécessaires seront contrôlées de manière permanente. Pour avoir une haute protection puissante d'une application, il faut inclure dans cette protection le matériel, le réseau et le système d'exploitation. Généralement, plusieurs produits sont utilisés afin de protéger plusieurs applications sur un même nœud, mais il existe des solutions capables de protéger autant d'applications que l'on veut. Les clusters de haute disponibilité typiques contiennent seulement quelques nœuds mais on peut utiliser des clusters regroupant 32 ou 64 nœuds.

5.2.2 Clusters de haute performance

La fonction principale d'un cluster à haute performance est d'augmenter la puissance d'un PC. Pour cela, il est nécessaire de découper une tâche en sous-tâches. L'unité de gestion - pour coordonner toutes les sous-tâches et le nœud destinataire du résultat sont les seuls points critiques (*single point of failure SPOF*). Ces composants pourront être protégés via un cluster de haute disponibilité. Le crash de l'un des nœuds n'est pas un désastre car le travail de ce nœud pourra être fait par un autre. La performance du cluster s'en verra affaiblit mais le cluster fonctionnera toujours.

5.3 Exemples de clusters

5.3.1 Beowulf

Beowulf est une architecture multi-ordinateurs qui peut être utilisée pour la programmation parallèle [170]. Cette architecture comporte habituellement un nœud serveur et un ou plusieurs nœuds clients connectés entre eux à travers Ethernet ou tout autre réseau. C'est un système construit en utilisant des composants matériels existants, comme tout PC capable de fonctionner sous Linux, des adaptateurs Ethernet standards et des switches. Il ne contient aucun composant matériel propre et est aisément reproductible. Beowulf utilise aussi des éléments comme le système d'exploitation Linux, les bibliothèques Parallel Virtual Machine (PVM) [13] et Message Passing Interface (MPI) [79]. Le nœud serveur contrôle l'ensemble du cluster et joue le rôle de serveur de fichiers pour les nœuds clients. Il est aussi la console du cluster et la passerelle *(gateway)* vers le monde extérieur. De grandes machines Beowulf peuvent avoir plus d'un nœud serveur, et éventuellement aussi d'autres nœuds dédiés à des tâches particulières, comme par exemple des consoles ou des stations de surveillance.

5.3.2 MOSIX

MOSIX est un cluster à répartition de charges entre processus [12]. En effet, chaque application pourra migrer entre les nœuds afin de tirer avantage de la meilleure ressource disponible. Son idée principale consiste à répartir sur plusieurs machines, non pas le calcul, mais un ensemble de multitâches. En fait, MOSIX est divisé en deux modules internes : une partie gérant le mécanisme préemptif de migration de processus (PPM), et un autre gérant un ensemble d'algorithmes permettant la migration et le partage de ressources du cluster. Ces deux fonctions permettent ainsi un monitoring régulier des processus, afin de les déplacer pour optimiser les performances du système.

6 Grille de calcul

L'augmentation constante des besoins en terme de puissance de calcul informatique a toujours été un défi auquel la communauté informatique est confrontée [61]. Même si les évolutions technologiques de ces dernières années ont permis d'aboutir à la création de machines de plus en plus puissantes, celles-ci ne fournissent toujours pas suffisamment de puissance pour effectuer des calculs d'une complexité très élevée, ou traitant un grand volume de données. Le calcul parallèle ou distribué fournit une solution à ce problème à condition de disposer d'infrastructures matérielles appropriées. Une des solutions consisterait à répartir un calcul complexe sur un ensemble de machines, reliées entre elles par des réseaux rapides. Parmi ces infrastructures, nous pouvons citer les grilles de calcul [15].

6.1 Définition

Le concept de grille de calcul étant encore relativement récent, nous pouvons en trouver plusieurs définitions de ce concept que ce soit dans la littérature scientifique ou sur Internet. Nous reprendrons, dans ce chapitre, les deux définitions données par I. Foster et C. Kesselman [60], et discuterons brièvement les différentes notions abordées dans celles-ci.
La première définition donnée par I. Foster et C. Kesselman date de 1998, dans le livre *"The Grid : Blueprint for a New Computing Infrastructure"* [60] : *" Une grille de calcul est une **infrastructure** matérielle et logicielle fournissant un accès **fiable** (dependable), **cohérent** (consistent), **à taux de pénétration élevé** (pervasive) et **bon marché** (inexpensive) à des capacités de traitement et de calcul ".*
C'est une **infrastructure** car une grille devra fournir des ressources (calcul, stockage, etc.) à grande échelle. Cela nécessite une quantité significative de matériels qui constituera les ressources de la grille et une quantité importante de logiciels pour bien utiliser, contrôler et superviser cet ensemble de ressources.
La nécessité d'un service **fiable** est fondamentale. Les utilisateurs d'une telle infrastructure s'attendent à recevoir un service prédictible, continu et performant.
La cohérence suggère, tout comme dans une grille d'électricité, la présence de services standards, accessibles via des interfaces standards et opérantes selon des paramètres standards.
Un **taux de pénétration élevé** permet de garantir que les services seront facilement accessibles par une large population.
Finalement, l'aspect **bon marché** est très important d'un point de vue viabilité économique.
En 2000, dans l'article *"The Anatomy of the Grid"* [62], de S. Tuecke, cette définition sera modifiée en y ajoutant des éléments sociaux et des politiques d'accès. Ainsi, les grilles de calcul sont concernées par *"le partage de ressources et la résolution coordonnée de problèmes dans des **organisations virtuelles** dynamiques et multi-institutionelles".*
La notion d'Organisation Virtuelle (OV) *(Virtual Organization)* est définie comme un ensemble d'individus et/ou d'institutions qui partagent des ressources et des services et qui sont soumis à des politiques de sécurité spécifiant les autorisations de ce partage. Ces organisations virtuelles peuvent varier fortement en taille, en struc-

ture, en but ou en durée. Elles nécessitent des mécanismes de partage très flexibles permettant de gérer notamment l'accès aux ressources, qui doit pouvoir être reconfiguré avec précision. Une ressource appartenant à une certaine OV est disponible pour certaines personnes, sous certaines conditions et à certains instants.

6.2 Caractéritiques d'une grille

Une grille de calcul consiste à exploiter pleinement les ressources de l'intégralité d'un parc informatique (serveurs et PC). C'est une forme d'informatique distribuée basée sur le partage dynamique des ressources entre des participants, des organisations et des entreprises dans le but de pouvoir les mutualiser, et faire ainsi exécuter des applications de calcul intensif ou des traitements utilisant de très gros volumes de données. Les grilles de calcul possèdent quatre principales caractéristiques [9] :

1. **Existence de plusieurs domaines administratifs :** les ressources sont géographiquement distribuées et appartiennent à différentes organisations, chacune ayant ses propres politiques de gestion et de sécurité. Ainsi, il est indispensable de respecter les politiques de chacune de ces organisations.
2. **Hétérogénéité des ressources :** les ressources dans une grille sont de nature hétérogène en terme de matériels, de logiciels, d'accès, etc.
3. **Passage à l'échelle *(scalability)* :** une grille pourra être constituée de quelques dizaines de ressources à des millions voire des dizaines de millions de ressources. Cela pose de nouvelles contraintes sur les applications et les algorithmes de gestion des ressources.
4. **Dynamicité des ressources :** les grilles sont caractérisées par leur aspect dynamique (arrivée de nouveaux membres, départ des membres existants, etc.). Cela pose des contraintes sur les applications telles que l'adaptation au changement dynamique du nombre de ressources, la tolérance aux fautes et aux délais d'allocation, etc.

6.3 Architecture d'une grille de calcul

L'architecture d'une grille de calcul est organisée en couches (voir Figure 1.1) [62]. Une couche est une abstraction représentant un ensemble de fonctions de la grille. Chaque couche fait appel aux services de toutes les couches inférieures. La couche Fabrique ou infrastructure matérielle fournit les ressources. Ce sont, d'un point de vue physique, des ressources telles que des processeurs pour le calcul, des unités de stockage ou des ressources réseau. La couche Fabrique est en relation directe avec le matériel pour mettre à la disposition des utilisateurs les ressources partagées. Lorsqu'une demande d'accès à une ressource est formulée, par le biais d'une opération de partage d'un niveau supérieur, des composants logiciels du niveau Fabrique sont invoqués. Le rôle de ces composants est d'agir directement sur les ressources de la grille.
La couche Connectivité implémente les principaux protocoles de communication et d'authentifications nécessaires aux transactions sur un réseau de type grille. Les protocoles de communication permettent l'échange des données à travers les ressources

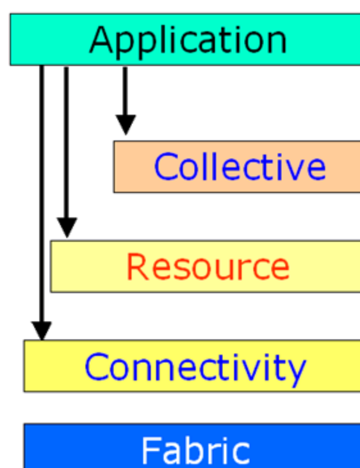

FIGURE 1.1 – Architecture d'une grille de calcul [62]

du niveau Fabrique. Ces protocoles d'authentification s'appuient sur les services de communication pour fournir des mécanismes sécurisés de vérification de l'identité des utilisateurs et des ressources.
La couche Ressource ou intergiciel utilise les services des couches Connectivité et Fabrique pour collecter des informations sur les caractéristiques des ressources, les surveiller et les gérer. Elle s'occupe également de l'aspect facturation et fournit les intergiciels nécessaires à la gestion des ressources, la coordination de l'accès, l'ordonnancement des tâches, etc.
La couche Collective ou environnement et outils de programmation regroupe tous les outils et les paradigmes pouvant aider les développeurs à concevoir et à développer des applications pouvant tourner sur une grille. On y trouve plus particulièrement des compilateurs, des librairies, des outils de développement d'applications parallèles ainsi que des interfaces de programmation ou API (découverte et réservation des ressources, des mécanismes de sécurité, stockage, etc.) que les développeurs d'applications pourront utiliser.
Enfin, la couche la plus haute du modèle est la couche Application qui correspond aux applications qui sont de nature variée : projets scientifiques, médicaux, financiers, ingénierie, etc.

6.4 Topologies d'une grille de calcul

Il existe trois types de topologies de grille [55] : Intragrilles (*Intragrids*), Extragrilles (*Extragrids*) et Intergrilles (*Intergrids*) :

- **Intragrille :** C'est la grille la plus simple, car elle est composée d'un ensemble relativement limité de ressources et de services et appartenant à une organisation unique. Les principales caractéristiques d'une telle grille sont l'interconnexion à travers un réseau performant et haut débit, un domaine de sécurité unique et maîtrisé par les administrateurs de l'organisation et un ensemble relativement statique et homogène de ressources.
- **Extragrille :** C'est une agrégation de plusieurs intragrilles. Les principales

caractéristiques d'une telle grille sont la présence d'un réseau d'interconnexion hétérogène haut et bas débit (LAN/WAN), de plusieurs domaines de sécurité distincts, et d'un ensemble plus ou moins dynamique de ressources.
– **Intergrille :** Elle consiste à agréger les grilles de multiples organisations en une seule grille. Les principales caractéristiques d'une telle grille sont la présence d'un réseau d'interconnexion très hétérogène haut et bas débit (LAN / WAN), de plusieurs domaines de sécurité distincts et ayant parfois des politiques de sécurité différentes, et d'un ensemble très dynamique de ressources.

6.5 Cadres d'utilisation de grille

Une grille de calcul offre un éventail de possibilités pour tous les domaines pouvant bénéficier de ses capacités de traitement et de stockage. Nous présenterons, brièvement, dans ce qui suit cinq grandes classes d'applications pour lesquelles une grille pourrait apporter une nouvelle vision de traitement de ses problèmes [15].

1. **Calcul distribué :** Les applications de calcul distribué sont évidemment d'excellentes candidates pour être utilisées sur une grille. Elles bénéficient ainsi d'un nombre beaucoup plus important de ressources de calcul leur permettant de résoudre des problèmes qui leur étaient auparavant inaccessibles. Parmi les principaux défis que doit relever l'architecture d'une grille pour de telles applications [15] :
   – L'ordonnancement à grande échelle des processus.
   – La souplesse des algorithmes et des protocoles qui doivent être capables de gérer un nombre de nœuds pouvant aller de la dizaine à des centaines voire des milliers de machines.
   – La tolérance des algorithmes aux temps de latence inhérents à la taille de la grille.
   – La possibilité d'atteindre et de maintenir un haut niveau de performances dans un système très hétérogène.
2. **Calcul haut débit :** Une grille de calcul sera utilisée pour ordonnancer en parallèle une importante quantité de tâches. Comme domaines d'application, nous pouvons citer la recherche de clés cryptographiques, les simulations de molécules, l'analyse du génome, etc.
3. **Calcul à la demande :** Ce type d'applications utilise une grille afin de satisfaire des besoins à court terme en ressources, tels qu'ils ne peuvent être satisfaits en local pour des raisons pratiques ou de rentabilité. Ces ressources peuvent être du temps de calcul, des logiciels, des données et des capteurs spécialisés.
4. **Traitement massif de données :** Dans ce type d'applications, le but est d'extraire de nouvelles informations à partir de grandes bases de données géographiquement distribuées. Généralement, ces types de traitement sont également de grands consommateurs de puissance de calcul et de bande passante. Les systèmes de prévisions météorologiques modernes utilisent énormément de données récoltées aux quatre coins du globe (comme des observations satellites par exemple). Le processus complet implique des transferts et des traitements de plusieurs dizaines de Téraoctets de données.

5. **Informatique collaborative :** Le but des applications collaboratives est de permettre et de favoriser les interactions entre les personnes. Elles sont souvent structurées sous forme d'espaces virtuels partagés entre les utilisateurs. La plupart de ces applications permettent de partager des ressources comme des données ou des résultats de simulations [15].

6.6 Grille de données

Les premières applications sur les grilles étaient pour la plupart basées sur une organisation des données en fichiers. Cependant, destinées à un spectre beaucoup plus large d'applications, aussi bien scientifiques que commerciales, il est nécessaire de disposer d'applications qui offrent les fonctionnalités d'une organisation en base de données (requêtes, transactions, etc.). Etant donné la diversité des gestionnaires de bases de données aussi bien dans leur paradigme (objet, relationnel) que dans les fonctionnalités qu'ils peuvent offrir, leur adaptation aux grilles s'avère être une problématique nécessitant la collaboration de plusieurs disciplines. Il est donc primordial de disposer d'un "middleware" générique pour la fédération des données stockées dans les grilles. Par exemple, les fonctionnalités nécessaires à un système de gestion de bases de données sur les grilles doivent recouvrir celles offertes par les SGBD courants [215]. Pour réaliser cet objectif, on peut tenter d'adapter les systèmes de bases de données actuels aux grilles [201], approche ne permettant pas de tirer plein parti de la puissance de stockage et de calcul de cette architecture, ou reconstruire un gestionnaire SQL dédié spécialement pour les grilles. Les briques de base concernant la gestion des données dans une telle architecture sont les suivantes [150] :

- **Stockage :** Les données disponibles sont stockées sur une infrastructure stable (ou leur pérennité est assurée par des sauvegardes régulières). Le service de stockage permet à un utilisateur d'enregistrer des données sur le système. Il est à noter que toutes les données n'ont pas vocation à être stockées : par exemple, celles issues de capteurs ne sont pas forcément toutes stockées. Faute de place, d'autres resteront dans des zones de stockage en dehors de la grille, pour des raisons de sécurité.
- **Réplication :** Deux motivations principales plaident pour un service de réplication. Tout d'abord, l'efficacité du système est renforcée quand plusieurs copies d'une donnée existent. En ajustant l'emplacement des copies en fonction des usages d'une donnée, l'une d'entre elles a des chances d'être assez proche de son utilisateur. La seconde motivation vient pour assurer une continuité d'accès aux données. En effet, certains dispositifs étant instables, réaliser une copie d'une donnée la rend plus disponible. Par contre, faute de place, toutes les données ne peuvent pas être répliquées et un mécanisme de choix judicieux des répliques et de leur emplacement doit exister.
- **Cache :** La présence d'un système de cache permet d'améliorer l'efficacité d'un système en gardant temporairement certaines données pour éviter de réexécuter des requêtes qui peuvent s'avérer coûteuses. La différence avec le service de réplication réside beaucoup plus dans son caractère système, caché, pour l'utilisateur final. Alors que ce dernier peut contrôler le nombre de répliques d'une donnée, les données cachées lui sont inconnues. Le service de cache sert

d'intermédiaire entre les requêtes des utilisateurs et les données elles-mêmes. Pour être efficace, les divers caches présents doivent collaborer, afin d'optimiser l'espace de stockage disponible pour le cache, et améliorer le pourcentage de requêtes aux données dans le cache par rapport aux requêtes hors cache, lors des accès aux données.
– **Accès :** Le service d'accès a deux rôles principaux. Tout d'abord, il récupère les données stockées dans le système (par le système de stockage), puis il s'interface aussi avec les données externes. Il réalise également la médiation entre les sources de données et les applications/utilisateurs qui en ont besoin.

6.7 Normes et grilles

Devant la prolifération des grilles et des outils associés, une standardisation internationale a été mise en place. Cette standardisation a donné lieu à deux normes : la norme OGSA (*Open Grid Services Architecture*) [59] et la norme OGSI (*Open Grid Service Infrastructure*) [61].

1. - Norme OGSA initiée en 1998 [59], cette norme tente de standardiser une architecture pour les grilles de calcul. OGSA est un projet en constante évolution et amélioration. Elle se base sur les concepts des Web Services [10], afin de définir son architecture et son implémentation. De ce fait, elle est une architecture orientée services. Notons qu'OGSA est une architecture et non pas une implémentation ou des outils particuliers de construction de grilles de calcul.
2. - OGSI [61] est une spécification technique utilisée pour implémenter les services d'une grille suivant la norme OGSA. La spécification de l'infrastructure OGSI a pour objectif de normaliser les services composant les grilles, afin de garantir l'interopérabilité de systèmes hétérogènes pour le partage et l'accès à des ressources de calcul et de stockage distribuées. Cette spécification définit ce qu'est un service de grille, c'est à dire un Web Service respectant un ensemble de conventions (interfaces et comportement) adaptées aux contraintes de la norme OGSA. La spécification OGSI s'appuie elle-même sur de nombreux standards, au niveau des protocoles afin de favoriser l'interopérabilité, et également au niveau des API utilisées pour faciliter la portabilité entre les environnements d'exécution.

6.8 Outils d'expérimentations des grilles

Les expérimentations sur les grille sont difficiles à manipuler, car cela nécessite des outils efficaces pour le contrôle et l'observation des paramètres d'une grille [3]. Il existe deux méthodes pour travailler sur les grilles : soit l'utilisation de modèles théoriques, soit des expérimentations sur des grilles physiques. Les modèles théoriques (simulateurs), bien qu'ils soient intéressants, ne permettent pas de capturer des conditions réelles d'utilisation d'une grille (codes exécutables, etc.) et de reproduire les conditions expérimentales. A l'inverse, les expérimentations réelles ne permettent pas le contrôle des paramètres, les observations fines et la reproductibi-

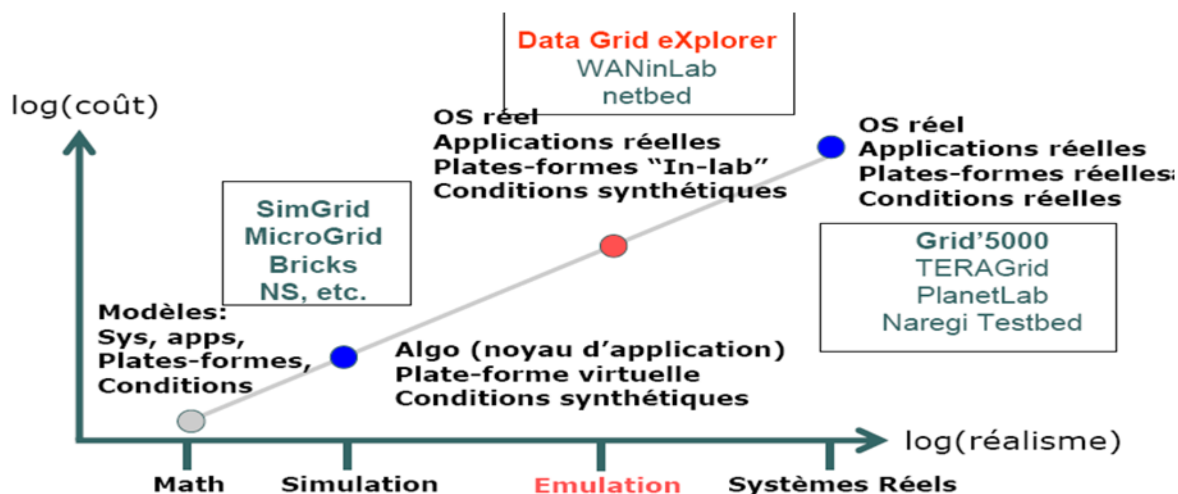

FIGURE 1.2 – Outils d'expérimentations sur les grilles [30].

lité. La Figure 1.2 montre quelques outils d'expérimentations sur les grilles [30].

6.8.1 Middlewares

Les grilles de calcul nécessitent une couche logicielle responsable de la gestion, de la coordination et de l'accès aux différentes ressources [62]. Elles utilisent le concept de middleware ou d'intergiciel pour désigner cette couche. Un middleware est la brique de base regroupant l'ensemble des éléments logiciels pour la mise en œuvre d'une grille. Il comprend notamment les fonctions suivantes [98] :

- Le partage et l'allocation des différentes ressources de la grille suivant des critères techniques de performance, mais également des critères économiques et d'éventuelles contraintes utilisateurs.
- L'exécution, l'ordonnancement et l'administration de la grille, intégrant toutes les fonctions de monitoring et de gestion.
- L'ensemble des procédures de sécurisation de la grille, notamment les outils d'authentification des utilisateurs, la gestion des restrictions d'accès, la confidentialité des données et des résultats.
- Les outils collaboratifs permettant aux divers acteurs de travailler ensemble et d'échanger des documents, des données, des logiciels, des résultats, etc., en garantissant leur cohérence au cours de l'ensemble des manipulations.
- Les outils d'évaluation des performances et de mesure de la qualité de services.
- Les outils de développement et les interfaces utilisateurs pour le déploiement des applications.

Ces middlewares s'appuient sur des protocoles standards de l'Internet tels que FTP (*File Transfer Protocol*) [153], LDAP (*Light Directory Access Protocol*) [184], HTTP (*HyperText Transfert Protocol*) [16]. Parmi les middlewares les plus utilisés actuellement, il faut citer LEGION [78], UNICORE [3] et GLOBUS [59].

6.8.2 Simulateurs de grille

A défaut de disposer de grilles réelles, il existe différents outils de simulation des grilles dont les plus importants sont :
SimGrid : SimGrid [31] est un logiciel développé par Henri Casanova et le groupe de recherche AppLe de l'université de Californie à San Diego. C'est un simulateur modulaire écrit en C. Il a été spécifiquement conçu pour permettre l'étude du comportement d'applications distribuées sur des plates-formes réalistes (de la grille au réseau de stations de travail).
GridSim : GridSim [28] est un simulateur de grille, développé par Rajkumar Buyya et Manzur Murshed. Il a été utilisé pour simuler les algorithmes d'ordonnancement simples ou multiples dans les systèmes distribués tels que des clusters de grilles. Il fournit un service complet pour la simulation de différentes classes de ressources hétérogènes, utilisateurs, applications et ordonnanceurs.
EDGSim : EDGSim [41] est une simulation du flux de jobs, de données physiques et de l'information autour d'une grille informatique.
OptorSim : OptorSim [14] est un simulateur de grille de données (*Data Grid*) écrit en Java. Il a été conçu pour étudier l'efficacité des algorithmes d'optimisation de répliques dans une grille de données.
GangSim : GangSim [48] est une amélioration de l'outil du *Ganglia Monitoring Toolkit* [130] pour les organisations virtuelles. Il simule un environnement qui comprend effectivement un très grand nombre de ressources, où des centaines d'établissements et des milliers d'individus contrôlent collectivement des dizaines ou des centaines de milliers d'ordinateurs et systèmes associés de stockage.

7 Grille mobile

Une solution possible au déficit de ressources dans les dispositifs mobiles consisterait à utiliser des services de grille pour permettre à des utilisateurs d'accéder aux ressources informatiques distribuées automatiquement [127]. Les divers services de grille peuvent augmenter les possibilités des dispositifs mobiles de sorte que des tâches complexes puissent être accomplies par des dispositifs mobiles à faibles capacités de traitement et de stockage. Par exemple, le téléphone intelligent du voyageur avec un appareil-photo intégré peut produire un grand volume de données qui devront être traitées. Il peut alors, grâce à son téléphone, rechercher des services de grille pour faire des opérations complexes sur de gros volumes de données qu'il aura réussi à capter par son appareil photo intégré à son téléphone.
Une des choses les plus critiques pour mettre en place une grille mobile serait d'avoir, d'abord, une définition cohérente et précise d'une grille mobile. Une grille peut être vue comme un système informatique à haute performance et une infrastructure de manipulation de données. Ce type de grilles doit avoir la capacité de déployer des réseaux ad-hoc et de fournir un réseau auto-configurable des ressources mobiles (des dispositifs matériels mobiles et des utilisateurs) reliées par des liaisons sans fil, formant des topologies arbitraires et imprévisibles.

8 Grille pervasive

Le développement des équipements mobiles et embarqués, ainsi que des réseaux sans-fil de proximité, a amené la notion d'environnement pervasif, permettant d'accéder à l'information à tout moment où que l'on soit. La complexité des requêtes et l'augmentation des quantités d'information à traiter ont fait émerger le besoin d'une grande puissance de calcul au sein de l'environnement pervasif. Les concepts de Grille de Calcul et de Système Pervasif commencent alors à converger vers une structure unifiée : la Grille Pervasive.

8.1 Définition

Une grille pervasive [150] est une architecture orientée services, permettant un partage de ressources à travers une infrastructure souple, prenant en compte les contraintes issues des deux mondes : la grille de calcul et l'environnement pervasif. Ce concept, adopté par le Global Grid Forum en 2005 [58], est encore au stade de la définition, telle que celle proposée par Parshar et Pierson [146].

8.2 Architecture générale d'une grille pervasive

L'architecture d'une grille pervasive [150] se différencie d'une architecture de grille de calcul par la prise en compte dans chaque développement de service, de l'instabilité, de la mobilité, de la contextualisation propre aux systèmes pervasifs. De manière complémentaire, elle se distingue d'une architecture pervasive classique en permettant l'utilisation d'une infrastructure stable pour mener à bien une partie des travaux coûteux, ou pour utiliser des ressources de stockage stables. Par rapport à une architecture de type P2P, chaque nœud ne va pas avoir la même finalité et le même rôle. Il s'agit en fait d'un mélange entre une architecture client/serveur et une architecture P2P, dont les rôles sont dynamiquement distribués. Nous pouvons dire que l'existence d'une partie stable de l'infrastructure tend vers une architecture client/serveur, mais que la partie instable tend vers le P2P.

9 Conclusion

Une grille de calcul se caractérise par un ensemble (ou plutôt une mutualisation) de ressources hétérogènes et distribuées à une très large échelle. Il apparaît clairement dans un tel contexte que l'utilisation de standards est un point particulièrement critique pour la réussite de la mise en place d'une infrastructure de grille, notamment pour faire face aux problèmes d'hétérogénéité du matériel et des logiciels. Il ressort également de cette définition d'une grille, le besoin d'avoir des mécanismes permettant de gérer plus facilement la volatilité de l'état des différents composants de la grille, qu'il s'agisse des ressources matérielles ou logicielles, de la constitution de groupes d'utilisateurs et/ou de ressources et de leurs connectivités, etc. L'évolution de la technologie grille est freinée par plusieurs problèmes d'ordre technique ou organisationnel. Parmi ces problèmes, le taux élevé de fautes est l'un

des plus cruciaux, car les grilles sont composées de plusieurs clusters reliés par un réseau à haut débit et servent principalement aux applications de calcul intensif. Ce type de calcul nécessitent des périodes très longues dans le temps (des jours ou des mois), et pendant ces périodes, il y a une forte probabilité qu'une panne survienne quelque part dans la grille. Les pannes peuvent être matérielles et/ou logicielles, et elles peuvent également toucher la partie communication (réseau). L'occurrence de telles pannes peut avoir des conséquences plus ou moins graves selon le ou les éléments qui sont tombés en panne, l'instant où ils sont tombés en panne, leurs conséquences sur les autres éléments de la grille et sur les utilisateurs, leur coût de réparation, etc. Ce sont autant de paramètres qui montrent toute la complexité de la prise en charge des pannes dans les grilles. Le chapitre deux de cette thèse sera consacré au problème de fautes dans les grilles de calcul et les différentes techniques de tolérance aux fautes adoptées.

Chapitre 2

Tolérance aux fautes dans les grilles de calcul

1 Introduction

La tolérance aux fautes dans les systèmes distribués est un thème de recherche récurrent à cause de sa dépendance avec les différents services de ces systèmes, les nouvelles technologies et les exigences des utilisateurs en matière de disponibilité, de sécurité et de fiabilité [8]. Avec l'émergence de la technologie des grilles de calcul, la tolérance aux fautes est devenue une de ses importantes propriétés car la fiabilité de ses ressources ne peut pas être totalement garantie. Les spécialistes cherchent à développer des modèles de tolérance aux fautes adaptés aux grilles de calcul et des techniques de tolérance aux fautes pour les applications de grille.

2 Sûreté de fonctionnement

La sûreté de fonctionnement *(dependability)* des systèmes informatiques est définie comme étant la propriété permettant aux utilisateurs d'un système de placer une confiance justifiée dans le service qu'il leur délivre [?]. Mettre en œuvre la sûreté de fonctionnement d'un système correspond à lutter contre les défaillances du système. Cette sûreté est basée sur les notions suivantes [8] :

Faute *(Fault)* : c'est toute cause (événement, action, circonstance) pouvant provoquer une erreur [112]. La faute dans un système informatique représente soit un défaut d'un composant physique, soit un défaut d'un composant logiciel de ce système. Elle peut être créée de manière intentionnelle ou accidentelle, à cause des phénomènes physiques ou à cause des imperfections humaines. Durant l'exécution du système, la faute reste dormante jusqu'à ce qu'un événement intentionnel ou accidentel provoque son activation [100].

Erreur *(Error)* : l'activation d'une faute durant l'exploitation du système peut

se manifester par la présence d'un état interne erroné dans ce système, ce qui va donner un résultat incorrect ou imprécis par rapport à celui attendu [100]. Cet état peut rester non détecté longtemps (latence de la faute) mais peut conduire à court ou à long terme à une défaillance [57].
Défaillance *(Failure)* : elle survient lorsque le service délivré par le système ne correspond plus à sa spécification [112].
Les attributs de la sûreté de fonctionnement d'un système mettent plus ou moins l'accent sur les propriétés que doit vérifier la sûreté de fonctionnement du système. Ces attributs permettent d'évaluer la qualité du service fourni par un système.
Parmi ces propriétés, nous trouvons :
Disponibilité *(Availability)* : Probabilité pour qu'un système soit disponible à un instant t [112].
Fiabilité *(Reliability)* : Probabilité pour qu'un système soit continûment en fonctionnement sur une période donnée (entre 0 et t) [91].
Sûreté *(Safety)* : C'est une propriété qui respecte la non occurrence de défaillance catastrophique [112], similaire à la fiabilité mais par rapport aux conséquences catastrophiques causées par les fautes [91].
Sécurité-confidentialité *(Security)* : Cette propriété concerne l'occurrence des accès non autorisés ou l'acquisition non autorisée d'informations [112]. Cette propriété évalue la capacité du système à fonctionner en dépit de fautes intentionnelles et d'intrusions illégales [91].
Intégrité *(Integrity)* : L'intégrité d'un système définit son aptitude à assurer des altérations approuvées des données [91].
Maintenabilité *(Maintenability)* : Définit l'aptitude aux réparations et aux évolutions [112]. C'est la probabilité pour qu'un système en panne à l'instant 0 soit réparé à l'instant t [57].
La sûreté de fonctionnement est obtenue par l'utilisation de méthodes et de techniques permettant de fournir à un système l'aptitude à délivrer un service qui soit conforme à sa spécification et d'accorder une certaine confiance à cette aptitude.
Quatre classes de méthodes de traitement de fautes peuvent être distinguées [8] :
Prévention des fautes *(Fault prevention)* : cette méthode vise à empêcher l'occurrence ou l'apparition de fautes par le développement des systèmes informatiques de manière à éviter l'introduction de fautes de conception ou de fabrication et à empêcher que des fautes ne surviennent en phase opérationnelle [99].
Tolérance aux fautes *(Fault tolerance)* : elle consiste à délivrer un service correct en dépit de l'occurrence de fautes [112]. Le degré de tolérance aux fautes se mesure par la capacité du système à délivrer son service en présence de fautes [91].
Elimination des fautes *(Fault removal)* : cette méthode consiste à réduire le nombre et la sévérité des fautes dans le but de les éliminer du système [112].
Prévision des fautes *(Fault forecasting)* : elle consiste à estimer le nombre de fautes (physiques, de conception ou malveillantes) courantes et futures ainsi que leurs conséquences [112]. La sûreté de fonctionnement peut être illustrée par le schéma de la Figure 2.1.

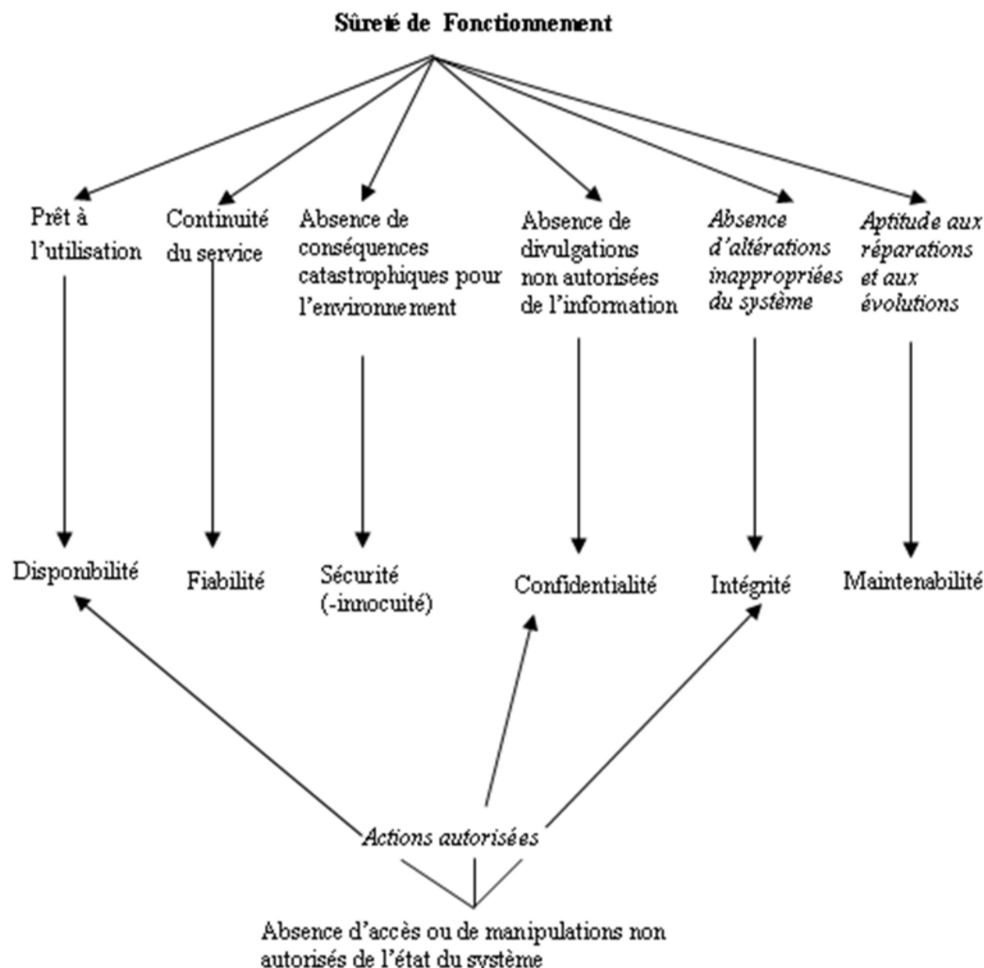

Figure 2.1 – Arbre de la sûreté de fonctionnement [116]

3 Taxonomie des fautes

Un critère pour classifier les fautes est la nature des fautes. En utilisant ce critère, nous distinguons généralement cinq types de fautes possibles [8]. Ainsi, nous pouvons distinguer ces fautes suivant qu'elles surviennent sur l'état ou sur le code d'un élément :

1. **Fautes d'état :** le changement des variables d'un élément peut être la conséquence de perturbations dûes à l'environnement (par exemple des ondes électromagnétiques), des attaques ou simplement des défaillances du matériel ou du logiciel utilisé. Il est par exemple possible que des variables prennent des valeurs qu'elles ne sont pas sensées prendre lors d'une exécution normale du système.
2. **Fautes de code :** le changement arbitraire du code d'un élément résulte la plupart du temps d'une attaque (par exemple le remplacement d'un élément par un adversaire malicieux), mais certains types moins graves peuvent correspondre à des bogues où à une difficulté à supporter la charge d'un élément

du système.

3. **Fautes franches ou de type crash :** à un point donné de l'exécution, un élément cesse définitivement d'être actif et n'effectue plus aucune action.
4. **Fautes d'omission :** à divers instants de l'exécution, un élément peut omettre de communiquer avec les autres éléments du système, soit en émission, soit en réception. Le composant cesse momentanément son activité puis reprend son activité normale.
5. **Fautes byzantines :** elles correspondent simplement à un type arbitraire de fautes, et sont donc les fautes les plus malicieuses et donc les plus complexes à tolérer. Un composant présentant ce type de faute agit de manière complètement imprévisible pour l'observateur extérieur.

Ces différents types de défaillance sont imbriqués les uns dans les autres. En effet, si un système peut tolérer que des composants agissent de manière totalement imprévisible (défaillance byzantine), il peut alors également tolérer un composant qui agit selon sa spécification mais qui omet des parties de son activité (défaillance par omission). De même, la défaillance franche est un cas particulier de défaillance par omission : l'omission concerne alors tout ce qui se passe après la défaillance.

4 Tolérance aux fautes dans les systèmes distribués

La tolérance aux fautes dans les systèmes distribués est un domaine de recherche qui a été et qui reste très largement étudié [145]. Les travaux dans ce domaine se différencient principalement selon trois critères : le type de fautes prises en compte (fautes de l'opérateur, fautes logicielles ou fautes matérielles), la technique de détection de fautes utilisée et l'approche de tolérance aux fautes proposée.

4.1 Etapes de la tolérance aux fautes

Plusieurs phases successives, non obligatoirement toutes présentes, font partie d'un processus de tolérance aux fautes [74] :

- **Détection :** Découvrir l'existence d'une faute (état incorrect) ou d'une défaillance (comportement incorrect).
- **Localisation :** Identifier le point précis (dans l'espace et le temps) où l'erreur (ou la défaillance) est apparue.
- **Isolation :** Confiner l'erreur pour éviter sa propagation à d'autres parties du système.
- **Réparation :** Remettre le système en état de fournir un service correct. Le composant défectueux est identifié et le système fonctionne comme si les composants défectueux ne sont pas utilisés ou sont utilisés d'une façon telle que la faute ne cause pas désormais une défaillance.

4.2 Techniques de détection des fautes

Les détecteurs de fautes sont un élément central dans les systèmes distribués tolérants aux fautes. La capacité d'un détecteur de fautes pour fonctionner de manière complète et efficace, en présence d'une messagerie non fiable ainsi que des composants sujetés à une forte occurance de fautes, peut avoir un impact majeur sur la performance de ces systèmes. "La complétude" est la garantie que la défaillance d'un membre du système soit finalement détectée par tous les autres membres. "L'éfficacité" signifie que les défaillances sont détectées rapidement et avec une précision acceptable. Le premier travail pour répondre à ces deux propriétés était par Chandra et Toueg [34]. Les auteurs ont montré l'impossibilité pour tout algorithme de détection de fautes d'atteindre à la fois la complétude et l'efficacité dans un système non fiable et asynchrone. Cette impossibilité résulte de la difficulté inhérente de déterminer si un processus à distance s'est réellement défaillant ou si ses transmissions sont simplement retardées. Il est donc impossible de mettre en œuvre un service de détection de fautes fiable sans faire plus d'hypothèses sur le système. Ce résultat a lancé une vague de recherches théoriques pour la classification des détecteurs de fautes.

4.2.1 Cas des systèmes synchrones/asynchrones

Dans un système synchrone, détecter une défaillance est une issue triviale. Puisque les délais sont liés et connus, les défaillances sont détectées à l'aide d'un délai de garde.
Un système asynchrone est un système pour lequel il n'y a aucune hypothèse temporelle sur les temps de transmission des messages ou sur les temps de calcul des processeurs. Comme dans le cas d'absence de communication en provenance d'un processus pendant une durée t, celui-ci est considéré potentiellement défaillant (suspect). Un temporisateur (délai de garde) est amorcé et un message spécial est envoyé à ce processus. En l'absence de réponse avant la fin du temporisateur, le processus est jugé défaillant et la tolérance à cette faute peut commencer. Cette technique simple n'est pas optimale, puisqu'elle a pour inconvénient de distinguer difficilement un processus lent d'un processus mort.

4.2.2 Messages "Ping/Pong"

De manière périodique ou à la demande, les nœuds envoient un message 'Ping' à tous les autres nœuds ou à une partie d'entre eux. Cette technique peut permettre une détection plus ciblée : elle permet de ne surveiller qu'un sous-ensemble de nœuds. En revanche, pour obtenir autant d'informations qu'avec la technique d'échange de messages de vie, deux fois plus de messages sont nécessaires (chaque message de vie 'Pong' étant réclamé explicitement par un message 'Ping') [137].

4.2.3 Echanges de messages de vie (*heartbeats*)

Chaque nœud envoie périodiquement un message de vie à tous les autres et attend donc, à chaque période, un message de vie de chacun d'entre eux. Lorsqu'un nœud

ne reçoit pas de message de vie d'un autre nœud, il le considère comme défaillant. De nombreux projets de recherche se sont concentrés sur la fiabilité de la suspicion de défaillance, en prenant en compte la variation de latence dans l'arrivée des messages de vie d'un nœud particulier [137].

4.3 Techniques de tolérance aux fautes

La majorité des techniques de tolérance aux fautes sont basées sur une duplication spatiale qui consiste à affecter un job à plusieurs nœuds, une duplication temporelle de l'exécution des jobs qui consiste à capter des états d'exécution de ces jobs pour les revenir en cas d'une défaillance ou bien une duplication informationnelle (redondance de données, codes, signatures) [91].

4.3.1 Tolérance aux fautes par duplication

Trois approches fondamentales du problème de la tolérance aux fautes, basées sur la duplication [80], sont proposées dans la littérature : l'approche masquante à base de la redondance active, l'approche recouvrante à base de la duplication passive et l'approche hybride à base de la duplication passive et active.

Duplication active : La duplication active peut être utilisée de manière efficace pour masquer la faute de plusieurs composants matériels (capteurs, processeurs, média de communication et actionneurs) dans un système distribué. Elle est bien adaptée à toutes les hypothèses de défaillance : silence (latence) sur défaillance, fautes transitoires, fautes temporelles et fautes byzantines (ou quelconques). Par exemple, lorsqu'une défaillance d'un processeur se produit, toutes les tâches exécutées sur ce processeur deviennent inactives, ce qui conduit à une défaillance du système. Afin de tolérer ces fautes, la duplication active permet de masquer k fautes de processeurs en répliquant activement chaque tâche t sur $k+1$ processeurs distincts. Pour cela, chaque réplique t_i de t doit recevoir ses données d'entrées en $k+1$ exemplaires. Par exemple, dans la Figure 2.2 b, la tâche A (resp. B) est répliquée en deux exemplaires, qui sont allouées à deux processeurs distincts *P1* et *P2* (resp. *P2* et *P4*) afin de masquer une faute de processeur. Dans cet exemple, la réplique *B2* de B reçoit ses données d'entrée en deux exemplaires (message m). De même, la perte

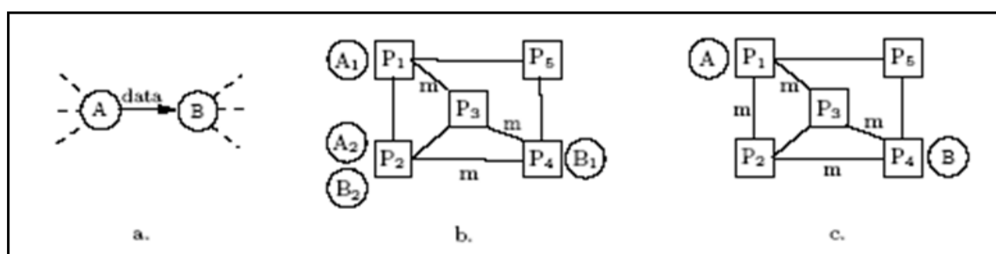

FIGURE 2.2 – Duplication active [199]

d'un message dans un réseau de communication dûe aux fautes de déconnexion, peut être masquée par la transmission de ce message via plusieurs routes disjointes.

Par exemple, dans la Figure 2.2 c, la communication $A.B$ (message m) est réalisée sur deux routes disjointes reliant le processeur *P1* à *P4*. Précisément, le processeur *P1*, exécutant l'opération A, envoie au processeur *P4*, exécutant l'opération B, la communication $A.B$ via les deux routes disjointes *P1,P2,P4* et *P1,P3,P4*.

Duplication passive : Comme dans la duplication active, afin de tolérer k fautes de processeurs ou de déconnexions, chaque tâche t est répliquée sur *k+1* processeurs distincts. Cependant, une seule copie t_1, appelée primaire, est exécutée tandis que les autres copies t_i $(i \neq 1)$, appelées sauvegardes ou secondaires, surveillent la copie primaire. Si le processeur exécutant la copie primaire devient défaillant, une copie de sauvegarde sera sélectionnée pour remplacer la copie primaire. Par exemple, dans la Figure 2.3 a, la tâche A (resp. B) est répliquée en deux exemplaires, qui sont alloués à deux processeurs distincts *P1* et *P2* (resp. *P2* et *P4*) afin de tolérer une faute de processeur ou de déconnexion. Dans cet exemple, afin de tolérer une faute de

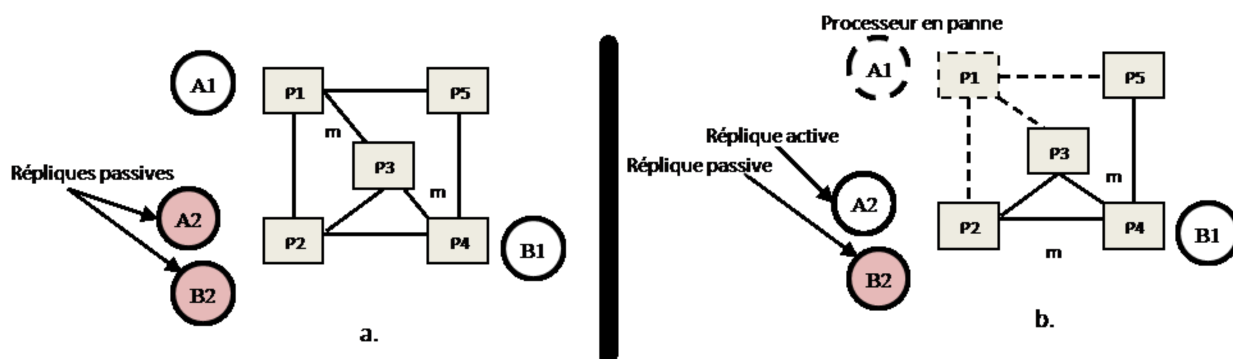

Figure 2.3 – Duplication passive

déconnexion, le processeur $P1$, exécutant la copie primaire $A1$, envoie au processeur $P4$, exécutant la copie primaire $B1$, la communication $A.B$ via une seule route. Si le processeur $P1$ est défaillant, le processeur $P2$ détecte la faute, exécute la copie de sauvegarde $A2$ et envoie le message m à $P4$ via une nouvelle route, comme cela est montré dans la Figure 2.3 b. La perte d'un message, dans le cas de la duplication passive des communications, dûe aux fautes de déconnexion, peut être tolérée par la retransmission de ce message via des routes disjointes dans un réseau. Cette technique de duplication nécessite des mécanismes particuliers de détection de fautes, et le choix du primaire ainsi que son intégration après la réparation.

Duplication hybride : La duplication hybride [52] est une combinaison de la duplication active et passive, dans laquelle par exemple, on utilise la duplication active pour les tâches et la duplication passive pour les communications, comme cela est montré dans la Figure 2.4. Dans cette figure, les deux tâches A et B sont répliquées activement en deux exemplaires, tandis que la communication $A.B$ ne sera envoyée que par la première réplique $A1$ de A (message m). Une propriété intéressante de la duplication active se situe dans le fait qu'une faute n'augmente pas la latence d'un système temps-réel, ce qui n'est pas le cas dans la duplication passive, où la faute de la réplique primaire peut de manière significative augmenter la latence du système. Cependant, la duplication passive présente l'avantage de réduire la surcharge sur les processeurs et les média de communication, ce qui permet une meilleure

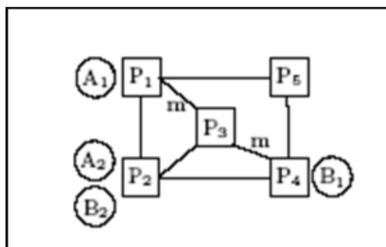

FIGURE 2.4 – Duplication hybride

exploitation des ressources matérielles offertes par l'architecture. Les duplications active et passive sont deux techniques complémentaires [52]. Ainsi, combiner efficacement ces deux techniques, pour tolérer les fautes des processeurs et/ou les fautes de déconnexion, est intéressant.

4.3.2 Technique *Rollback-Recovery*

Le Rollback-Recovery *(RR)* est une technique pour rendre un système distribué plus fiable et tolérant aux fautes. Deux approches sont essentiellement utilisées pour implémenter un système de *RR* [50]. Ces deux techniques sont basées sur les points de reprise (*checkpoint*) ou sur les fichiers logs. A la suite d'une faute, le rollback-recovery restaure l'état du système associé au plus récent ensemble consistant de points de reprise, qui est appelé ligne de recouvrement *(Recovery Line)* [157]. Le roolback basé sur les fichiers logs (ou fichiers de trace) utilise le fait qu'une exécution d'un processus peut être modélisée comme étant une séquence d'intervalles d'états déterministes, où chacun commence par l'exécution d'un événement indéterministe [190]. Un tel événement peut être la réception d'un message d'un autre processus ou un événement interne au processus lui-même, comme par exemple les événements probabilistes [136]. L'envoi d'un message, par contre, n'est pas un événement indéterministe [35]. Le *RR* basé sur l'enregistrement suppose que tous les événements indéterministes peuvent être identifiés et que les déterminants correspondants peuvent être enregistrés sur un support fiable de stockage. Durant l'exécution, chaque processus enregistre les déterminants de tous les événements indéterministes qu'il observe sur un support fiable de stockage. Un déterminant est un ensemble d'informations qui permet de ré-exécuter les programmes de telle sorte que les évènements indéterministes soient exécutés à chaque fois de la même façon (leurs effets sur le déroulement du programme est le même). De plus, chaque processus effectue des points de reprise pour réduire le rollback à la suite d'une faute et avant la ré-execution. A la suite d'une faute, le processus défaillant est ré-exécuté en utilisant les points de reprise et les déterminants déjà enregistrés pour se comporter de la même façon que l'exécution défaillante (notamment l'ordre de réception des messages).

5 Tolérance aux fautes dans les grilles de calcul

Les grilles de calcul sont des systèmes à grande échelle dont les principales caractéristiques sont l'hétérogénéité, le passage à l'échelle et la dynamicité [60]. La nature dynamique des grilles est à la fois un atout et un défi. Un atout car elle permet d'ajouter des ressources au fur et à mesure de leur disponibilité ou d'en retirer pour différents besoins. Un défi, car pour le concepteur du système, cette nature dynamique soulève de nombreux problèmes. Les conséquences de la nature dynamique des grilles de calcul sont multiples, la première étant directement liée à une caractéristique des grilles : la grande échelle. Plus le nombre de nœuds est grand, plus la probabilité d'occurrence d'une faute est importante [84].
Par exemple, Considérons un système composé de n processeurs. Par définition la fiabilité $R(t)$ de ce systéme est la probabilité qu'il soit opérationel pour tout instant $t_i \in [0, t]$. Les auteurs de [185] montrent que $R(t) = e^{-\lambda tn}$ où λ est une constante représentant l'intensité de défaillance sur un processeur. La constante λ est l'inverse du temps moyen entre deux fautes, notée *(Mean Time Between Failures MTBF)* du système, $\lambda = \frac{1}{MTBF}$. La probabilité $F(t)$ que le système ne soit pas opérationel est la non fiabilité du systéme sur l'intervalle de temps [0, t] ; elle s'écrit [185] $F(t) = 1 - R(t)$. Si nous considérons un $MTBF$= 2000 jours pour chaque processeur et le temps d'exécution varie d'un jour à 30 jours. La probabilité de défaillance augmente considérablement en fonction du nombre de processeurs utilisé dans le système et de leurs temps d'exécution (voir Figure 2.5).
Dans un cadre général, quand nous prenons en compte la nature dynamique d'un système distribué, nous ne pouvons plus émettre l'hypothèse qu'un nœud ne puisse rester indéfiniment fonctionnel. Lorsqu'un nœud s'arrête (à cause d'une défaillance ou d'un arrêt volontaire), il ne joue plus son rôle dans le système. En particulier, les données qu'il stockait ne sont plus accessibles, les chemins réseau qui passaient par lui sont coupés, les nœuds qui communiquaient avec lui risquent d'être bloqués, etc. Il est donc nécessaire de prendre des précautions afin que le système puisse continuer d'évoluer malgré la faute. Il n'existe pas de solution générique pour faire face à tout degré de défaillance. Les solutions classiques consistent à utiliser de la duplication (au niveau des liens réseau afin d'avoir plusieurs routes possibles, au niveau des données afin de pouvoir conserver une copie dans le système, etc.) ou de sauvegardes périodiques. La tolérance aux fautes est l'un des moyens de la sûreté de fonctionnement dont le principe consiste à réagir "à chaud " aux états erronés d'un système et à empêcher que ces erreurs ne conduisent à un dysfonctionnement visible pour l'utilisateur du système considéré [116].

5.1 Modèle de fautes pour les grilles de calcul

Avec la croissance de la fonctionnalité, la complexité et le passage à l'échelle des grilles de calcul, les défaillances de ressources sont inévitables. Pour les applications scientifiques, les défaillances peuvent entraîner une dégradation des performances fréquente, ou dans le pire des cas, l'arrêt prématuré de l'exécution ou la corruption de données et leurs pertes. Pour les applications commerciales, les défaillances peuvent entraîner la violation des accords au niveau de service, et causer une perte dévastatrice de clients et des revenus [88].

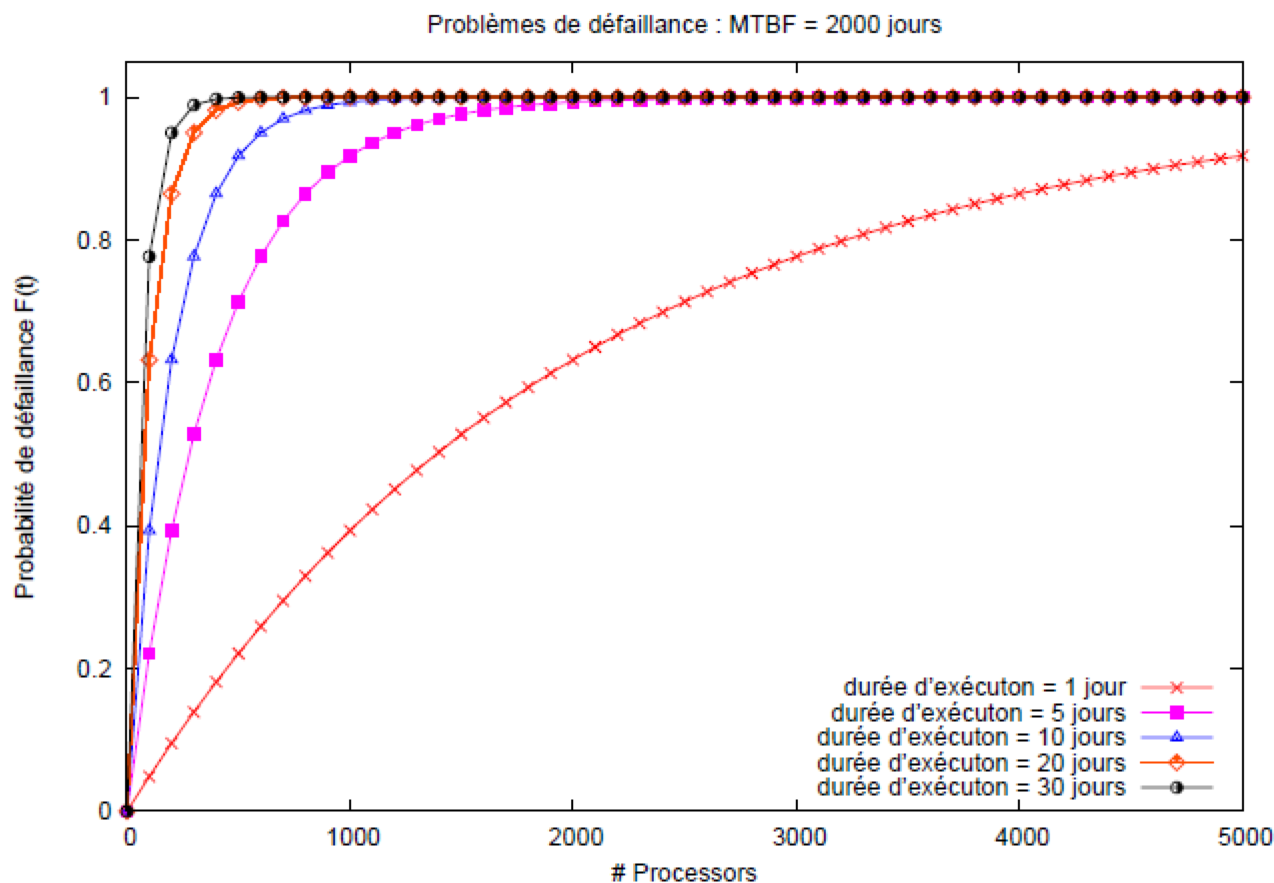

FIGURE 2.5 – Probabilité de dé faillance dans les environnements parallèles ré, où λ=0.0005 [91]

Kola et al. [104] ont analysé des fichiers logs de quatre applications de production des données intensives. Ils ont déterminé que les défaillances se produisent plus souvent de :

1. Fautes intermittentes du réseau étendu.
2. Défaillances de transfert de données :
   a Suspension indéfinie (généralement due à des accusés de réception manquant dans les transferts FTP).
   b Corruption de données (généralement dû à un matériel défectueux).
   c Crash de serveur de stockage (en raison du nombre élevé des transferts simultanés de données d'écriture).
3. Fautes accidentelles des machines, les temps d'arrêt pour les mises à niveau matérielles/logicielles et les corrections de bugs.
4. Mauvais comportement des machines causé par une mauvaise configuration et/ou les bugs logiciels.
5. Espace disque insuffisant pour des fichiers d'entrée ou pour écrire dans des fichiers de sortie.

Même un scénario simplifié de soumission de job dans une grille, qui est un processus qui passe par plusieurs couches de la grille. Les défaillances peuvent se produire au cours de nombreuses étapes de ce processus [133] :

1. **Défaillance du service d'exécution :** le service d'exécution de l'application tels que le gestionnaire des jobs dans la grille (une composante d'oronnancement entre les brokers de ressources), un broker de ressources ou un ordonnanceur local de tâches pourraient se conduire à un crash ou ne pourraient pas être en mesure de satisfaire les exigences du job.
2. **Défaillance dans l'environnement local :** l'application peut tomber en panne à cause d'une défaillance dans l'environnement d'exécution local, telle que le redémarrage de la machine, la défaillance de la machine virtuelle ou l'épuisement de tout l'espace disque disponible.
3. **Défaillance de ressources :** l'application peut dépendre d'une ressource qui tombe en panne, par exemple le réseau pourrait être en congestion, le web-service ne répond pas ou le transfert de données n'est pas arrivé à terme.
4. **Défaillance spécifique de l'application :** L'application peut tomber en panne à cause d'un bogue dans le code, par exemple effectuer une opération illégale conduisant à une faute.

Dans les grilles de calcul, il est nécessaire d'évaluer les possibilités d'occurrence de défaillances pour les différents types de fautes et pour les développeurs d'application. En effet, il ne serait pas judicieux de mettre en place des mécanismes coûteux servant à tolérer des défaillances peu probables.
De manière générale, les défaillance dans une grille de calcul peuvent être des :

5.1.1 Fautes franches

Le grand nombre de nœuds présents dans une grille de calcul implique une éventualité non négligeable de fautes (défaillances franches) de ces nœuds. Des fautes simultanées sont également envisageables.

5.1.2 Fautes par omission

La présence de certains équipements réseau tels les routeurs, notamment entre les différentes institutions composant une grille de calcul, rend possibles certaines défaillances par omission. Lorsque le réseau est congestionné, les routeurs ignorent en effet volontairement des messages pour limiter la congestion. Des pertes de messages peuvent également avoir lieu lorsqu'un nœud reçoit des paquets réseau plus vite qu'il ne peut les traiter.

5.1.3 Fautes byzantines

Les grilles de calcul sont le résultat de la mise en commun des ressources d'institutions qui s'accordent mutuellement une certaine confiance et qui mettent en place des mécanismes coordonnés de contrôle d'accès. Les grilles de calcul sont par conséquent des environnements relativement clos et sécurisés. La tolérance des fautes

byzantines requiert la mise en place de mécanismes réellement coûteux, engendrant de nombreuses communications. Lors de la mise en place de mécanismes de tolérance aux fautes, il est nécessaire de prendre en compte aussi bien les coûts de ces mécanismes par rapport aux possibilités d'occurrence des différents types de défaillances. Au sein des grilles de calcul, il semble judicieux d'offrir des mécanismes permettant de supporter des défaillances de type franches ainsi que des pertes de messages. En effet, ces défaillances sont très courantes. En revanche, des mécanismes coûteux permettant de supporter des défaillances byzantines ne devraient être mis en place que lorsque cela est vraiment nécessaire.

5.1.4 Fautes des middlewares

Un middleware permet la communication entre les clients et les serveurs ayant des structures et des implémentations différentes. Il permet l'échange d'informations dans tous les cas et pour toutes les architectures. Enfin, le middleware doit fournir un moyen aux clients pour trouver leurs serveurs, aux serveurs pour trouver leurs clients et en général de trouver n'importe quel objet atteignable [214]. Les principaux types de défaillances sont liés à l'environnement de configuration. Selon certaines études [105], le manque de contrôle des ressources sur la grille est la principale source des fautes de configuration. Par la suite, nous avons les fautes liées aux middlewares avec 48%, les fautes associées aux applications avec un taux de 43% et enfin, les défaillances matérielles avec un taux de 34% [105].

5.2 Analyse des défaillances dans les grilles de calcul

Compte tenu de la complexité des grilles de calcul et les facteurs sujets aux défaillances, il est important pour tout système de gestion des grilles de calcul d'inclure une certaine "intelligence" pour faire face à ces défaillances. Une analyse approfondie des défaillances est cruciale pour comprendre les propriétés statistiques et donner un aperçu sur l'évaluation des solutions qui peuvent améliorer la qualité de service au niveau de la grille [70, 71, 106]. Cependant, peu de travaux ont été présentés pour l'analyse des défaillances dans les grilles de calcul parce qu'il est, en général, difficile de recueillir des traces au niveau de toute la grille. Dans [180], les auteurs ont analysé les données de défaillance couvrant des systèmes à grande échelle des sites de calcul de haute-performance. Les données ont été recueillies au cours de 9 années à Los Alamos National Laboratory et comprend 23.000 défaillances enregistrées sur plus de 20 systèmes. Ils ont étudié les statistiques des données, y compris la cause de défaillances, le temps moyen entre deux fautes, et le temps moyen de réparation. Ils ont trouvé, par exemple, que les taux de défaillance moyens diffèrent énormément entre les systèmes, allant de 20 à 1000 défaillances par an, et que le temps entre deux fautes est bien modélisé par une distribution de Weibull avec la diminution de taux de risque. Le temps moyen de réparation varie, d'un système à l'autre, de moins d'une heure à plus d'un jour, et les temps de réparation sont bien modélisés par une distribution log-normale.

Dans [88], une première analyse des défaillances des jobs dans une grille de données a été réalisée sur la grille de calcul mondiale WLCG *Worldwide LHC Computing*

Grid pour le LHC (*Large Hadron Collider*), qui est une collaboration entre plusieurs centres de calcul répartis dans le monde entier [121]. Elle a pour but de fournir les ressources nécessaires au stockage, à la distribution et à l'analyse des 15 pétaoctets (15 millions de gigaoctets) de données générées chaque année par le LHC. Presque tous les jobs sont des tâches indépendantes et trivialement parallèles, nécessitant une puissance de calcul pour traiter une grande quantité de données. Le système de Monitoring en temps réel de WLCG était opérationnel depuis Octobre 2005 [119]. L'étude était réalisée sur trois périodes. Les périodes sélectionnées sont du 20 au 30 Novembre, du 7 au 10 Décembre et du 19 au 30 Décembre. Le taux de soumission quotidien des jobs, des deux premières périodes, est assez stable avec une moyenne de 22000 jobs. La troisième période a reçu quotidiennement de 36000 à 60000 jobs. Ils ont constaté que le taux global de défaillance des jobs est important, allant de 25% à 33% parmi tous les jobs soumis. Plusieurs modèles statistiques ont été évalués pour les défaillances entre les temps d'arrivée. Il est démontré que cette défaillance suit les distributions MMPP4 (*Markov Modulated Poisson Process*).
Pour faciliter la conception, la validation et la comparaison des modèles et des algorithmes à tolérance aux fautes, les auteurs dans [107] ont créé un archive des traces de défaillance (*Failure Trace Archive* FTA) comme un dépôt public en ligne de traces de disponibilité prélevés sur divers systèmes distribués et parallèles. Ils ont décrit la conception de l'archive, en particulier le format du standard FTA et la conception d'une boîte à outils qui facilite l'analyse automatisée des traces des ensembles de données. En utilisant la boîte à outils, les auteurs ont réalisé une analyse statistique uniforme et globale des défaillances dans neuf systèmes distribués. Le résultat clé, de cette analyse, est que les distributions de Weibull et Gamma sont souvent les meilleurs candidats pour la disponibilité et l'indisponibilité. L'archive des traces de défaillance, y compris la documentation technique sur le format de données, la boîte à outils et les ensembles de données de trace sont disponibles en ligne à l'adresse : http ://fta.inria.fr.

5.3 Détection de fautes dans les grilles de calcul

S'appuyant sur les travaux antérieurs sur la détection des fautes et des défaillances dans les systèmes distribués [84, 116, 205], les chercheurs ont étudié les méthodes scalables de détection de fautes pour les environnements de grille. Des travaux ont également porté sur les techniques de reconnaissance des différents types de fautes, particulièrement les fautes difficiles à détecter, qui devraient se produire entre les ressources hétérogènes dans des conditions dynamiques, y compris l'isolation des fautes et les méthodes de diagnostic [54].

5.3.1 Limites des méthodes actuelles de détection de fautes

Les méthodes de détection de fautes développées pour les systèmes distribués ne conviennent pas généralement aux systèmes de grille à grande échelle, dynamiques et hétérogènes. Une des raisons est que les protocoles disponibles de surveillance du réseau et des outils, tels que ceux basés sur le protocole SNMP (*Simple Network Management Protocol*) [154], s'appuient sur une connaissance approfondie de

la structure du réseau. Ces informations sont susceptibles d'être toujours disponibles dans des environnements dynamiques et scalables composés de différents domaines administratifs et de sécurité. Un autre problème bien connu pour la détection des défaillances se produit dans les systèmes distribués asynchrones, quand les fonctions de gestion (y compris les détecteurs de défaillance) sont décentralisées et peuvent elles-même tomber en panne. Ces conditions sont susceptibles d'être rencontrées dans les systèmes de grille à grande échelle. Les recherches actuelles dans les grilles de calcul reflètent aussi le manque de capacités de détection des fautes dans les environnements scalables. Zanikolas et Sakellariou [224] ont conduit une enquête sur 19 systèmes de surveillance des grilles basés sur l'architecture GGF de monitoring de grille [195], qui prescrit les exigences pour une fonction de surveillance afin de supporter la détection de fautes. L'enquête a révélé que la plupart de ces systèmes ont été conçus sans mécanismes de détection de fautes. En outre, la plupart n'ont pas la possibilité de passer à l'échelle dans leurs fonctions de surveillance. Ces lacunes ont motivé la recherche sur les détecteurs de fautes dans les grilles de calcul.

5.3.2 Méthodes de détection de fautes scalables

Les premiers travaux sur un détecteur distribué de fautes pour le middleware Globus [187] abordent les issues de complétude et de précision [154] en permettant aux détecteurs de défaillances, potentiellement peu fiables, de communiquer l'information sur la probabilité de défaillance d'une ressource. L'approche a également été conçue pour améliorer la flexibilité, l'efficacité et la scalabilité par le découplage du monitoring, la détection et les fonctions de notification. Dans [92], un détecteur de fautes a été proposé pour accomplir la scalabilité en organisant les ressources de la grille en groupes de vie sur la base de la topologie du réseau. Horita et al. [87] ont proposé un système de détection de fautes scalable et auto-organisé basé sur les travaux antérieurs des protocoles de l'appartenance à un groupe et de détecteurs de fautes dans les systèmes distribués [44, 84]. Dans cette approche, chaque processus est surveillé par un petit groupe (4 ou 5) de processus choisis au hasard sur des nœuds distants. Le processus de monitoring établit une connexion au processus surveillé et transmet périodiquement des messages de vie. Il en résulte la création d'un sous-réseau virtuel de surveillance dans une grille, qui peut être constitué de ressources hétérogènes. Dans [87], les ressources affectées à une application sont placées dans des domaines distincts où ils émettent des messages de vie aux moniteurs des domaines, qui sont organisés de façon hiérarchique.

Nous classifions les approches de détection de fautes scalables comme suit :

1. **Approche hiérarchique :** Dans [20], les auteurs proposent une version hiérarchique de détecteurs de défaillances utilisant des échanges de messages de vie [19, 21]. L'ensemble des nœuds est partitionné en sous-ensembles et les échanges de messages de vie de type 'tous-vers-tous' sont restreints à un sous-ensemble. Chaque sous-ensemble choisit un représentant et les différents représentants s'échangent des messages de vie. Les auteurs de [135] proposent un mécanisme de détection de fautes en intégrant avec chaque message de vie un temps estimatif d'arrivée. Le modèle comprend trois composantes principales qui sont un gestionnaire de nœuds (*Node Manager NM*), un serveur

d'index (*Index Server EST*) et un contrôleur de messages de vie (*Heartbeat Monitor HBM*) ; (i) Tout d'abord, chaque nœud membre doit s'inscrire au niveau du EST en stockant et mettant à jour les informations qui doivent être surveillées. Une fois un nœud est inscrit sur le NM, l'information fournie par le nœud sera ajoutée et indexée. Le gestionnaire d'index (*Index Manager IM*) va recevoir une notification lorsque le nœud est ajouté à l'index ou modifié depuis la dernière indexation. (ii) L'IM sera avisé chaque fois que le NM enregistre un nouveau nœud ou met à jour les informations de nœud depuis la dernière indexation. Pendant le processus de surveillance, l'IM reçoit un fichier journal généré par HBM pour chaque message de vie. (iii) L'HBM est chargé de surveiller l'état des nœuds enregistrés. Dans le cas d'une anomalie de leurs comportements habituels, il informe l'IM pour prendre l'action nécessaire. Chaque nœud envoie périodiquement un message indiquant sa vitalité à l'HBM. Cette approche permet d'obtenir des détecteurs de défaillances adaptés aux systèmes à grande échelle comme les grilles.

2. **Approche probabiliste et apprentissage automatique :** Une autre approche permettant de concevoir des détecteurs de défaillances pour les systèmes à grande échelle est l'utilisation de techniques probabilistes. Dans SWIM [44], les nœuds envoient des messages de type "Ping" périodiquement à un ensemble aléatoire de nœuds. De plus, les messages "Ping" et "Pong" contiennent des informations sur les nœuds suspects permettant ainsi de propager les détections. Les mécanismes probabilistes n'utilisent pas la notion de représentant et ne s'appuie pas sur une topologie spécifique. Leur maintenance est donc plus aisée, ce qui les rend bien adaptés aux systèmes à grande échelle. En revanche, les défaillances sont détectées avec une forte probabilité mais non avec certitude.
   La capacité à identifier de manière autonome les anomalies a également été démontrée en utilisant les cartes auto-organisatrice de Kohonen [103], et les modèles de Markov cachés (HMM) dans [177]. Ces approches mettent l'accent sur les comportements prédictifs en tirant parti des données historiques recueillies à partir d'un système local. Schneider et al., dans [178], étendent leur approche proposée dans [177] pour évaluer de manière autonome la source d'une faute dans un système en utilisant les Machines de Boltzmann restreintes (*Restricted Boltzmann Machines MRA*) pour prédire l'état d'une caractéristique. Cette prédiction peut être utilisée pour identifier la source d'une faute via une comparaison entre le résultat attendu et le résultat fourni par le système.

3. **Approches basées sur des modèles mathématiques :** Ces modèles ont émergé comme des approches éminentes pour le diagnostic des fautes des systèmes à temps discret ou continu [64, 208]. Ces approches sont basées sur des modèles mathématiques du processus contrôlé, de sorte que les résidus peuvent être calculés en prenant la différence entre les valeurs estimées des variables de sortie du système et les valeurs mesurées. Les résidus sont ensuite comparés à des seuils appropriés pour la détection des fautes afin de fournir une décision assurant le bon fonctionnement du système. Dans [22], un modèle distribué pour la détection de fautes est développé pour les systèmes à large échelle. L'architecture du détecteur de fautes est composée de deux couches :

(i) le système physique, décomposé en n sous-systèmes et (ii) l'architecture de détection de fautes se décompose ainsi de n entités des diagnostiqueurs de fautes locaux (*Local Fault Diagnosers LFD*). Chaque LFD est chargé de surveiller exactement un sous-système : en prenant des mesures locales et en communiquant seulement avec les LFD voisins. Pour la détection des fautes, chaque LFD est équipé d'un estimateur adaptatif non-linéaire de l'état local. L'estimateur local détecte la faute sur la base de la différence entre les valeurs estimatives et les sorties du système.
Les auteurs dans [182] ont exploité les données du système sous forme de séries chronologiques, pour développer une approche de détection de fautes. L'approche modélise les données sous forme de relations invariables dans le temps en utilisant des modèles autorégressifs avec des entrées exogènes. Comme il est illustré dans la Figure 2.6. Cette approche est composée de deux phases :
Phase I : Modélisation par les invariants : La figure 2.6 (a) montre le graphe de relations invariantes extraites d'un ensemble de données de séries chronologiques du monde réel comprenant 20 indicateurs. Chaque nœud dans le graphe représente une métrique et un lien entre deux nœuds dénote une relation invariante entre eux. Ce graphe invariant a 39 arêtes et cinq composantes connexes. La plus grande composante connexe se compose de 11 nœuds, et se compose de deux sous-composantes denses connectés avec un nœud commun. Il ya trois invariants isolés, les nœuds de degré 1, et une petite composante connectée avec trois nœuds. Par conséquent, nous pouvons construire un modèle de fonctionnement du système à partir des données de mesure de la série chronologique en utilisant les invariants, et de le représenter sous forme d'un graphe. La structure de ce graphe invariant (par exemple des composants connectés, distribution des degrés, etc.) fournit des informations globales sur les dépendances entre les paramètres.
Phase II : Détection de fautes en utilisant les invariants : Si à un certain moment les données de mesure du système ne s'adaptent pas à une partie (ou à la totalité) des relations invariantes, i.e. les invariants ne sont pas vérifiés, alors une faute ou une anomalie est détectée. La figure 1 (b) montre les invariants manquants (lignes pointillées) à un certain moment. Nous pouvons voir que tous les invariants brisés ont la métrique $m1$ en commun, et dans ce cas, le composant associé à $m1$ est susceptible d'avoir une faute. Cette approche a été appliquée pour la détection des fautes et d'anomalies dans plusieurs systèmes informatiques [36, 94].

4. **Approches basées sur la fouille de données :** L'exploitation des techniques de la fouille de données pour la détection des fautes dans les système à large échelle a fait l'objet de plusieurs travaux de recherche [65]. Chen et al. [37] présentent une approche basée sur l'apprentissage supervisé des arbres de décision. Le système nécessite des exemples étiquetés de défaillance et une connaissance du domaine. Pelleg et al. [147] explorent la détection de fautes des machines virtuelles en utilisant des arbres de décision. Le système supervisé, nécessite un apprentissage sur des exemples étiquetés et des compteurs sélectionnés manuellement. Sahoo et al. [169] comparent trois approches de prédiction des fautes : à base de règles, réseau bayésien et d'analyse de séries chronologiques. Ils ont appliqué avec succès leurs méthodes sur un cluster de

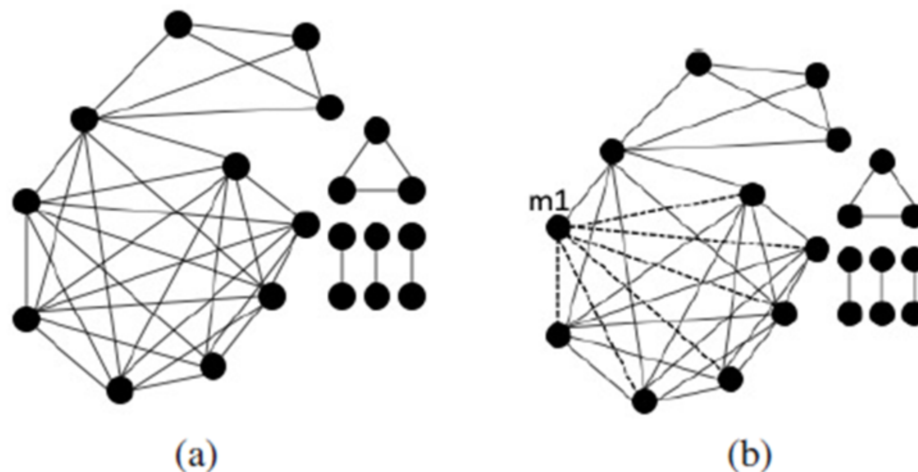

FIGURE 2.6 – Exemple de détection de fautes par les invariants du graphe [182]

350 nœuds pour une période d'une année. Leurs méthodes sont supervisées et s'appuient en outre sur la connaissance substantielle du système surveillé. Cohen et al. [39] induisent une classification par les réseaux bayésienx basée sur un arbre augmenté (*Tree-augmented Bayesian network*). Bien que cette approche ne nécessite pas de connaissances de domaine autre que d'un ensemble d'apprentissage étiqueté, mais il est sensible à l'évolution des charges de travail.

5. **les protocoles Gossip [67] :** Ces protocoles sont basés sur du gossiping, inspirés du service de détection de panne proposé par Renesse et al. [207]. Le principe du gossiping est que chaque élément du système distribué à surveiller est accompagné d'un détecteur local. Ces détecteurs locaux s'échangent de manière périodique, l'état de vitalité de ressource qu'ils ont en responsabilité, mais aussi leurs connaissances les plus récentes de l'état de vitalité concernant les ressources distantes. Chaque détecteur matérialise cette connaissance globale par une table associant à chaque ressource un compteur de messages de vie. Lors d'un échange, un détecteur reçoit une ou plusieurs de ces tables, et garde pour chaque ressource, le message de vie le plus élevé communiqué, ou le dernier message de vie observé s'il s'agit de sa propre ressource. Il ré-émet ensuite la table vers un certain nombre d'autres détecteurs. Les Protocoles de gossip sont généralement régis par trois paramètres clés : le temps de gissip, le temps de nettoyage et le temps de consensus. Temps de Gossip est l'intervalle de temps entre deux messages de gissip consécutifs. Le temps de Nettoyage est l'intervalle de temps après lequel un hôte est soupçonné d'être défaillant. Enfin, le temps de consensus est l'intervalle de temps après lequel le consensus est atteint d'un nœud défaillant. Sur chaque détecteur, quand la différence dans le nombre de messages de vie entre une ressource distance et la ressource locale dépasse un seuil, une panne est suspectée. Le détecteur entre alors dans un consensus pour avoir confirmation que les autres détecteurs ont également suspecté la panne. S'il y a consensus sur la panne, chaque détecteur peut en informer sa ressource locale. Dans la pratique, la difficulté est de choisir une

bonne valeur de seuil : une grande valeur rend le temps de détection long, une petite valeur provoque des fausses détections. Le choix du seuil est lié à la stratégie de routage des messages de gossip, car il faut s'assurer qu'un message donnant une information de l'état exact de vitalité d'une ressource ait le temps de parvenir à tous les détecteurs. Un routage déterministe Binary Round-Robbin a été proposé pour améliorer ce protocole [68].

5.3.3 Difficultés de détection de fautes

Dans les environnements de type grille hétérogène et scalable, certains types de fautes peuvent être difficiles à détecter. Kola et al. [105] ont rapporté les travaux sur l'élaboration d'un modèle de fautes "silencieux" dans le système Condor. Ces travaux se sont aussi intéressés sur l'utilisation des algorithmes de consensus afin de déceler les fautes byzantines. Dans [134, 213], des composants répliqués comparent les résultats de l'exécution de calculs identiques en même temps pour détecter les composants affectés par des fautes byzantines. Parce que les fautes byzantines sont difficiles à détecter et récurrents, elles peuvent perturber les calculs au fil du temps. Enfin, dans les grands systèmes distribués, des fautes peuvent provenir d'un composant et se propager à travers le système, créant potentiellement des effets en cascade. Jusqu'à présent, il semble que peu de chercheurs aient abordé cette classe de fautes dans les systèmes de grille. Un exemple est présenté dans [167], où une méthode de diagnostic a été proposée pour l'exécution de tests temps-réel sur les éléments interdépendants d'un réseau afin d'isoler l'origine de la faute et déterminer les mesures de rétablissement. Des travaux liés ont été rapportés dans [155] sur les méthodes d'isolement des défauts provenant d'un composant particulier et se propageant à travers toute la grille. Comme il est difficile de détecter les fautes, les défaillances fréquentes diminuent la confiance des utilisateurs dans une grille. Par conséquent, la poursuite des efforts dans le développement des détecteurs de fautes scalables et les méthodes d'isolement qui diagnostiquent correctement et avec précision les fautes difficiles à détecter sont essentiels.

5.4 Techniques de tolérance aux fautes dans les grilles de calcul

La tolérance aux fautes dans les grilles est un problème qui mobilise beaucoup de travaux de recherche. Certains pensent que la dégradation des performances dûe au déploiement d'un outil de tolérance aux fautes sur une grille de calcul n'est pas justifiable, et préfèrent utiliser les grilles sans ces outils. De cette manière, beaucoup de travaux sur la tolérance aux fautes se focalisent sur les grilles de données en utilisant des techniques de réplication [104]. Pour assurer la tolérance aux fautes dans les grilles de calcul, on utilise généralement les mêmes protocoles pour les clusters avec quelques modifications dans certains cas. Nous retrouvons dans cette catégorie la nouvelle version de FT-MPI *(Fault Tolerance Message Passing Interface)* [66], contenant des améliorations de mécanismes de détection et de gestion des erreurs et permettant d'avoir de meilleures performances sans être particulièrement adaptées

aux grilles de calcul. RPC-V [47] est un autre protocole de tolérance aux fautes pour les applications RPC *(Remote Procedure Call)*, qui est un modèle de programmation peu utilisé pour les applications de calcul numérique de haute-performance, contrairement à MPI *(Message Passing Interface)*. RPC-V *(RPC for Internet Connected Desktop Grid with Volatil Nodes)* est basé sur la journalisation des messages et la réplication passive. FT-Grid *(Fault tolerance within a grid environment)* [198] est un protocole de détection des erreurs de calcul accidentelles ou malintentionnées en envoyant la même requête d'un client à n groupes séparés de nœuds ; les résultats seront ensuite comparés entre eux au niveau du portail utilisateur. Cette technique n'est pas adaptée aux applications scientifiques car elle utilise un nombre élevé de nœuds. MPICH-GF [217] fournit la tolérance aux fautes pour les applications MPI *(version MPICH-G)* en utilisant un mécanisme de points de recouvrements coordonné. L'inconvénient de cette technique est le surcoût de calcul dû à la coordination entre les nœuds à chaque point de recouvrement et la relance de tous les nœuds pour recouvrir une faute. Plus récemment, des recherches qui s'intéressent à évaluer les coûts et les implications énergétiques liés aux mécanismes de tolérance aux fautes, y compris les checkpointing. A cette fin, Diouri et al. explorent l'impact de la consommation d'énergie des protocoles de checkpointing non coordonnés et coordonnés sur une charge de travail HPC MPI [49].
Quatre techniques pour la duplication ont une importance particulière :

1. Les points de reprise (*checkpointing*), ou l'enregistrement périodique de l'état d'un processus en cours d'exécution sur une ressource de calcul de sorte que, en cas de faute, il peut migrer vers une ressource opérationnelle ;
2. La réplication qui permet de maintenir un nombre suffisant de répliques d'un processus exécutés en parallèle avec le même état mais sur des ressources différentes, de sorte qu'au moins un réplica donne une garantie pour terminer le processus correctement ;
3. En cas de défaillance, l'ordonnancement consiste à trouver des ressources qui acceptent d'exécuter des tâches défaillantes ; et,
4. Des techniques de tolérance aux fautes basées sur le paradigme " Diviser pour régner" afin de contourner la scabilité des nœuds d'une grille.

5.4.1 Checkpoint et recouvrement

Comme dans d'autres systèmes, cette méthode est largement utilisée dans les réseaux [89, 96, 97]. Toutefois, si le hôte tombe en panne, le processus peut être transféré vers un environnement d'exécution différent. La migration d'un processus, qui est incapable de poursuivre sur son processeur d'origine, vers un nouveau processeur est connue sous le nom de basculement *(failover)*.

- **Points de reprise et de récupération dans les systèmes de grille** : La technique de checkpoint et les méthodes de migration de processus ont été largement utilisées dans des environnements de calcul haute performance [81,132]. De nombreuses grilles composées d'une fédération de clusters, exploitent le checkpoint et les techniques de migration pour un ensemble de processus parallèles qui sont fondées sur les méthodes utilisées dans le calcul à haute performance [40]. Ces systèmes offrent aussi le recouvrement pour les gestionnaires

de serveurs, qui gèrent en même temps l'exécution concurrente des processus. Par exemple, l'approche décrite dans [40] fournit une infrastructure de grille tolérante aux fautes pour les gestionnaires de serveurs et les nœuds des clusters. Les recherches sur les grilles pour les applications scientifiques utilisent aussi les techniques de point de reprise et de migration de processus basés sur le calcul haute performance. Les premiers efforts dans l'utilisation des points de reprise et de migration de processus dans les réseaux à grande échelle ont été signalés dans le middleware Légion [141] et Cactus [76, 115]. Condor [194] offre également la tolérance aux fautes des serveurs de site en répliquant les serveurs et en utilisant la migration de processus lorsqu'un serveur tombe en panne.

- **Méthodes de recherche en checkpointing** : Trois stratégies de checkpointing ont été considérées pour les processus concurrents par Elnozany et al. dans [225]. Des points de reprises coordonnés ont été implémentés dans MPI, en LAM/MPI [172]. Yeom et al [218] ont proposé une version tolérante aux fautes de MPI, MPICH-GF, pour les grilles qui utilisent les points de reprise coordonnés avec blocage. D'autres extensions de MPI qui utilisent les points de reprise coordonnés ont été proposées pour les environnements de grille par Bouteiller et al. [25] et en particulier dans [97], où la performance a été démontrée dans des conditions limitées de scalabilité. Récemment, Les auteurs de [175] présentent le *Fault Tolerant Messaging Interface* (FMI), pour une tolérance aux fautes rapide et transparente. FMI fournit des message-passing sémantique similaires à MPI, mais les applications écrites avec FMI peuvent fonctionner malgré l'occurrence d'une défaillance.
- **Migration de processus dans des environnements hétérogènes** : Une autre issue importante est le développement de techniques de recouvrement qui permettent la migration d'un processus dans des environnements d'exécution différents et avec différents domaines administratifs ou de sécurité. Des exemples de travaux préliminaires qui précèdent l'émergence de réseaux sont décrits [156, 223]. Pour les systèmes de grille, le problème a été étudié dans [203]. Fernandes et al. [53] ont proposé une méthode pour le transfert de processus concurrents d'une application parallèle dans un environnement de grille ayant des architectures de processeurs différents, ainsi que différentes implémentations de MPI. D'autres travaux de recherche relatifs aux points de reprise dans les environnements de grille comprennent l'élaboration de méthodes pour stocker les données des points de reprise dans des annuaires distribués [29, 166] ou pour sélectionner des intervalles de point de reprise optimaux [166].

5.4.2 Réplication des ressources d'une grille

Deux aspects de la recherche dans la réplication des ressources sont pris en considération : (1) le développement d'algorithmes pour déterminer les caractéristiques optimales (ou quasi-optimales) de placement de répliques de façon à accroître la tolérance aux fautes ; et, (2) des méthodes de synchronisation des répliques pour s'assurer de leur cohérence. Tous deux demeurent des questions de recherche, pour lesquelles des solutions proposées ont été mises en œuvre principalement dans des

expérimentations limitées ou dans des simulations.
Sélection des répliques et méthodes de placement : Une première tentative d'évaluer la tolérance aux fautes et les propriétés de scalabilité par des algorithmes adaptatifs pour les services de placement des répliques dans des systèmes distribués dynamiques a été présentée dans [4]. Weissman et Lee [216] ont proposé un système de gestion des répliques qui alloue dynamiquement les ressources répliquées sur la base des demandes des utilisateurs. Le système de recherche SRIRAM [209], pour la réplication automatique des ressources informatiques dans les grilles et d'autres environnements distribués, a été conçu pour améliorer la disponibilité des ressources et la tolérance aux fautes. D'autres systèmes de réplication ont été proposés. Valcarenghi [204] a proposé une approche de réplication de services où les répliques sont situées à proximité les unes des autres pour former des îles dans le réseau d'une grille. Un approche de réplication et d'ordonnancement distribuée et évolutive, appelée DistReSS, a été présentée dans [32]. Dans le cadre du projet e-Demand, [197], il a été proposé une méthode de réplication qui détecte les fautes de calcul dans un workflow de grille composé de plusieurs tâches. Genaud et Rattanapoka [69] ont développé un mécanisme pour MPI qui utilise une réplication de ressources visant à accroître la tolérance aux fautes des calculs parallèles. D'autres méthodes pour la réplication des calculs sur les ressources ont été proposées dans [1, 27] qui ont été vérifiées expérimentalement dans des conditions de passage à l'échelle. Bougeret et al. [24] étudient une approche simple où une application toute entière est répliquée. Ils fournissent une étude théorique d'un schéma d'exécution avec réplication lorsque la distribution des fautes suit une loi exponentielle. Ils proposent des algorithmes de détermination des dates de sauvegarde quand la distribution des fautes suit une loi quelconque.
Synchronisation des répliques : Peu de travaux semblent avoir été faits en ce qui concerne la proposition de méthodes efficaces et scalables pour la synchronisation des répliques. Une méthode, basée sur le placement de répliques sélectif, a été proposée dans [204]. Zhang et al. ont étudié, dans [225], cette question pour les répliques de services qui présentent un comportement non-déterministe et utilisent la messagerie asynchrone. Dans [113], les chercheurs ont proposé une version optimisée de l'algorithme Paxos [113] pour la synchronisation des répliques dans des environnements distribués et ils ont démontré l'efficacité de leur approche dans des conditions locales et à large échelle. Lors des travaux antérieurs [226], une approche plus traditionnelle de copie primaire a été utilisée pour étudier la réplication des services de grille et qui a été implémentée en utilisant OGSI [200] et Globus Toolkit [59]. Dans [226], il a été constaté que cette stratégie pourrait être facilement implémentée et a abouti à une plus grande disponibilité de services dans des environnements locaux. Toutefois, les coûts imposés par la notification OGSI afin de synchroniser les répliques des services qui se comportent non déterministe étaient significatifs.

5.4.3 Tolérance aux fautes par ordonnancement

Plusieurs études ont été faites sur l'ordonnancement tolérant aux fautes [95], comme, Charlotte [6] introduit un mécanisme de tolérance aux fautes appelé ordonnanceur *(eager)* pour l'équilibrage de charge et le masquage des défaillances. Les défaillances sont manipulées sans qu'il soit nécessaire de les détecter. En affectant

une tâche à plusieurs processeurs, ceci garantit également qu'une machine défaillante ou une machine lente sera remplacée automatiquement par une machine plus rapide. Bayanihan [173] utilise un mécanisme de tolérance aux fautes basé sur la crédibilité d'ordonnancement *(eager)*. Ce mécanisme fournit une tolérance aux fautes de processeurs bénévoles malveillants. Javelin [142] utilise un ordonnancement avancé qui améliore la scalabilité. Javelin construit une arborescence pour suivre l'état d'un calcul. La sélection de l'hôte pour réaffecter le job utilise un algorithme basé sur une arborescence. Si un hôte tombe en panne, Javelin fournit une tolérance aux fautes à l'aide d'un schéma de réparation de l'arborescence. Dans Gucha [118], il a été proposé un algorithme d'ordonnancement basé sur le transfert de l'information et des politiques de placement. Gucha implémente un algorithme d'ordonnancement où les hôtes, ayant de meilleures capacités, sont sélectionnés pour l'exécution des tâches. Bien que ces mécanismes d'ordonnancement tolèrent les fautes crashes et les fautes de déconnexion en utilisant un ordonnancement, ils ont quelques inconvénients comme des re-calculs redondants et ils ne considèrent pas l'autonomie des défaillances volontaires. Lorsque les mécanismes d'ordonnancement sont appliqués sur des défaillances volontaires autonomes, ils aboutissent à un problème live-lock indépendant [38]. Pour surmonter les problèmes évoqués, les mécanismes d'ordonnancement suivants ont été proposés. *Distributed Fault Tolerant Scheduling (DFTS)* [1] et *Volunteer Availability based Fault Tolerant Scheduling (VAFTS)* [38]. Dans Khanli et al. [102], les informations sur l'historique d'occurrence des fautes des ressources (*Resource Fault Occurrence History* RFOH) sont utilisées par des algorithmes génétiques. La tolérance aux fautes est basée sur un ordonnancement des tâches du noeud défaillant vers la ressource la plus fiable, fournie par l'algorithme génétique. Malarvizhi et al. [140] réplique les points de reprise pour un mécanisme de tolérance aux fautes. La réplication est basée sur un algorithme d'ordonnancement des tâches, qui calcule le minimum du temps global d'accomplissement d'un job (*Minimum Total Time to Release* MTTR). Le temps d'accomplissement d'un job comprend le temps de transmission des données et du job, le temps d'attente et le temps de son exécution sur une ressource. Le MTTR sélectionne la ressource en se basant, aussi, sur les exigences du job et les caractéristiques des ressources demandées.

5.4.4 Tolérance aux fautes pour les applications de type "Diviser pour régner"

"Diviser pour régner" est un paradigme populaire et efficace pour développer des applications parallèles de la grille [206]. Le paradigme "Diviser pour régner" est une généralisation du paradigme maître/esclave recommandé par le Global Grid Forum, comme un paradigme efficace pour le développement des applications de grille [120]. Les applications "Diviser pour régner" fonctionnent en divisant un problème de manière récursive en sous-problèmes. La subdivision récursive continue jusqu'à ce que les sous-problèmes deviennent triviaux à résoudre. Après la résolution des sous-problèmes, les résultats sont combinés de façon récursive jusqu'à ce que la solution finale est assemblée. Cela conduit à un arbre d'exécution de tâches imbriquées. Différents mécanismes de tolérance aux fautes pour des applications "Diviser pour régner" ont été développés [188]. **DIB *(Distributed Implementa-***

tion of Back tracking) [56] utilise le parallélisme "Diviser pour régner" et le vol de tâches. Dans DIB, lorsqu'un processeur n'arrive pas à exécuter un job, il émet une requête de vol, mais en attendant il commence à refaire le travail inachevé qui a été volé plus tôt. Cette approche est robuste parce que les défaillances peuvent être manipulées, même sans être détectées. "Diviser pour régner" se prête très bien pour la tolérance aux fautes, car tout job de l'arbre d'exécution d'une application peut toujours être recalculée en cas où son résultat a été perdu. Cependant, cette stratégie peut conduire à des calculs redondants très élevés. **Système MW *(Master.Worker)*** [77]. MW est un framework de programmation qui fournit une API pour implémenter les applications de grille " maître/esclave ". Dans le paradigme " maître/esclave ", les jobs à exécuter sont soumis à un processeur, appelé le maître, qui va les répartir à plusieurs processeurs, appelés esclaves. La tolérance aux fautes dans les systèmes "maître/esclave" peut être fournie par une réexécution des jobs perdus dans une défaillance. Etant donné que seul le processeur maître génère le job et recueille les résultats, il n'y a pas de problème de jobs orphelins, tant que le maître reste vivant. La défaillance du maître doit, par contre, être traitée séparément. Dans [219], les auteurs ont présenté un mécanisme qui permet la tolérance aux fautes, la malléabilité et de la migration pour les applications "Diviser pour régner". L'approche proposée utilise les résultats partiels par la restructuration de l'arbre d'exécution, ce qui permet de réduire énormément le calcul redondant.

6 Conclusion

Dans ce chapitre, nous avons fait un survol sur les différentes techniques de détection et de tolérance aux fautes qui datent depuis l'émergence des systèmes distribués. Les politiques de tolérance aux fautes comme la réplication et le recouvrement donnent des résultats très satisfaisants dans les systèmes distribués ordinaires. Le passage à l'échelle de ces systèmes, qui s'est accentué par l'apparition des clusters et plus tard par les grilles de calcul, a créé des contraintes nouvelles, comme la dynamicité et l'hétérogénéité. Ceci a conduit les chercheurs à faire des choix stratégiques, entre la mise à jour des anciennes techniques de tolérance aux fautes et à les adapter aux nouveaux systèmes ou le développement de nouvelles techniques répondant convenablement aux nouvelles contraintes.

Chapitre 3

Modèles hiérarchiques de tolérance aux fautes

1 Introduction

La présence de fautes et les défaillances qui peuvent en découler sont l'une des conséquences de la dynamicité des grilles. Les principales fautes sont de type fautes franches et concernent essentiellement les nœuds d'une grille [84]. Il existe de nombreux travaux de recherche, aussi bien sur la détection que sur la tolérance de ce type de fautes [198]. La plupart de ces travaux ont proposé des approches basées sur des techniques de réplication et sur des mécanismes de points de reprise [104]. A l'exception de l'approche décrite dans [128], ces travaux n'ont pas pris en compte la topologie réseau des grilles de calcul. Aussi, de nombreuses solutions proposées reposent sur des synchronisations entre groupes de nœuds. Dès lors que l'on souhaite adapter ce type de solutions aux grilles de calcul, il est nécessaire de prendre en compte la topologie du réseau utilisé pour définir une stratégie de tolérance aux fautes qui puisse donner de bons résultats. Dans ce chapitre, nous proposons deux modèles hiérarchiques de tolérance aux fautes dans les grilles de calcul. Ces modèles se caractérisent par : (i) l'utilisation d'une structure hiérarchique des nœuds ; (ii) l'existence d'un gestionnaire pour superviser chaque niveau de la hiérarchie ; (iii) une coopération entre les gestionnaires pour assurer la tolérance aux fautes au niveau de toute la grille..

2 Modèle hiérarchique dynamique

Une grille peut contenir plusieurs centaines de milliers de ressources hétérogènes et dynamiques, réparties sur une très large échelle. Parmi les modèles de structuration de ces ressources, nous trouvons les modèles hiérarchiques ou arborescents répondant favorablement aux architectures client/serveur, aux différents niveaux de

connexion réseau (LAN et WAN) et aux capacités des différentes ressources. La majorité des modèles proposés sont statiques, dans lesquels une grille est formée d'un ensemble de clusters reliés par une connexion WAN. Chaque cluster est constitué d'un ensemble de sites locaux gérant un ensemble de nœuds chargés d'exécuter les tâches des utilisateurs. Malgré que ces modèles aient donné des résultats encourageants, ils restent limités sur certains aspects :

1. Mauvaise gestion du volume de la grille : cette mauvaise gestion concerne aussi bien l'ajout que la suppression des nœuds d'une grille. Dans le cas de la réduction de ressources, il serait inutile de les structurer en un nombre de niveaux élevés ; dans le cas d'une augmentation, le nombre de niveaux peut devenir relativement grand.
2. Mauvaise gestion de la dynamicité : parmi les caractéristiques fondamentales des grilles de calcul, la dynamicité reste un élément essentiel. Or cette dynamicité ne peut être en aucun cas être traitée par un modèle hiérarchique statique.

Partant de ce constat, nous proposons un modèle hiérarchique dynamique, composé d'une racine, d'un nombre variable de niveaux intermédiaires et d'un niveau feuille (terminal) contenant les nœuds chargés d'exécuter les jobs. La dynamicité du modèle proposé est liée au nombre de ressources disponibles dans la grille et de leurs distributions dans l'arborescence [163].

2.1 Modèle proposé

Le modèle proposé structure une grille de calcul, du point de vue topologique, en un arbre d'interconnexion virtuel dynamique composé de (N) niveaux définis comme suit :

Niveau feuille : Chaque nœud de ce niveau représente une feuille ayant pour fonctions :

1. Exécution des jobs.
2. Envoi de l'état de ses jobs au niveau supérieur (père).

Niveaux intermédiaires qui ont pour fonctions :

1. Détection et tolérance des fautes de leurs fils.
2. Mise à jour des informations sur l'état des jobs au niveau des fils.
3. Envoi de l'état de leurs fils au père.

Niveau racine : Ce niveau est constitué d'un nœud associé à toute la grille, et que nous avons appelé gestionnaire de grille. Il a pour rôle la :

1. Détection et tolérance des fautes de ses fils.
2. Mise à jour de l'état des jobs au niveau des fils.
3. Envoi de l'état de la grille à la racine dupliqué.

La Figure 3.1 illustre le modèle proposé, dans lequel nous utilisons les notations suivantes :

R : Nœud Racine.

RD : Nœud Racine Dupliqué.

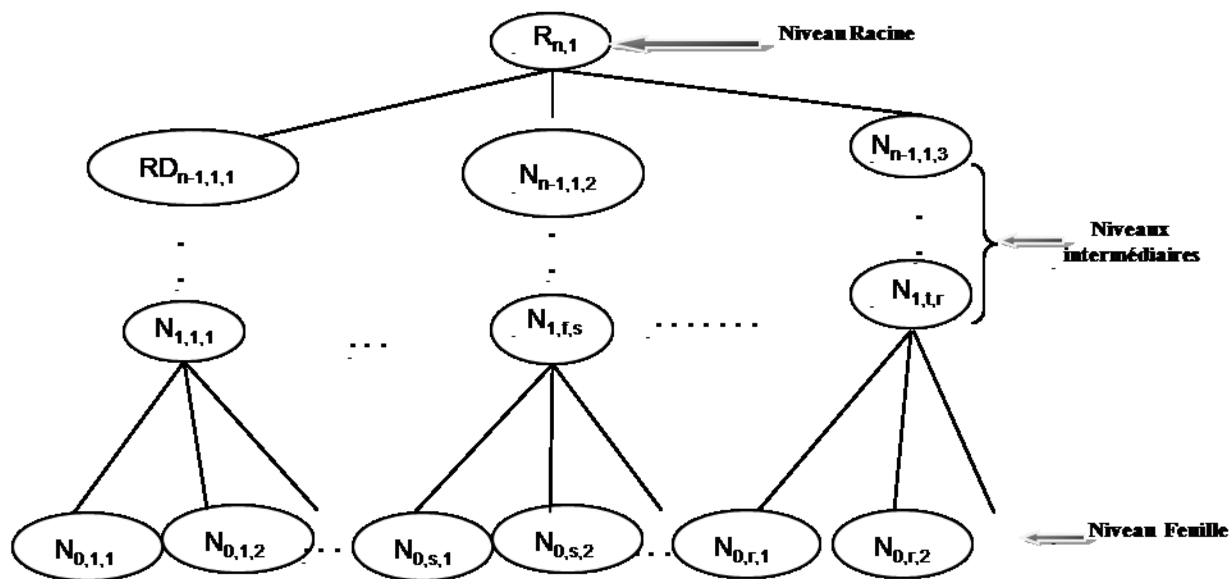

FIGURE 3.1 – Modèle hiérarchique dynamique de grille

N : Nœud.
$N_{i,j,k}$: représente un nœud où les indices (i, j, k) indiquent son niveau (i), le numéro de son père (j) et son numéro (k).
Dans le but de construire une arborescence initiale équilibrée, en fonction du nombre des fils et du nombre des niveaux, nous avons défini la fonction $A(N, F)$ qui retourne le nombre de nœuds nécessaires pour construire l'arborescence demandée.

$$A(N, F) = \frac{F^{N+1} - 1}{F - 1} \tag{3.1}$$

où :
F : nombre de fils pour chaque nœud ($F > 1$).
N : nombre de niveaux.
Exemple : Pour créer une arborescence de 3 niveaux où chacun possède 2 fils, le nombre de nœuds nécessaires est calculé comme suit :
$A(3, 2) = 15$ nœuds.

2.2 Notion de famille

Dans notre modèle, nous définissons la notion de famille d'un nœud comme étant un triplet <père, fils, frères>, comme illustré dans la Figure 3.2.
Définitions :
- **Père** : le nœud $N_{i+1,j,k}$ est le père du nœud $N_{i,k,L}$ si $N_{i+1,j,k}$ appartient au niveau $i + 1$ supérieur du nœud $N_{i,k,L}$ et qu'il y ait un lien entre eux.
- **Frère** : le nœud $N_{i,k,p}$ est le frère du nœud $N_{i,k,L}$ s'ils appartiennent au même niveau (k) et qui ont le même père $N_{i+1,j,k}$.

– **Fils** : le nœud $N_{i-1,L,q}$ est le fils du nœud $N_{i,k,L}$ si $N_{i-1,L,q}$ appartient au niveau inférieur $i-1$ du nœud $N_{i,k,L}$ et qu'il y ait un lien entre eux.

Les membres d'une famille peuvent changer de statut (père, frère, fils) dans le temps suivant leur position dans l'arborescence à un instant t.
Un nœud peut avoir plusieurs fils, plusieurs frères mais un seul père.
De manière générale, une feuille possède des frères et un père, un nœud des niveaux intermédiaires possède tous les membres d'une famille (père, fils, frères) mais la racine ne possède que des fils.

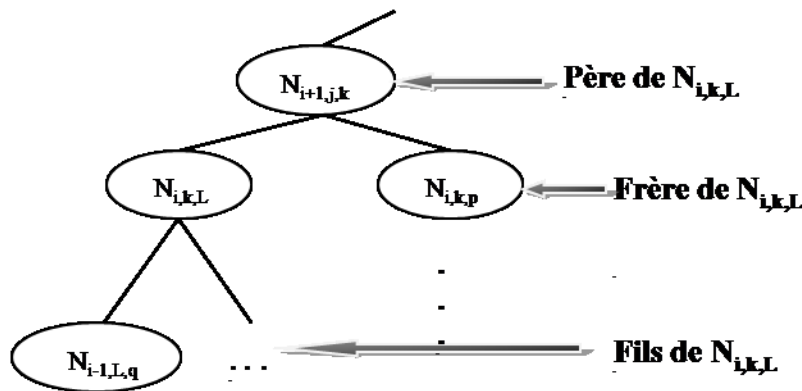

FIGURE 3.2 – Famille du nœud $N_{i,k,L}$

2.3 Types de fautes traitées

Nous considérons, dans notre modèle, les fautes franches et les fautes de déconnexion.

1. **Fautes franches :** Le grand nombre de nœuds présents dans une grille de calcul implique une éventualité non négligeable de pannes (fautes franches) de ces nœuds.
   - **a Niveau feuille :** quand une feuille est en faute franche, les traitements (exécution des jobs) sont stoppés. Il ne sera plus possible de soumettre d'autres jobs à cette feuille et elle n'envoie et ne reçoit aucun message.
   - **b Niveaux intermédiaires :** quand un nœud d'un niveau intermédiaire est en faute franche, la gestion de la tolérance aux fautes au niveau du nœud sera stoppée. Il n'envoie plus aucun message au père (message de son état) et il ne reçoit plus aucun message à partir de ses fils (messages relatifs à leurs états).
   - **c Niveau racine :** quand la racine est en faute franche, l'arborescence n'existe plus et tous les jobs seront perdus.
2. **Fautes de déconnexion :** Ces fautes correspondent à des déconnexions physiques (rupture d'un cable, équipement réseau hors service) ou logiques (panne du serveur DNS par exemple).

2.4 Détection de fautes

La détection de fautes consiste à identifier des situations de fonctionnement anormales pour mettre en place des procédures de tolérance appropriées. Pour cela, nous utilisons comme détecteur de fautes le message de vie au niveau des nœuds intermédiaires et la racine. La détection de ce type de fautes est à la charge du père. Chaque père envoie périodiquement un message de vie à tous ses fils. Si au bout d'un temps prédéfini, le père n'a pas reçu de message d'un de ses fils, il considérera que ce dernier est défaillant. S'il reçoit une réponse d'un de ses fils à travers un autre chemin (différent de celui qui les relie normalement), il considérera que le fils en question est en faute de déconnexion.

2.5 Tolérance aux fautes

Dans une grille, les jobs peuvent être lancés dans un seul nœud, comme ils peuvent être répliqués sur un certain nombre de nœuds. Nous proposons, dans ce modèle, deux techniques de tolérance aux fautes basées sur la distribution et le remplacement. Quand un père détecte une faute franche d'un de ses fils, il compte le nombre de frères de ce dernier :
si ce nombre est égal à 1, il utilisera comme technique de tolérance la méthode de remplacement, alors que si ce nombre est supérieur à 1 , il utilisera la méthode de distribution.

2.5.1 Méthode de distribution

A la détection d'une faute franche d'un nœud, sa tolérance est à la charge de son père. Celle-ci consiste à distribuer les fils du nœud défaillant sur ses frères (distribution locale). Le père du nœud défaillant compte premièrement les fils de ce dernier, puis il les divise équitablement sur ses frères, par la méthode "DIV (nombre des fils, nombre des frères)". Le reste des nœuds "il le trouve par la méthode MOD(nombre des fils, nombre des frères)" est divisé sur les frères du nœud défaillant de façon équitable. Une fois le nœud défaillant est réparé, il devient une feuille dans l'arborescence. La distribution des nœuds se fait de gauche à droite.
La section 2.7 détaillera toutes les scénarios d'exécution.
Exemple : La Figure 3.3 présente un cas de tolérance par la méthode de distribution. Le un nœud $N_{i-1,k,1}$ tombe en panne, sa tolérance est à la charge de son père $N_{i,j,k}$.
Le nœud défaillant $N_{i-1,k,1}$ a 3 fils ($N_{i-2,1,1}$, $N_{i-2,1,2}$ et $N_{i-2,1,3}$) et 2 frères ($N_{i-1,k,1}$ et $N_{i-1,k,2}$). Le père va distribuer les 3 fils sur les 2 frères comme suit :

1. Le père calcule les deux valeurs X et Y comme suit :
   X = DIV (3,2) =1
   Y = MOD (3,2)=1
   où :
   X : nombre de fils à distribuer aux frères de façon équitable.
   Y : nombre à affecter aux frères de gauche à droite.
2. Le père affecte à chaque frère X fils du nœud défaillant (le fils $N_{i-2,1,1}$ sera affecté au nœud $N_{i-1,k,2}$ et le fils $N_{i-2,1,2}$ ser affecté au nœud $N_{i-1,k,3}$).

3. Le père affecte les Y fils du nœud défaillant non encore affectés à ses frères de gauche à droite (le fils $N_{i-2,1,3}$ sera affecté au nœud le plus à gauche $N_{i-1,k,2}$).

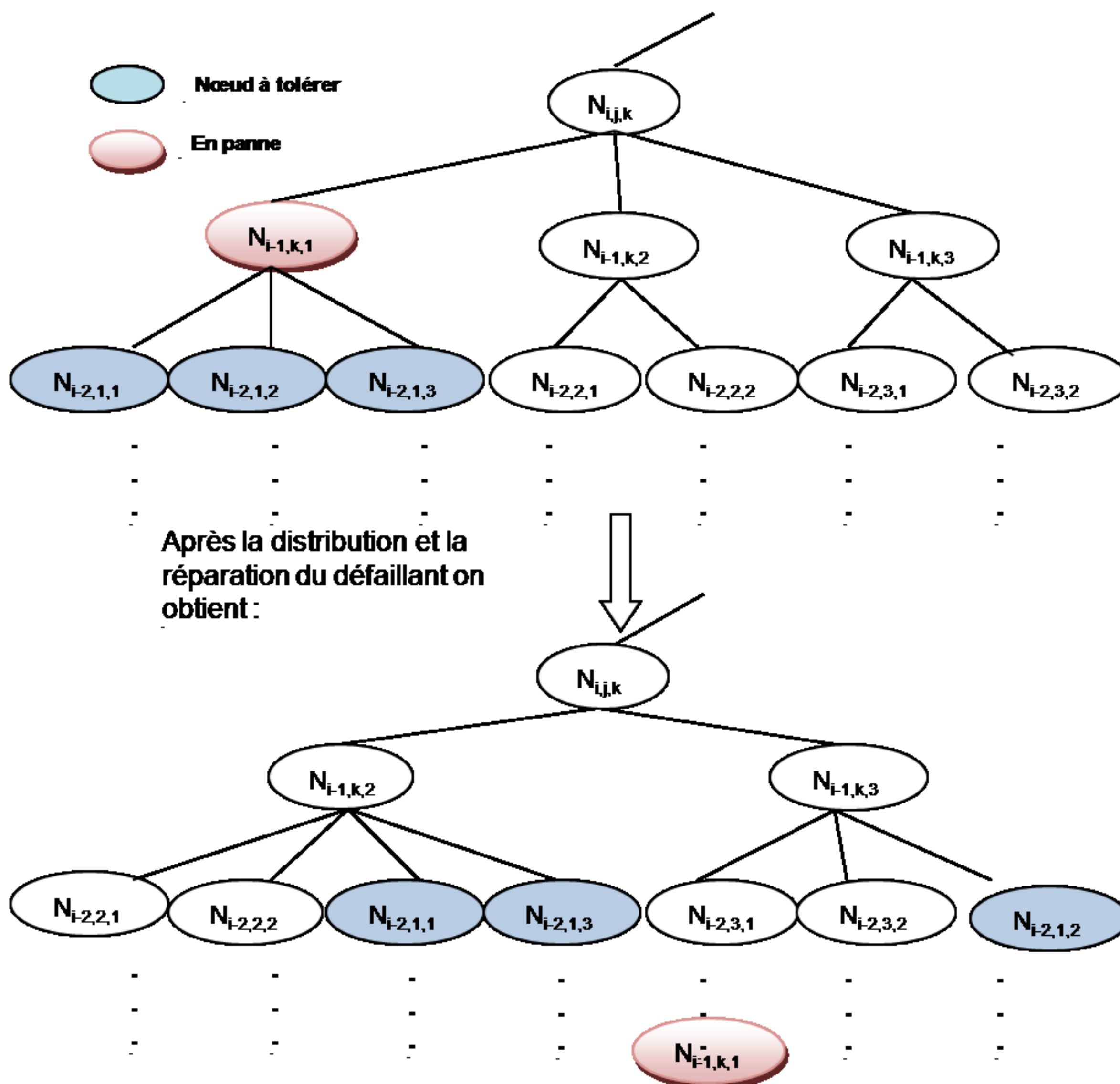

FIGURE 3.3 – Tolérance aux fautes en appliquant la méthode de distribution

2.5.2 Méthode de remplacement

Dans le cas où le nœud défaillant ne possède qu'un seul frère, nous aplliquons la méthode de remplacement. Cette méthode consiste à remplacer un nœud défaillant par une feuille dans l'arborescence. Pour cela, nous choisissons la feuille la moins chargée pour minimiser les jobs à tolérer et le nœud défaillant deviendra une feuille une fois qu'elle aura été réparée et réinsérée dans l'arborescence.
Deux cas sont envisagés :

a Remplacement d'un nœud appartenant à un niveau intermédiaire : Dans ce cas, quand un nœud détecte une faute franche d'un de ses fils et qui ne possède qu'un seul frère. Il cherche parmi les feuilles de l'arborescence, une feuille moins chargée pour remplacer le nœud défaillant. Les jobs de la feuille sélectionnée seront distribués sur les feuilles du même niveau. Dans le cas où ces feuilles sont chargées et il reste des jobs de la feuille sélectionnée, ils seront distribués sur les éléments du niveau supérieur (voir Figure 3.4).

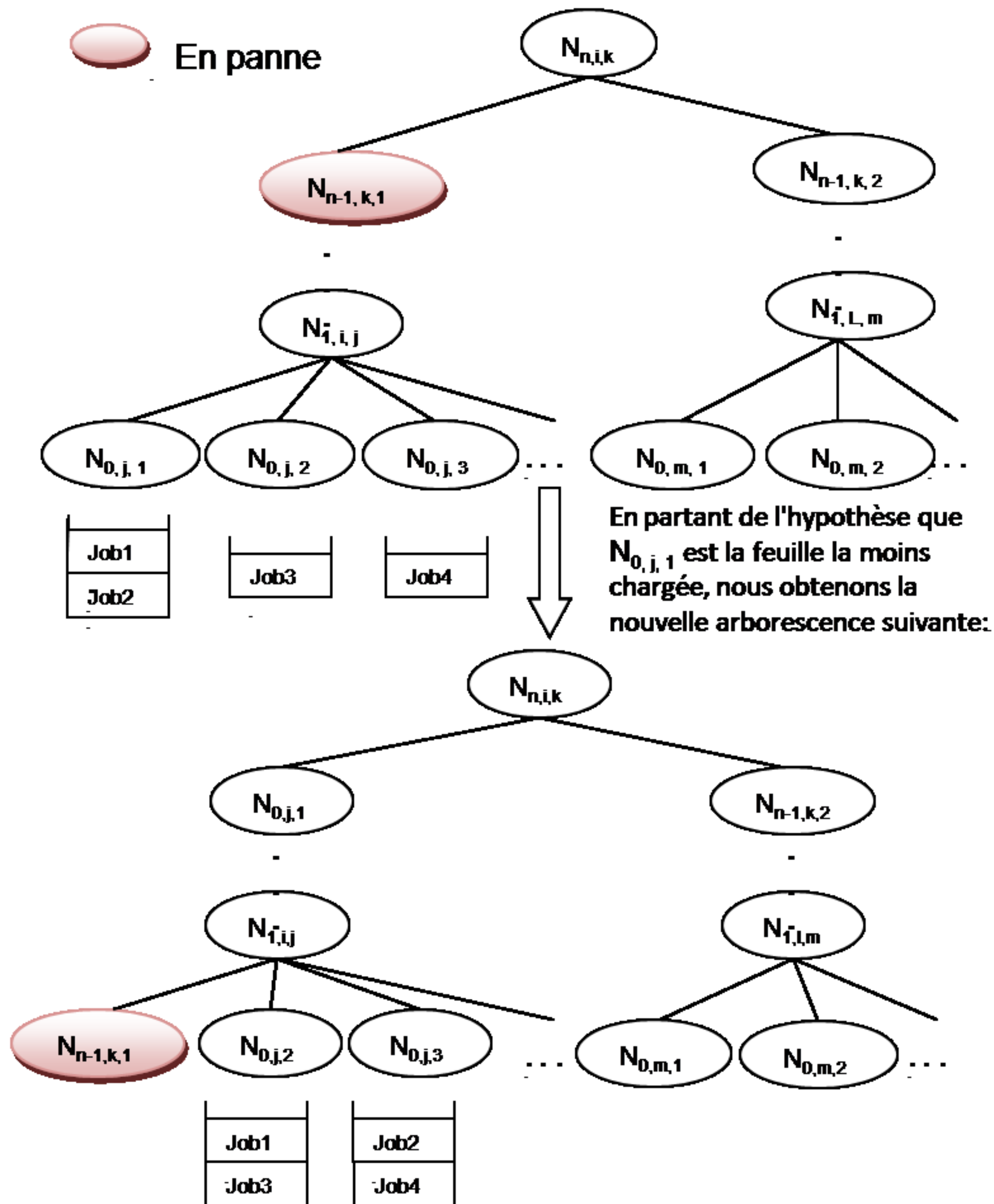

FIGURE 3.4 – Remplacement d'un nœud défaillant qui appartient à un niveau intermédiaire dans l'arborescence

b Remplacement de la racine par la racine dupliquée : La défaillance de la racine remet en cause tout le modèle, car l'arborescence devient inaccessible.

Pour faire face à cette situation, nous avons ajouté au modèle, une racine dupliquée qui fait partie des fils de la racine. Cette racine dupliquée est chargée de détecter les fautes de type crash (ou franche) de la racine et de se substituer à elle en cas de défaillance. Pour cela, la racine met à jour périodiquement la racine dupliquée.
Quand la racine dupliquée ne reçoit pas un message 'Ping' depuis la racine, elle vérifie si elle est déconnectée ou bien si la racine est tombée en panne franche. Si c'est le cas, nous remplaçons la racine par la racine dupliquée. Cette dernière sera elle même remplacée par une feuille de l'arborescence (la feuille la moins chargée) et la racine redeviendra une feuille une fois qu'elle aura été réparée et réinsérée dans l'arborescence.

2.6 Restructuration de l'arborescence

Suite aux changements possibles dans l'arborescence, dûs essentiellement à la tolérance de certaines défaillances par la distribution et l'insertion de nouvelles feuilles, l'arborescence deviendra déséquilibrée par rapport à son état initial, dans lequel les fils étaient équitablement distribués entre tous les niveaux. Pour éviter cette situation de déséquilibre, nous procédons à une restructuration de l'arborescence, qui consiste à restructer l'arborescence, de telle sorte que chaque niveau possède à peu près le même nombre de nœuds avec le même nombre de fils.
La restruturation touche les feuilles et les niveaux intermédiaires, comme suit :

1. **Feuilles :** Si le nombre des feuilles est acceptable mais elles sont mal distribuées sur leurs pères. La restructuration consiste à les redistribuer équitablement sur les nœuds du premier niveau (gestionnaire des feuilles). Mais quand le nombre des feuilles atteint une valeur maximale (cette valeur est définie en fonction du nombre de fils initial de l'arborescence). La restructuration consiste à créer des nouveaux gestionnaires des feuilles (premier niveau) et distribuer équitablement les feuilles sur leurs gestionnaires.
2. **Niveaux intermédiaires :** Suite à la restructuration des feuilles, les niveaux intermédiaires vont, aussi, subir un déséquilibre. Certains niveaux vont contenir un nombre de nœuds plus élevé par rapport aux autres niveaux. La restructuration des nœuds du niveau (i) consiste à sélectionner parmi les nœuds de ce niveau des nouveaux gestionnaires, ils seront mis au niveau supérieur $(i + 1)$ puis redistribuer les nœuds équitablement entre leurs gestionnaires.

2.7 Scénarios d'exécution

Nous allons présenter, dans ce qui suit, les différents scénarios d'exécution possibles pour chaque niveau. Nous commençons par le premier niveau N_1 qui représente le gestionnaire des feuilles, puis les niveaux intermédiares et enfin la racine. Nous présentons pour chaque niveau les structures de données utilisées pour assurer la tolérance aux fautes et les différentes techniques utilisées dans chaque niveau.

2.7.1 Niveau N_1 : gestionnaire des feuilles

1. **Structures de données :** nous utilisons pour ce scénario les structures de donnés suivantes :

   a La table db_fils composée des attributs : *id_n*$_0$, *ip_n*$_0$, *nb_job*, *size_list*, *size_list_free*.
   (*id_n*$_0$: identifiant de l'élément de traitement, *ip_n*$_0$: adresse IP de l'élément de traitement, *nb_job* : nombre de jobs soumis au nœud, *size_list* : nombre maximum de jobs que la file d'attente peut admettre, *size_list_free* : nombre de jobs que la file d'attente peut admettre).
   Clé primaire : *id_n*$_0$.

   b La table db_famille composée des attributs : *id_noeud*, *ip_noeud*, *type*.
   (*id_noeud* : identifiant d'un nœud, *ip_noeud* : adresse IP du nœud, *type* : type du nœud (père ou frère)).
   Clé primaire : *id_noeud*.

   c La table db_jobs composée des attributs : *id_n*$_0$, *id_job*, *job*, *job_state*, *duplique*, *tolere*.
   (*id_job* : identifiant de job, *job* : contient les paramètres suivants : (executable, argument, stdout, stderr), *job_state* : état du job qui peut être (Active (en exécution), Failed (en panne), Pending (en attente) ou Done (exécution terminée)), *duplique* : le job peut être soumis en réplication Passive ou Active, *tolere* : indique si la tolérance aux fautes d'un job est achevée ou non).
   Clé primaire : *id_n*$_0$ + *id_job*.

2. **Tolérance aux fautes locales :** Lorsque le gestionnaire de tolérance aux fautes du niveau N_1 détecte une faute d'un de ses fils ayant des jobs à exécuter (avec état de duplication égale à : "passive") et qu'il trouve suffisamment de fils potentiels qui peuvent recevoir d'autres jobs, il soumet ces jobs aux fils potentiels avec les mêmes identifiants et il envoie son état au niveau supérieur (son gestionnaire).

3. **Tolérance aux fautes vers le niveau supérieur :** Lorsque le gestionnaire de tolérance aux fautes au niveau N_1 détecte une faute d'un fils ayant des jobs à exécuter et qu'il ne trouve pas suffisamment de fils locaux pour recevoir ces jobs, il soumet une partie des jobs aux fils locaux avec les mêmes identifiants. Ensuite, il soumet le reste des jobs au niveau supérieur (le gestionnaire) avec les mêmes identifiants et il envoie son état au niveau supérieur (son gestionnaire).

2.7.2 Niveau intermédiare N_i

1. **Structures de données** utilisées au niveau $(N_i, i > 1)$ sont :

   a Les table (db_fils, db_jobs) qui représentent une agrégation des données stockées dans les tables du niveau inférieur (N_{i-1}) en ajoutant un attribut indiquant l'identifiant de chaque nœud (de niveau N_{i-1}).

   b La table db_famille qui est identique à la table utilisée pour le niveau N_1.

c La table db_fils composée des attributs : $id_n_{i-1}, id_n_{i-2}, ..., id_n_1, id_n_0$, ip_n_0, nb_job, $size_list$, $size_list_free$.
$id_n_{i-1}, id_n_{i-2}, ..., id_n_1$: identifiants des nœuds de niveaux (n_{i-1}, $n_{i-2}, ..., n_1$).

d La table db_jobs qui est composée des attributs : $id_n_{i-1}, id_n_{i-2}, ..., id_n_1$, id_n_0, ip_n_0, id_job, job, job_state.

2. **Tolérance aux fautes :** Lorsque le gestionnaire de tolérance aux fautes du niveau N_i détecte une faute d'un de ses fils, il compte premièrement le nombre de frères du nœud (fils) défaillant. Si ce nombre est supérieur à 1, il utilisera la méthode de distribution pour tolérer la faute, sinon il utilisera la méthode de remplacement et il envoie son état au niveau supérieur (son gestionnaire).

2.7.3 Niveau Racine

Les structures de données utilisées dans la racine sont les mêmes que celles utilisées dans le niveau N_i à l'exception de la table $db_famille$. Il est en de même pour la technique de tolérance, sauf que pour cette dernière il n'y a pas un niveau supérieur au dessus de la racine. Comme conséquence, les résultats ne seront pas envoyés vers un niveau supérieur.

2.8 Discussion

Dans ce chapitre, nous avons proposé un modèle de tolérance aux fautes adapté aux grilles de calcul qui tient en compte surtout de la dynamicité des ressources et le passage à l'échelle. Ce modèle est totalement indépendant de toute architecture physique. Partant d'une structure arborescente, le modèle permet de transformer une grille de calcul en un arbre virtuel. Avec ce modèle, nous avons proposé deux techniques de tolérance aux fautes basés sur la distribution et le remplacement. Ces techniques diffèrent selon les niveaux de l'arborescence, et prennent en considération les fautes franches et de deconnexion. Ils reposent sur le principe de recouvrement d'erreurs, en utlilisant une réplication active et passive des jobs soumis à une grille.

3 Modèle multi-arborescent pour une grille mobile tolérante aux fautes

Durant la dernière décennie, le monde de l'environnement mobile a été témoin d'une grande évolution [86]. Les progrès technologiques, surtout dans le domaine des communications, ont vu l'émergence d'un nouveau type de communication, à savoir la communication sans fil, grâce à laquelle nous sommes passés des réseaux filaires aux réseaux sans fil interconnectant des machines mobiles. Plusieurs dispositifs sans fil, tels que les ordinateurs portables, les téléphones cellulaires et les assistants personnels numériques, inondent le marché et font partie de notre vie quotidienne.

D'autre part, une grille de calcul a pour objectif de mutualiser des ressources, souvent réparties géographiquement sur plusieurs sites. Ainsi, tout utilisateur dispose d'une puissance de calcul et d'espace de stockage dont il a besoin pour lancer des applications, sans se préoccuper de savoir sur quelles machines elles seront déployées et exécutées.
La combinaison de l'environnement mobile avec les grilles de calcul a permis d'ouvoir un nouveau champ de recherche où les dispositifs mobiles peuvent être efficacement incorporés à une grille comme fournisseurs ou consommateurs de services, en construisant une architecture appelée grille mobile. Une telle intégration soulève beaucoup de défis (connectivité intermittente, hétérogénéité des dispositifs, sécurité, etc.) qui doivent être relevés.
Un environnement de calcul mobile est caractérisé par une disponibilité limitée de ressources, une mobilité élevée, une faiblesse au niveau de la largeur de la bande passante et une déconnexion fréquente qui provoquent la défaillance des applications, ce qui nécessite de fournir un service de tolérance aux fautes pour de tels environnements.
Dans ce contexte, nous nous sommes intéressés au développement d'un modèle multi-arborescent pour une grille mobile, composé de trois niveaux hiérarchiques : (i) ses racines, représentant les nœuds d'une grille de calcul, qui jouent le rôle de gestionnaires de groupes mobiles ; (ii) les interlocuteurs, servant d'intermédiaires entre les gestionnaires et les dispositifs mobiles qui se trouvent au niveau le plus bas ; et, (iii) les subordonnées sont chargés d'exécuter les jobs. Notre modèle prend en compte les contraintes liées à la mobilité de ces dispositifs, notamment la forte occurrence de fautes. Pour cela, nous avons soutenu notre modèle par des mécanismes de tolérance aux fautes des dispositifs mobiles [160].

3.1 Rôle des dispositifs mobiles dans une grille mobile

Les capacités des dispositifs mobiles augmentent d'un jour à l'autre en matière de puissance de calcul et d'espace de stockage. Ces dispositifs peuvent jouer le rôle de consommateur ou de fournisseur de ressources.

3.1.1 Dispositifs mobiles comme consommateurs de ressources

Dans ce cas de figure, les dispositifs mobiles sont considérés comme ayant des capacités de calcul et/ou de mémoire limitées [131]. Cette hypothèse (qui est en fait une réalité) est tout à fait vraie pour les dispositifs mobiles en général, et c'est la raison principale pour laquelle les défenseurs de cette approche proposent de les intégrer dans une grille. La grille peut fournir les ressources manquantes dans le dispositif mobile sur demande. Néanmoins, des problèmes liés à la nature de ces dispositifs surgissent comme les déconnexions fréquentes et une durée de vie limitée de batterie rendent difficile à la mise en œuvre de procédures de tolérance aux fautes directement sur un système de grille. La soumission des jobs et la réception des résultats n'est pas aussi franche qu'elle semble, à cause des contraintes répandues dans les communications sans fil. Dans [171], les auteurs proposent l'utilisation des "proxies" qui actent comme passages à la grille. Ces "proxies" prennent le rôle de médiateur

entre le dispositif mobile et le système de grille, et essayent de masquer l'instabilité de l'environnement sans fil/mobile. En agissant au nom du dispositif mobile, le médiateur est responsable de soumettre les jobs, de surveiller leur exécution et retourner les résultats aux clients. Dans d'autres approches [186], les dispositifs sont consacrés à l'accès sans avoir besoin de capacités de traitement et/ou de mémoire. Ainsi, le rôle d'une grille consiste à fournir toutes les fonctionnalités supplémentaires exigées par les utilisateurs mobiles.

3.1.2 Dispositifs mobiles comme fournisseurs de ressources

Deux architectures fondamentalement différentes ont été définies pour exploiter les ressources d'un dispositif mobile : les grilles mobiles sur site et les grilles mobiles ad-hoc.

1. **Grille mobile sur site (*Mobile Grids On-site*)** : Pour mettre en place une grille mobile sur site [149], les dispositifs mobiles résidant dans un secteur bien défini, tel qu'une cellule dans les réseaux cellulaires ou un WLAN hot-spot (*Service Area*, *SA*), sont coordonnées par une entité centrale (résidant dans le point d'accès : la station de base, "*Base Station*", *BS*) afin d'accomplir une tâche dans la grille de calcul. Dans ce type d'architectures, les dispositifs fournissent la description de leurs possibilités et leur degré de disponibilité au *BS*. En exploitant ces informations, la *BS* a la responsabilité de décomposer une demande entrante et diviser l'exécution globale d'un job en fournissant des tâches spécifiques à chacun des dispositifs mobiles participants. Une demande peut venir d'un ou d'autres dispositifs mobiles dans une *SA* ou d'un client différent en dehors de cette *SA* (fixe ou mobile). L'avantage de cette approche est que la *BS* peut agir en tant que médiateur capable de masquer, au nœud demandant un service, l'hétérogénéité des dispositifs participants, de coordonner l'exécution globale du job soumis et permettent même au système de grille d'apparaître, vis à vis du reste du réseau, comme un nœud ordinaire de grille [149]. Dans les architectures présentées dans [111], les auteurs n'adressent pas réellement le problème de la mobilité. Ils considèrent que, si un nœud mobile quitte sa *BS*, la tâche qu'il exécutait est avortée dans le sens qu'elle sera soumise une autre fois plus tard.

2. **Grilles mobiles ad-hoc (*Mobile ad-hoc grids*)** : Dans le cas des grilles mobiles ad-hoc [122], il n'y a aucune autorité centrale responsable de la coordination de l'exécution globale d'un job. D'autres problèmes surgissent à cause de la nature ad-hoc de tels systèmes. L'absence de la coordination centrale impose des difficultés en découverte de service, d'ordonnancement des jobs, etc. L'approche suivie pour surmonter ces difficultés, est de former une virtuelle dorsale Internet (*backbone*) composée d'un certain nombre de nœuds mobiles. Ces derniers sont responsables de coordonner l'ensemble des nœuds mobiles résidants dans un certain secteur du réseau ad-hoc. L'instabilité de la topologie du réseau induit plus de difficultés dues aux caractéristiques ad-hoc des réseaux.

3.2 Modèle de fautes dans un environnement mobile

En environnement mobile, le dispositif mobile est sujet à des déconnexions, comme par exemple, lors du passage de l'utilisateur dans une zone d'ombre radio. En outre, l'utilisateur exécute des applications réparties dont la défaillance d'un des processus entraîne la défaillance de toute une application. Ces deux propriétés extra fonctionnelles (gestion des déconnexions et tolérance aux fautes) reposent chacune sur des mécanismes de détection (de connectivité, de déconnexion et de défaillance). Dans le cadre de nos travaux, nous considérons deux types de déconnexions : les déconnexions volontaires et les déconnexions involontaires.

1. Les premières, décidées par l'utilisateur depuis son dispositif mobile, sont justifiées par les bénéfices attendus sur le coût financier des communications, l'énergie, la disponibilité du service applicatif, et la minimisation des désagréments induits par des déconnexions inopinées.
2. Les secondes sont le résultat de coupures intempestives des connexions physiques du réseau, par exemple, lors du passage de l'utilisateur dans une zone d'ombre radio.

3.3 Modes de fonctionnement des dispositifs mobiles

La Figure 3.5 schématise les différents modes de fonctionnement des dispositifs mobiles (connecté, partiellement connecté, déconnecté et en veille) ainsi que les différentes transitions possibles entre ces modes [152]. Dans le mode connecté, le dispositif mobile dispose d'une bonne connexion au réseau, comme une station fixe. Dans le mode partiellement connecté, le dispositif mobile ne dispose plus de charge suffisante pour communiquer avec le réseau, ce n'est qu'un lien faible à la suite, par exemple, de perturbations du réseau hertzien ou d'un faible niveau de la batterie. Dans le mode déconnecté, le dispositif mobile est isolé soit parce qu'il n'est plus physiquement relié au réseau, soit parce qu'il est impossible de maintenir une connexion sans fil. Enfin, dans le mode en veille utilisé par les dispositifs mobiles pour préserver les ressources énergétiques, la connexion réseau est maintenue, mais le terminal mobile ne s'en sert pas.

Les transitions entres les modes de fonctionnement correspondent aux changements de valeur de la disponibilité de la ressource considérée. Dans les environnements mobiles, la détection de la connectivité peut être assurée par un détecteur de connectivité [193]. Ce dernier donne des informations sur une ressource telle que la bande passante ou la charge de batterie pour déterminer localement le niveau de l'utilisation de l'interface réseau par le dispositif mobile. La prédiction du niveau de ressource du dispositif mobile joue un rôle clé dans l'établissement du mode de fonctionnement. Dans la problématique de gestion des défaillances des ressources de communication, la prédiction permet d'anticiper les besoins de chargement du cache en prévision de déconnexions. Par contre, dans une situation de très bonne connectivité, elle pourrait être utilisée pour décider le lancement d'exécutions distantes (*off-loading*) afin de préserver les ressources locales (processeur, batterie, mémoire, etc.) [152].

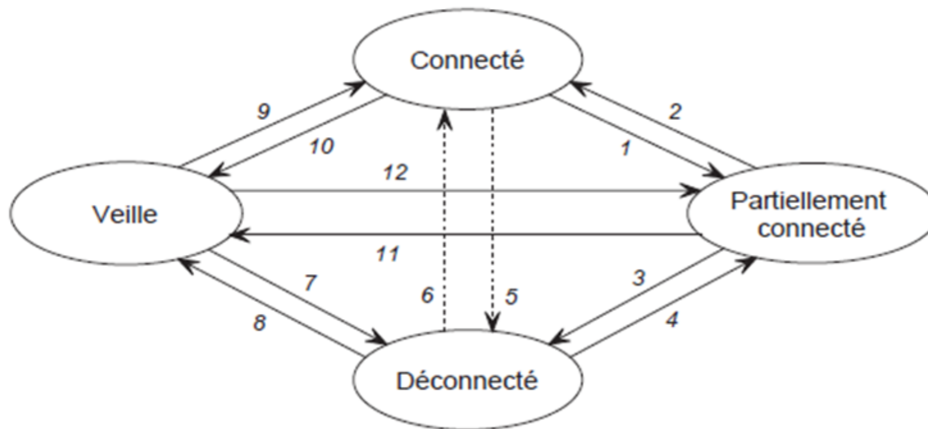

Figure 3.5 – Modes de fonctionnement des dispositifs mobiles [152]

3.4 Architecture proposée pour une grille mobile

Le modèle de grille, comme le montre la Figure 3.6, est composé d'une grille fixe et d'un ensemble de dispositifs mobiles détectés à travers des points d'accès reliés à la grille. Les nœuds de la grille fixe, ayant en général une puissance de calcul et une capacité de stockage importantes, sont reliés entre eux par une liaison filaire de haut débit. Les nœuds participants à la grille mobile disposent des points d'accès leur permettant de détecter les dispositifs mobiles dans son rayon de couverture. Un sous-ensemble des dispositifs mobiles détectés peuvent participer à une grille de calcul. Ces derniers doivent être capables de communiquer avec la grille par des interfaces de communication avancées, de pouvoir supporter le chargement d'un middleware dédié aux grilles de calcul, d'être capable d'exécuter des jobs et de transmettre les résultats à la grille et avoir une capacité de stockage acceptable en cas de nécessité de traitement sur des données particulières.
La construction d'une telle grille va permettre de bénéficier des capacités de calcul des dispositifs mobiles. Ces derniers se caractérisent par : (i) une limitation en terme de puissance de calcul et de stockage ; (ii) une mobilité qui peut être très aléatoire et forte ; (iii) un débit faible et très aléatoire ; (iv) une autonomie de batterie limitée. Dans ces conditions, la grille mobile sera sujette à une fréquence élevée de défaillances.
Dans ce qui suit, nous proposons un mécanisme de tolérance aux fautes pour une grille mobile, tel que les dispositifs mobiles seront structurés en un ensemble des groupes mobiles (GM) autour de chaque point d'accès (PA). Chaque groupe est géré par un interlocuteur (INT) chargé d'assurer une interface entre les éléments du groupe et le nœud de la grille fixe.

3.4.1 Fonctionnement du modèle

La Figure 3.7 présente les différents acteurs de l'architecture de la grille mobile proposée ci-dessus.

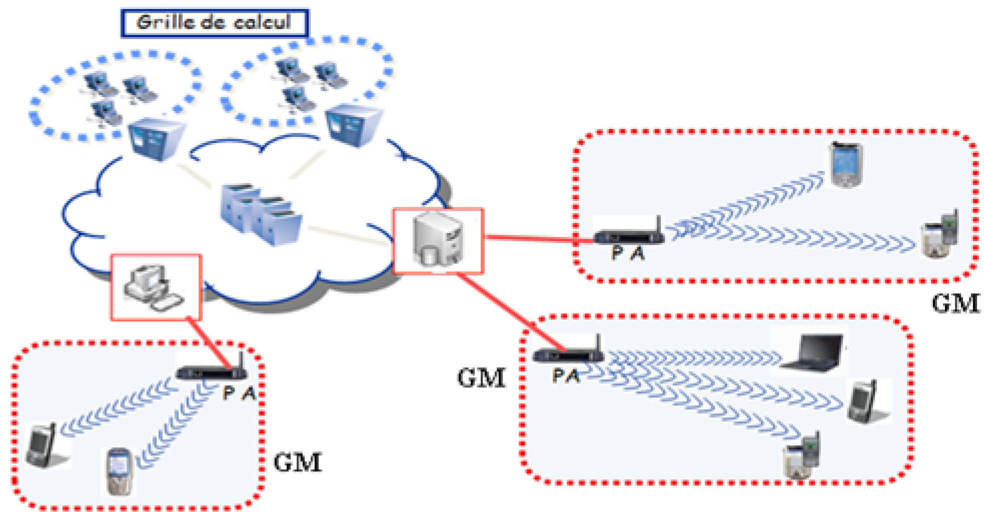

FIGURE 3.6 – Environnement d'une grille mobile

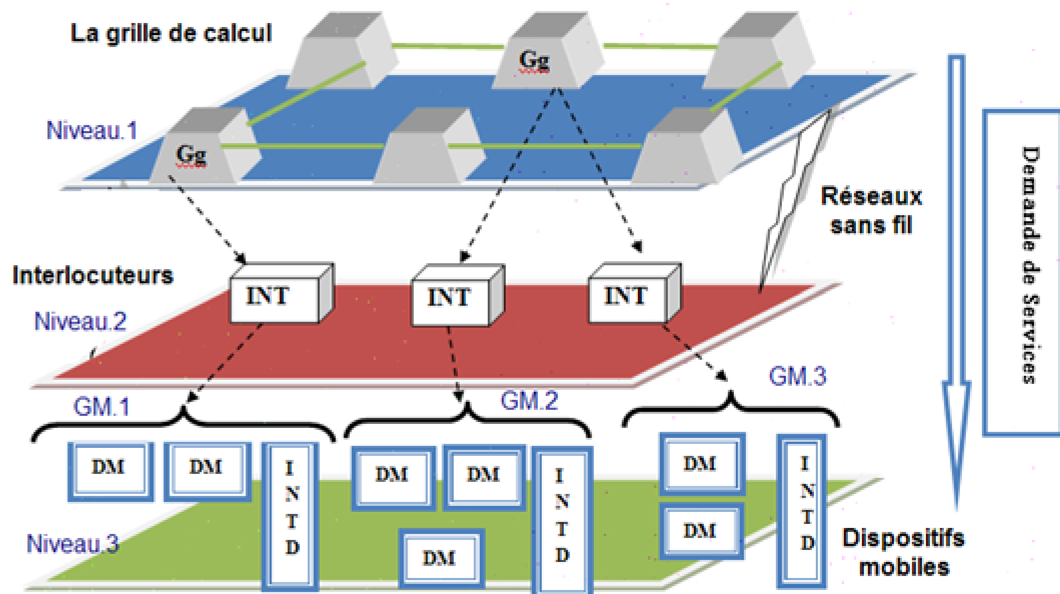

FIGURE 3.7 – Architecture d'une grille mobile

3.4.2 Acteurs du modèle de grille mobile

1. **Gestionnaire de groupe** (Gg) **:** c'est un nœud appartenant à la grille fixe, qui supervise un ou plusieurs groupes mobiles, en communiquant avec les dispositifs mobiles par le biais de leurs interlocuteurs. Dans le cas d'une défaillance de l'interlocuteur (INT), le Gg informe l'interlocuteur dupliqué ($INTD$).

2. **Interlocuteur (INT) :** c'est un dispositif mobile, chargé de gérer un groupe mobile. L'ensemble de ses dispositifs seront ses subordonnés.
3. **Interlocuteur dupliqué ($INTD$) :** parmi les subordonnés d'un interlocuteur, un sera choisi comme réplique. Il subit les mises à jour périodiques à partir de l'interlocuteur sur l'état du groupe, pour qu'il soit capable de le remplacer en cas de défaillance.
4. **Subordonné (DM) :** c'est un dispositif mobile appartenant à un groupe mobile, qui a pour rôle d'exécuter des jobs.

3.4.3 Niveaux du modèle

Dans le modèle proposé, nous nous intéressons qu'aux demandes de service venant de la grille de calcul vers les DM's à travers le Gg. Quand une demande de service ou des jobs arrive aux Gg's, ces derniers les répartissent sur leurs propres INT's. Les demandes arrivées à chaque INT seront réparties sur ses subordonnés (DM). Après que les demandes aient été réparties entre les DM's, ces derniers vont effectuer les traitements associés, et envoyer les résultats à leurs INT, qui à leurs tours les envoient vers le demandeur (Gg). L'INT partage toutes ses informations en permanence avec son Interlocuteur Dupliqué ($INTD$), en lui envoyant périodiquement des mises à jour et aussi à chaque fois où il y a une demande de service. L'$INTD$ est un dispositif mobile se trouvant dans chaque groupe mobile , qui a pour principale fonction de remplacer son INT en cas de défaillance.

3.5 Modèle hiérarchique associé à une grille mobile : le modèle $G/I/S$

Pour représenter une grille mobile, la solution proposée est de la transformer de manière univoque en un arbre virtuel d'interconnexions. Cet arbre donne un modèle de représentation générique, noté $G/I/S$, où G représente le nombre de gestionnaires de groupes, I le nombre d'interlocuteurs et S le nombre des subordonnés. La Figure 3.8 illustre ce modèle par plusieurs arbres, chacun ayant trois niveaux :

1. **Niveau 3 :** chaque nœud de ce niveau représente un dispositif mobile, noté DM, considéré comme élément de calcul et ayant pour fonctions :
   (a) Exécution des jobs.
   (b) Envoi l'état des jobs à son interlocuteur.
2. **Niveau 2 :** ce niveau est rattaché aux dispositifs mobiles spécifiques appelés interlocuteurs (notés INT), chacun gérant un groupe mobile (un ensemble de subordonnés). Choisi par l'administrateur d'une grille selon ses caractéristiques matérielles spécifiques (puissance de calcul, capacité de stockage, charge de batterie), il aura pour rôle de :
   (a) Gérer les jobs provenant de la grille de calcul.
   (b) Détecter de nouveaux mobiles dans sa zone de couverture.

(c) Transmettre les résultats au niveau supérieur.

3. **Niveau 1 :** ce niveau correspond à la racine de l'arbre ; il est constitué d'un nœud spécifique de la grille de calcul appelé gestionnaire de groupes (noté Gg). Il est chargé de superviser un ou plusieurs interlocuteurs, en communiquant avec eux dans sa zone de couverture. Pour cela, il doit être fixé au préalable.

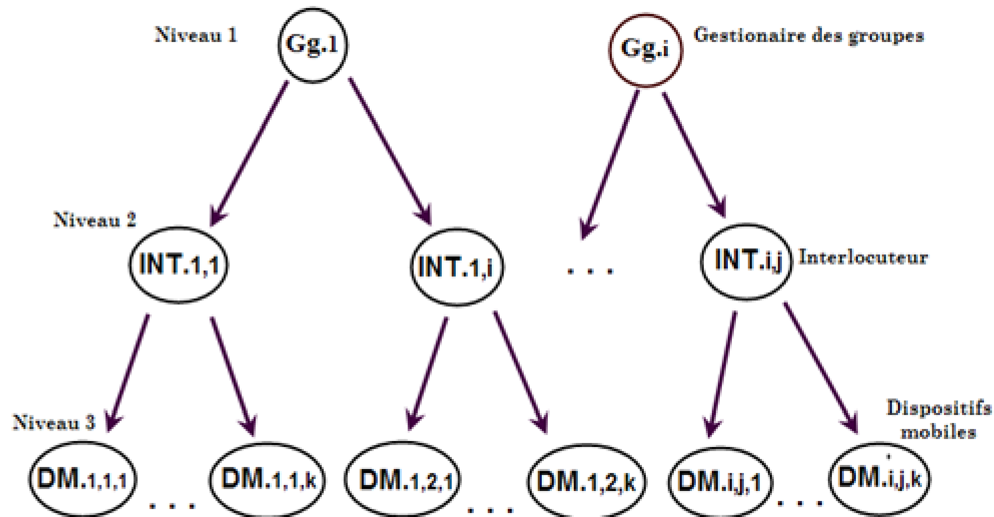

FIGURE 3.8 – Architecture hiérarchique du modèle d'une grille mobile

$Gg(i)$: gestionnaire de groupes mobiles de numéro i.
$INT(i, j)$: interlocuteur j associé au gestionnaire de groupes de numéro i.
$DM(i, j, k)$: dispositif mobile k associé à l'interlocuteur j qui est associé au gestionnaire de groupes i .

3.5.1 Détection d'un nouveau DM

Comme cela est illustré dans la Figure 3.9, l'$INT(1.1)$ détecte un nouveau dispositif mobile $DM(1, 1, 3)$ dans sa zone de couverture. Il doit donc l'associer à son groupe et mettre à jour son $Gg(1)$.

3.5.2 Déconnexion d'un DM

A tout moment, un dispositif mobile peut quitter son groupe (en se déconnectant). Une fois que son INT a détecté sa déconnexion, il met à jour son Gg.

A titre d'exemple dans la Figure 3.10, l'interlocuteur $INT(1, 2)$ détecte la déconnexion du dispositif $DM(1, 2, 2)$. Pour cela, il met à jour sa liste de subordonnés et envoie un message à son gestionnaire de groupe, qui va à son tour mettre à jour sa liste des dispositifs. Le $DM(1, 2, 2)$ sera détecté par l'$INT(2, 1)$ qui va le joindre à son groupe et il sera identifié par $DM(2, 1, 2)$; par contre le $DM(1, 2, 1)$ sera totalement déconnecté de la grille.

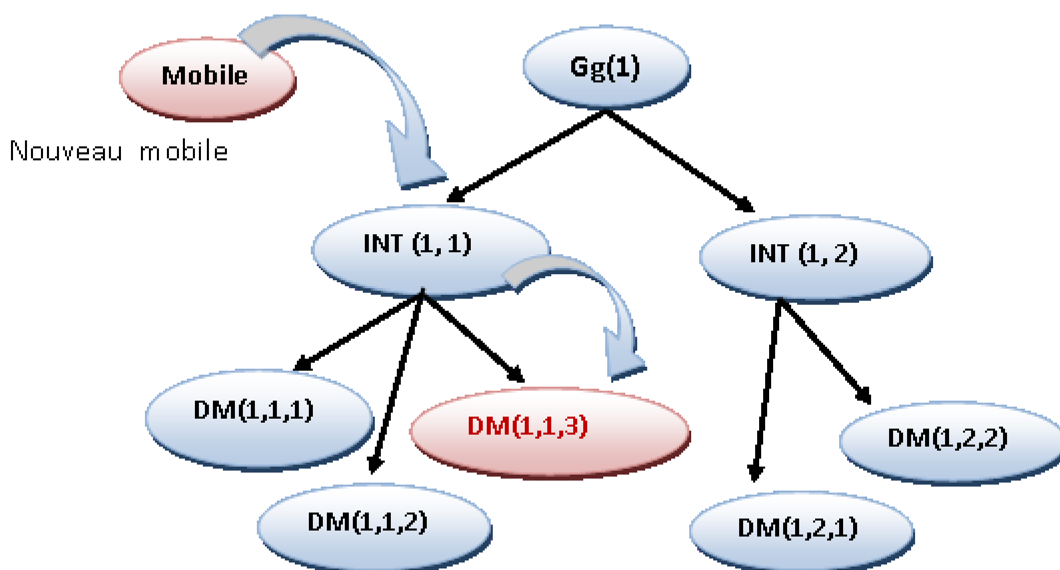

FIGURE 3.9 – Schéma décrivant la détection d'un nouveau dispositif

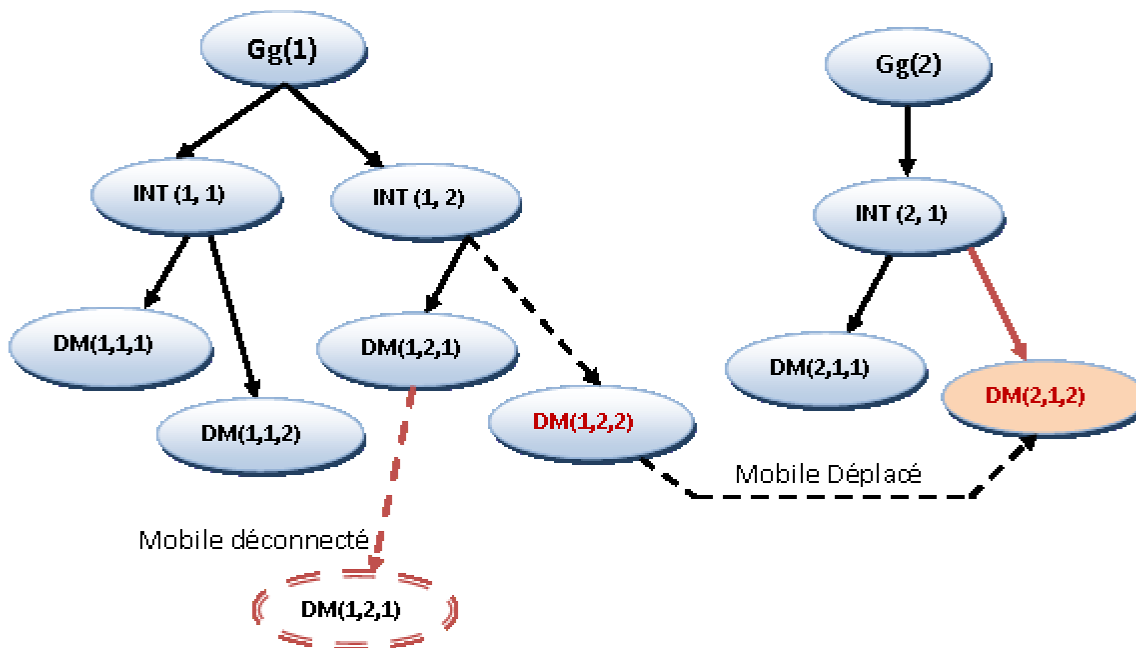

FIGURE 3.10 – Schéma de déconnexion et de déplacement d'un dispositif mobile

3.5.3 Déconnexion d'un INT

Lorsqu'un Gg détecte la déconnexion d'un INT, il envoie un message à l'$INTD$ afin de remplacer l'INT défaillant. Le nouveau INT prend en charge de terminer ce que l'ancien INT était en train d'exécuter, mais il doit d'abord élire un nouveau $INTD$ pour le remplacer en cas de défaillance.

3.6 Tolérance aux fautes

Afin d'assurer le bon fonctionnement d'une grille mobile, il faut tolérer les fautes qui interviennent à tout moment, que ce soit par les fautes de la partie stable (grille de calcul) ou par la défaillance des éléments mobiles (problèmes de déconnexion, d'énergie, etc.). Nous nous intéressons à une catégorie précise des fautes des dispositifs mobiles : les fautes de déconnexion, les fautes crash et la dégradation de qualité de service. Pour cela, un service de tolérance aux fautes est nécessaire dans une grille mobile.

3.6.1 Tolérance au niveau interlocuteur

1. Tolérance aux fautes locale : Lorsque le gestionnaire de tolérance aux fautes, au niveau d'interlocuteur détecte une faute d'un subordonné ayant des jobs en état "Actif" ou en "Attente", il cherche localement des subordonnés libres qui peuvent prendre en charge les jobs du subordonné défaillant (voir Figure 3.11). La technique de tolérance à suivre est décrite dans ce qui suit :

1. Soumettre les jobs du subordonné défaillant aux subordonnés libres avec les mêmes identifiants du subordonné qui est tombé en panne.
2. Suivre l'état des jobs par leurs identifiants.
3. Une fois les jobs terminés, libèrer les subordonnés qui ont traité ces jobs.
4. Envoyer l'état de l'interlocuteur à son gestionnaire de groupe.

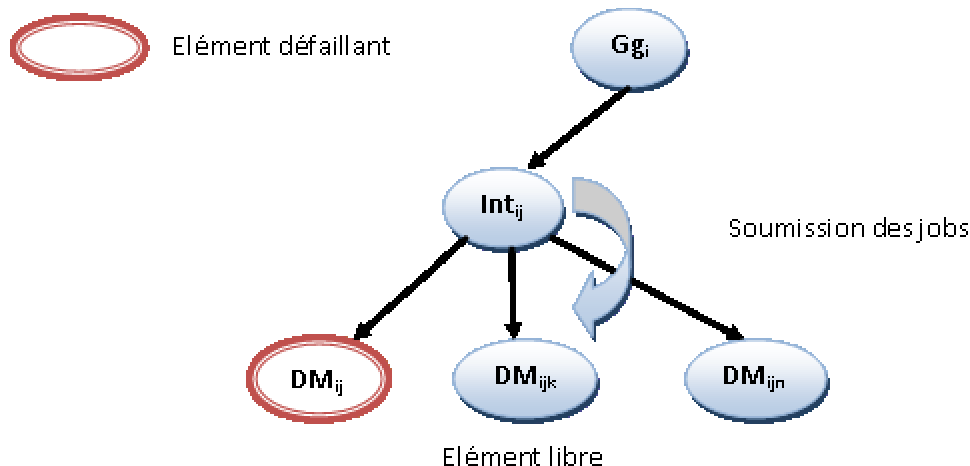

FIGURE 3.11 – Tolérance aux fautes locale (au niveau interlocuteur)

2. Tolérance aux fautes vers le niveau supérieur : Lorsque le gestionnaire de tolérance aux fautes au niveau interlocuteur détecte une faute d'un subordonné et qu'il ne trouve pas un nombre suffisant de subordonnés libres localement, (voir Figure 3.12), nous procédons comme suit :

1. Soumettre les jobs du subordonné défaillant à son gestionnaire de groupe avec les mêmes identifiants.

2. Envoyer l'état de l'interlocuteur à son gestionnaire de groupe.

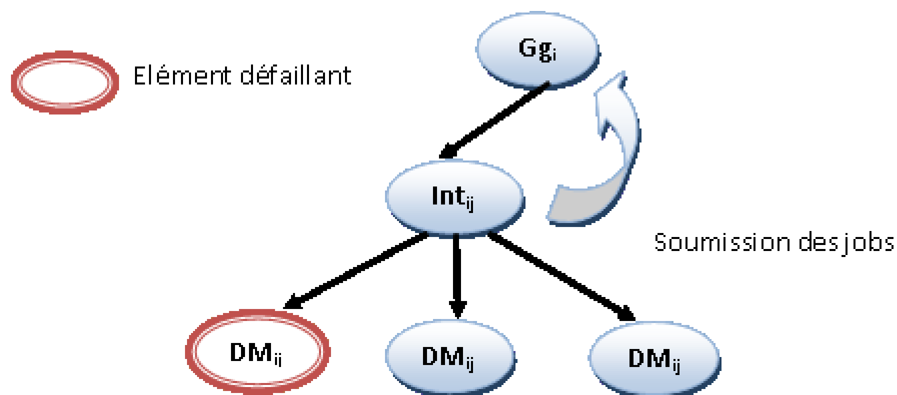

FIGURE 3.12 – Tolérance aux fautes vers le niveau supérieur (au niveau interlocuteur)

3.6.2 Tolérance au niveau gestionnaire de groupe

Au niveau d'un gestionnaire de groupe, nous pouvons avoir deux traitements concernant la tolérance aux fautes. Le premier vise à tolérer les fautes provenant des interlocuteurs et le deuxième concerne la tolérance au niveau des dispositifs mobiles.

1. **Fautes des interlocuteurs :** Lorsque le gestionnaire de groupes $Gg(i)$ ne détecte aucune réception de présence de vie de la part de l'interlocuteur $INT(i,j)$, il prend l'initiative de lancer le processus de tolérance aux fautes en remplaçant l'interlocuteur $INT(i,j)$ par sa réplique $INTD(i,j)$. Ce dernier prend en charge de gérer le groupe mobile $GM(i,j)$. Le gestionnaire élit un nouvel interlocuteur qui jouera le rôle de réplique pour le nouvel interlocuteur (voir Figure 3.13).

2. **Fautes des dispositifs mobiles :** Lorsque le gestionnaire de groupe reçoit des listes de jobs à tolérer à partir des interlocuteurs, il prend la charge de réallouer ces jobs à d'autres interlocuteurs qui ont des dispositifs mobiles libres (voir Figure 3.14).
   A : $INT(i,1)$ envoie les jobs non tolérés à son gestionnaire $Gg(i)$.
   B : $Gg(i)$ intercepte les jobs non tolérés et les envoie aux autres $INT(i,2)$.
   C : $INT(i,2)$ alloue les jobs aux $DM(i,2,n)$ libres.

3.7 Discussion

Dans le modèle proposé ci-dessus, nous avons mis en évidence les problématiques de tolérance aux fautes dans un environnement mobile. L'informatique mobile est un domaine en plein essor qui profite des percées technologiques dans le domaine des

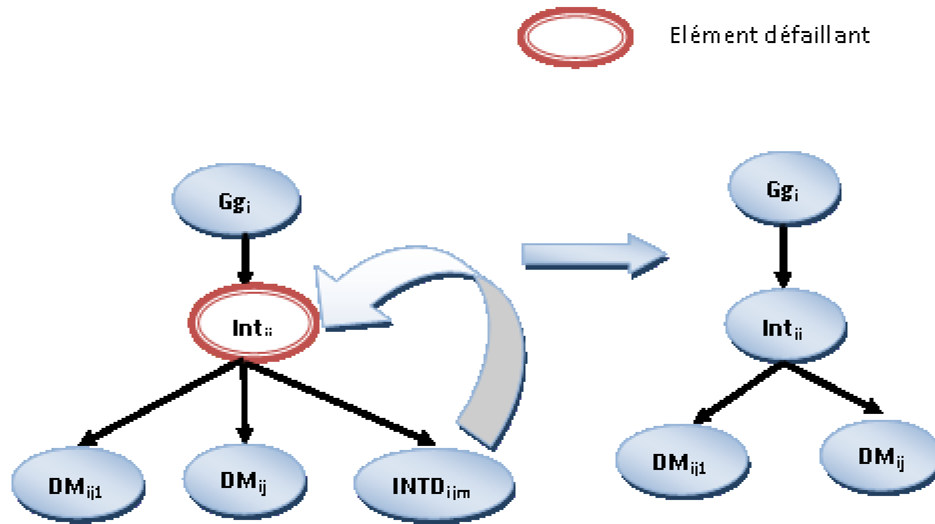

FIGURE 3.13 – Tolérance aux fautes des interlocuteurs (au niveau gestionnaire de groupe)

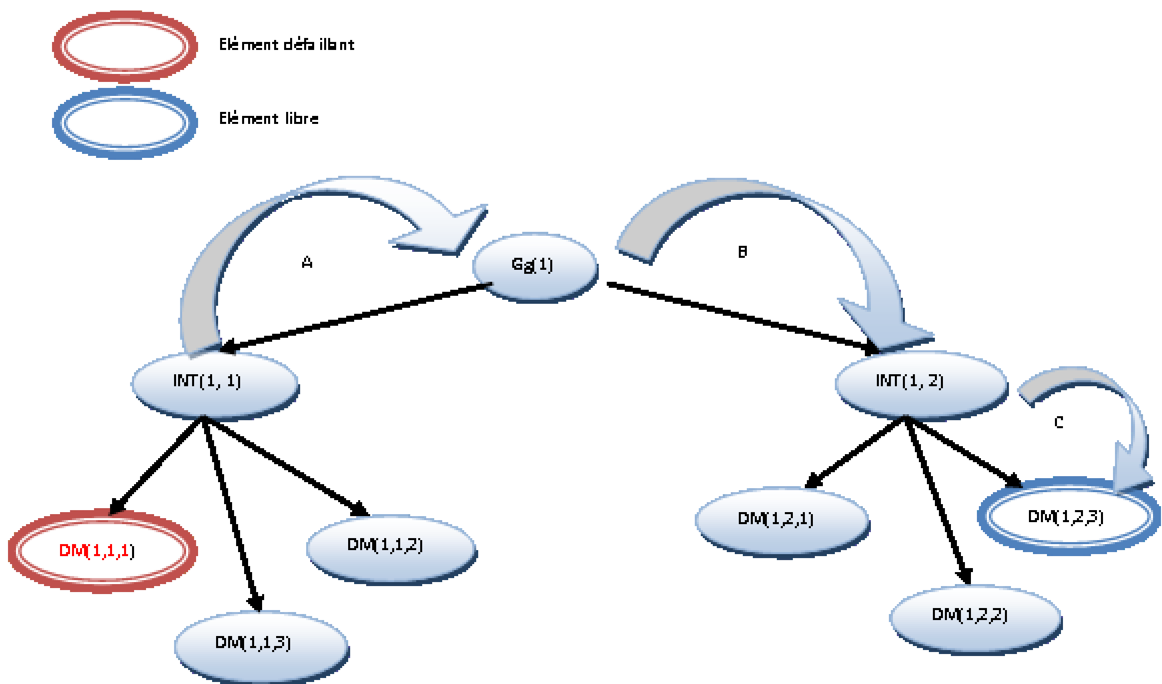

FIGURE 3.14 – Tolérance aux fautes des dispositifs mobiles (au niveau gestionnaire de groupe)

dispositifs mobiles et des réseaux de communication sans fil. Pour tolérer les fautes de ce type d'environnement, nous avons proposé de grouper un ensemble de dispositifs mobiles et pour en faire une seule ressource virtuelle. Nous avons ensuite défini un modèle multi-arborescent pour une grille mobile intitulé $G/I/S$, pour prendre en compte les contraintes liées à la mobilité de ces dispositifs mobiles.

3. MODÈLE MULTI-ARBORESCENT POUR UNE GRILLE MOBILE TOLÉRANTE AUX FAUTES

4 Conclusion

Une grille de calcul est une infrastructure particulièrement complexe, car elle est composée de milliers de machines hétérogènes et réparties géographiquement sur une très large échelle dans des institutions qui mettent en commun leurs ressources. De plus, de nouvelles ressources peuvent être ajoutées à tout moment, de même que des ressources présentes peuvent disparaître (défaillances, arrêts volontaires). Plusieurs techniques ont été développées dans le domaine de tolérance aux fautes à partir des systèmes distribués jusqu'à l'émergence de la technologie "grille" qui a mis les chercheurs devant des situations plus complexes (passage à l'échelle, hétérogénéité, dynamicité, asynchronisme, etc.). Pour prendre en charge l'aspect faute dans les grilles, nous avons proposé, dans ce chapitre, deux modèles hiérarchiques de tolérance aux fautes dans les grilles de calcul. Ces modèles permettent de transformer les ressources d'une grille en une structure hiérarchique, où chaque sous-ensemble est supervisé par un gestionnaire. Malgré que ces modèles ont donné généralement des résultats satisfaisants pour la majorité des cas, que nous avons expérimenté leurs performances se dégradent quant les ressources de la grille passent à l'échelle. En effet, nous avons constaté une surcharge du gestionnaire et une augmentation du volume de la structure. Pour remédier à ces limites, nous présenterons dans le chapitre suivant, deux modèles décentralisés basés sur les graphes colorés dynamiques.

Chapitre 4

Modèles de tolérance aux fautes basés sur les graphes colorés dynamiques

1 Introduction

Une grille de calcul est une architecture distribuée à grande échelle, hétérogène et hautement dynamique. Avec de telles caractéristiques, les fautes ne sont plus des exceptions mais deviennent une partie du comportement des grilles, d'où la nécessité d'un modèle de tolérance aux fautes pour assurer le bon fonctionnement des grilles. Dans notre proposition, qui sera décrite tout au long de ce chapitre, nous modélisons une grille de calcul par un graphe coloré dynamique, où l'ensemble des sommets représente les nœuds de la grille et chaque arc représente les communications entre les nœuds reliés par cet arc. Dans le premier modèle, nous colorons les sommets en tenant compte d'un ensemble d'attributs que nous définirons par la suite. Chaque attribut est colré par trois couleurs de base (rouge, vert et bleu) en fonction de son niveau de performance. Plus tard, nous définissons pour chaque nœud dans la grille trois classes de substituts (identique, plus performant et moins performant) suivant leurs niveaux de performance. La technique de tolérance aux fautes proposée offre au nœud défaillant un sensemble de subtituts par une recherche hiérarchique dans les trois classes de substituts trouvées [164].
Le deuxième modèle de tolérance aux fautes est basé sur la recherche, pour chaque nœud, d'un ensemble de substituts capables de le remplacer en cas de défaillance. Chaque nœud possède un ensemble des voisins collaborateurs qui acceptent de collaborer avec lui en cas de défaillance. Nous avons, également, défini un degré de tolérance (α) qui donne le nombre des nœuds optimal pour tolérer les défaillances dans une grille, puis nous avons coloré les sommets du graphe par trois couleurs de base suivant la valeur de α. Ainsi, nous obtenons des sommets instables (nombre de voi-

sins collaborateurs $<\alpha$), des sommets stables (nombre de voisins collaborateurs$=\alpha$) et des sommets hyperstables (nombre de voisins collaborateurs$>\alpha$.). De part ses caractéistiques, notre modèle a pu tolérer les fautes de type crash et celles relatives aux déconnexions des nœuds de la grille. Cette propriété a pu être obtenue grâce à une technique décentralisée basée sur la distribution des jobs sur les voisins collaborateurs et une deuxième technique basée sur l'élection d'un remplaçant capable de se substituer à un ensemble de nœuds défaillants [161].

2 Définitions

Un graphe G un couple $G=(X,U)$, où X est un ensemble non vide de sommets et U est un ensemble de paires de sommets de X. Ces paires sont appelées arêtes. $X(G)$ et $U(G)$ représentent, respectivement, l'ensemble des sommets et d'arêtes du graphe G. Nous utiliserons la notation $G=(X,U)$ pour représenter un graphe [46].

2.1 Terminologie

Avant de continuer la présentation de notre modèle de tolérance aux fautes basés sur les graphes, il est utile de présenter la terminologie qui sera utilisée tout au long de ce chapitre [23] (voir Tableau 4.1).

2.2 Graphe dynamique

Un graphe dynamique est un graphe qui subit des modifications au cours du temps. Ainsi, il est nécessaire de considérer le processus de l'évolution d'un graphe ainsi que la base temporelle dans laquelle ce processus s'exécute [151].
Définition 1 : Un graphe dynamique G est défini par un triplet (G_0,T,P) tel que :
- $G_0=(X_0,U_0)$ est un graphe statique initial.
- T est une base temporelle continue ou discrète.
- P est un processus d'évolution.

Nous noterons par G_t le graphe statique image du graphe dynamique G à l'instant t.

2.2.1 Notion de base temporelle

La base temporelle peut être continue ou discrète. En effet, dans l'hypothèse où le graphe étudié modélise un réseau spatial composé d'entités dynamiques. Cette dynamicité peut être caractérisée par la présence et l'absence de l'entité ou par un changement d'état. Celle-ci est une fonction continue du temps. La base temporelle d'un tel graphe doit donc, en toute généralité, être continue. Notons malgré tout, que ce formalisme n'exclut en rien la possibilité de considérer une base temporelle discrète, comme cela est fait classiquement dans le domaine de la simulation à temps discret. Cependant, le graphe dynamique, définit par une base temporelle discrète, peut alors être considéré comme inclus dans le graphe dynamique défini par une base

N^o	Terme	Signification
1	Graphe orienté	Chaque arc est représenté par la paire orientée (x_1, x_2) ; x_1 étant l'origine et x_2 l'extrémité de l'arc. Par conséquent, (x_2, x_1) et (x_1, x_2) représentent deux arcs différents
2	Graphe non orienté	La paire de sommets représentant une arête n'est pas ordonnée. Autrement dit, (x_1, x_2) et (x_2, x_1) représentent la même arête
3	Adjacence	Deux sommets sont adjacents s'il existe un arc, ou une arête, les reliant
4	Chemin	Suite d'arcs connexes reliant un sommet à un autre. Par exemple $(x_0, x_1)(x_1, x_2)(x_2, x_3)(x_3, x_4)(x_4, x_5)$ est un chemin reliant x_0 à x_5 ; on le note $(x_0, x_1, x_2, x_3, x_4, x_5)$
5	Chaîne	Séquence d'arêtes avec une extrémité commune dans un graphe non orienté
6	Degré d'un sommet	Nombre d'arcs arrivant et partant d'un sommet dans un arc orienté.
7	Stable	Un sous-ensemble S de points de X est dit stable si ses points ne sont pas adjacents entre eux.
8	Nombre chromatique	Le plus petit nombre de couleurs nécessaires pour colorier un graphe

TABLE 4.1 – Quelques éléments de la théoire des graphes

temporelle continue. De la même façon, plusieurs bases temporelles discrètes peuvent être envisagées pour la définition d'un graphe dynamique. Pour une propriété donnée existante dans le modèle de graphe dynamique à base temporelle continue, la meilleure "granularité temporelle" correspond à l'intervalle de temps durant lequel la propriété peut être observée.

2.2.2 Processus d'évolution

Le processus d'évolution d'un graphe dynamique peut être décrit selon un formalisme algorithmique. Il comprend également les "équations dynamiques", qui correspondent à des processus d'évolution locaux, dont le principe a été proposé et décrit par Bersini (2000) dans [17] puis dans son livre [18].
Si la base temporelle est discrète, alors le processus d'évolution peut s'exprimer par :
$P : N \circ G \rightarrow G$
$t, G_{t-1} \rightarrow G_t = P(t, G_{t-1})$
où N représente l'ensemble des entiers naturels et G l'ensemble des graphes statiques. Une telle base temporelle permet de définir des étapes, durant lesquelles le processus d'évolution agit sur le graphe courant. Si la base temporelle est continue, alors le processus d'évolution peut s'exprimer par :
$P : R_{+} \circ G \rightarrow G$
$t, G_{courant} \rightarrow G_{nouveau} = P(t, G_{courant})$
où R représente l'ensemble des réels et G l'ensemble des graphes statiques. Dans cette dernière formulation, rien n'empêche de considérer que P est un processus d'évolution continu.

2.3 Coloration des graphes

La coloration de graphes est un problème très connu, parfaitement illustré par le problème des quatre couleurs posé par Francis Guthrie en 1852 [110]. Il s'agit de savoir s'il est possible de colorer toute carte géographique avec au plus quatre couleurs de sorte que deux régions qui ont une frontière en commun aient des couleurs différentes. La coloration de graphes permet de modéliser de nombreux problèmes réels, depuis le placement de personnes autour d'une table ou de pièces sur un échiquier jusqu'au problème d'ordonnancement et de planning (réservation de ressources, logistique, transport, réseau, etc.) [196]. Il n'y a pas une seule façon de colorer les graphes mais plusieurs (un très grand nombre même). On peut déjà vouloir colorer différents éléments d'un graphe (les sommets, les arêtes, les faces, un mélange de ces éléments, des sous-structures, etc.) avec ou sans contraintes particulières. La contrainte la plus courante est celle qui spécifie que : deux éléments voisins doivent avoir des couleurs différentes. Bien que différents éléments d'un graphe puissent être colorés, il est toujours possible se ramener à un problème de coloration de sommets. C'est donc le paramètre de coloration le plus général (et le plus étudié) [110].

2.3.1 Définition

En général, il existe deux types de coloration : une coloration des sommets et une coloration des arêtes. La coloration des sommets (respectivement des arêtes) d'un graphe G correspond à l'affectation d'une couleur à chacun des sommets du graphe (respectivement à chacune des arêtes) de telle sorte que deux sommets (respectivement deux arêtes) adjacents ne soient pas porteurs de la même couleur. Un graphe est dit *p-chromatique* si ses sommets admettent une coloration en p couleurs [93]. nous appellerons nombre chromatique $\gamma(G)$ (respectivement indice chromatique $q(G)$),

le nombre minimum de couleurs distinctes nécessaires pour effectuer une coloration des sommets (respectivement des arêtes) de G.

2.3.2 Coloration des sommets

La coloration des sommets d'un graphe consiste à affecter à tous les sommets de ce graphe une couleur, de telle sorte que deux sommets adjacents ne portent pas la même couleur. Une coloration avec k couleurs est donc une partition de l'ensemble des sommets en k stables (voir Tableau 4.1). Le nombre chromatique d'un graphe $G = (X, U)$, noté $g(G)$, est le plus petit entier k pour lequel il existe une partition de V en k sous-ensembles stables [46].

2.3.3 Coloration des arêtes

La coloration des arêtes d'un graphe consiste à affecter à toutes les arêtes de ce graphe une couleur q, de telle sorte que deux arêtes adjacentes ne portent pas la même couleur [110].

3 Graphe coloré dynamique

Dans cette thèse, nous proposons un nouveau type de graphe, appelé graphe coloré dynamique, et nous ne parlerons pas de coloration de graphes. La couleur peut être associée à un sommet pour modéliser la différence existante entre les sommets qui ne sont pas forcément de même nature (voir Figure 4.1). Nous définissons un graphe coloré comme suit :

Définition 2 : (GRAPHE COLORE)
Un graphe coloré $G(X, U, C, E)$ est un quadruplet tel que :

- X : ensemble fini non vide de sommets.
- $U : X \rightarrow X$: ensemble des arêtes.
- C : ensemble non vide de couleurs.
- $E : X \rightarrow C$: ensemble des couleurs attribuées à chaque sommet.

Etant donné qu'un graphe coloré peut être dynamique, nous définissons un graphe coloré dynamique comme suit :

Définition 3 : (GRAPHE COLORE DYNAMIQUE)
Un graphe coloré dynamique est défini par un triplet (G_0, T, P) comme suit :

- $G_0(X, U, C, E)$ est un graphe coloré statique initial (instant 0).
- T est une base temporelle continue ou discrète.
- P est un processus d'évolution.

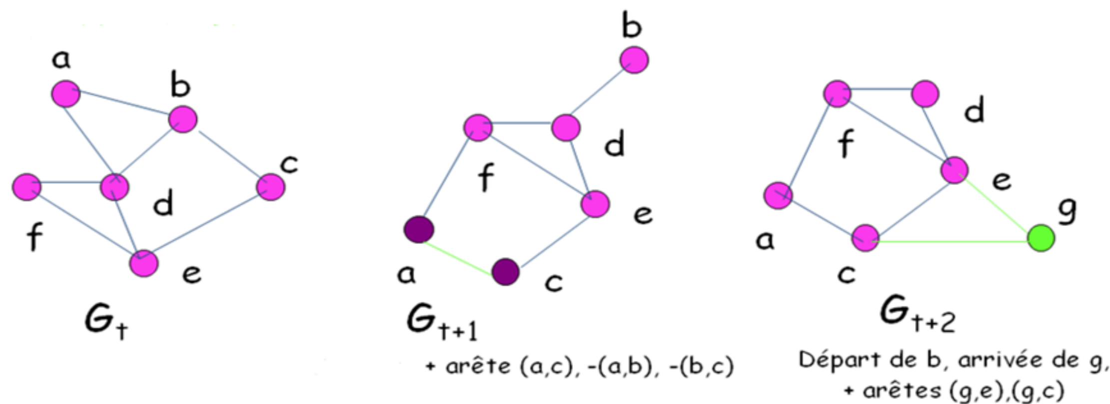

FIGURE 4.1 – Exemples de graphes colorés dynamiques

4 Modèle de tolérance aux fautes basé sur les niveaux de performance

Dans cette section, nous présentons le premier modèle de tolérance aux fautes basé sur les niveaux de performance des nœuds d'une grille de calcul.

4.1 Environnement du système de grille

L'Open Grid Services Architecture (OGSA) décrit une architecture des environnements de grille de calcul orientés services à usage professionnel et scientifique [59]. Un des éléments le plus important de cette architecture est le système de gestion des ressources (*RMS : Resource Management System*). Un nœud virtuel (NV) est une unité générale intégrée dans la grille, qui peut exécuter des jobs ou partager des ressources, comme un ordinateur, un élément de calcul, un système embarqué, un produit numérique, etc. Dans ce modèle, nous considérons un NV comme un site qui héberge un ensemble de nœuds et chaque nœud peut avoir un seul ou plusieurs éléments de calcul, la taille de la RAM, disque dur disponible, etc. Les nœuds d'un site sont reliés par un réseau d'interconnexion rapide. Chaque site utilise un système de file d'attente pour exécuter des jobs, i.e., quand un job est soumis au site, il est mis dans le système de file d'attente jusqu'à ce qu'il puisse être affecté à un nœud. Les sites sont reliés entre eux par des liens de réseau. Cependant, le RMS est un composant logiciel qui s'exécute sur chaque site. Nous supposons que les jobs sont mutuellement indépendants et peuvent être exécutés sur n'importe quel site à condition que le site peut répondre à la demande des ressources et la quantité de transmission de données.
Dans le modèle que nous proposons dans ce chapitre, nous avons fait les hypothèses suivantes :

1. Le RMS est entièrement fiable et le temps de traitement d'un job (envoyer les jobs aux ressources, recevoir les résultats et envoyer la réponse à l'utilisateur) est négligeable par rapport au temps de traitement du job.

2. Les fautes des nœuds et des liens sont indépendantes.
3. Il n'y a aucune faute de logiciel.
4. Les informations connexes, tels que la vitesse de CPU et la taille mémoire des ressources devraient être fournies lorsqu'un utilisateur demande des ressources au RMS.

4.2 Modélisation de grille par un graphe coloré dynamique

Nous modélisons toute grille de calcul par un graphe non orienté $G(X,U)$, où X est un ensemble des sommets du graphe tel que chaque sommet correspond à un nœud et $U = \{u_1, u_2, ..., u_m\}$ représente l'ensemble des arêtes du graphe, tel que chaque arête correspond à une connexion physique entre deux nœuds. Chaque arête $u_i \in U$ est muni d'un poids non-négatif qui représente la bande passante. La distance entre deux sommets x et y, notée $d(x, y)$, est la somme des poids des arêtes du plus court chemin entre x et y. Pour prendre en compte la dynamicité des nœuds d'une grille, le graphe G sera dynamique, i.e., qu'à tout instant t, le graphe peut subir des adjonctions et/ou des suppressions de sommets ou d'arêtes. Pour cela, nous définissons par $G_t(X,U)$, l'ensemble des sommets et arêtes présents à l'instant t. $Voisin_t(x)$ représente l'ensemble des sommets voisins du sommet x à l'instant t. Pour prendre en compte l'hétérogénéité des nœuds, nous associons à chaque sommet du graphe un ensemble de couleurs indiquant le type du nœud qui lui est associé. De cette manière, l'ensemble des couleurs représente les types de nœuds présents dans la grille (voir Définition 2). En regroupant toutes ces propriétés, nous pouvons modéliser toute grille de calcul par un graphe coloré dynamique (voir Définition 3). Dans cette partie, nous modélisons toute grille de calcul par un graphe coloré dynamique et nous proposons un modèle de tolérance aux fautes pour les grilles de calcul basé sur ces graphes. Ce modèle présente deux caractéristiques fondamentales : (i) la première concerne la prise en charge des défaillances des nœuds : elle définit, pour chaque nœud défaillant trois ensembles de substituts potentiels (identiques, plus performants et moins performants) ; (ii) la deuxième caractéristique de notre modèle est que nous tolérons les fautes sans aucune réplication et sans aucune restructuration des ressources d'une grille.

4.3 Règle de coloration des sommets

Notre objectif est de trouver, pour chaque nœud de la grille qui tombe en panne, des substituts (nœuds) qui sont en mesure de le remplacer. La détermination de ces substituts dépend d'un ensemble d'attributs. Pour chaque nœud de la grille, nous définissons un ensemble de k attributs $A = \{a_1, a_2, ..., a_k\}$, dont les valeurs peuvent être statiques ou dynamiques selon l'état du nœud. Chaque nœud x possède un vecteur d'état $V_t(x) = [va_1, va_2, ..., va_k]$ qui définit les valeurs des attributs de l'ensemble A à l'instant t. Pour chaque attribut a_i, un intervalle de performance, noté $I_i = [vi_1..vi_2]$, est défini au préalable qui est unique pour toutes les nœuds de la grille. Si la valeur $va_i \in I_i$, nous colorons cet attribut par la couleur verte. Si $va_i < vi_1$, le nœud sera considéré comme moins performant pour cet attribut et nous le colorons en rouge, et si $va_i > vi_2$, le nœud sera considéré comme plus performant

pour cet attribut et nous le colorons en bleu. Nous pouvons ainsi modéliser une grille par un graphe coloré dynamique, où chaque sommet (associé à un nœud) x possède un vecteur d'état $V_t(x)$ composé de k attributs colorés par trois couleurs de base (rouge, vert, bleu) en fonction des valeurs de chaque attribut par rapport à l'intervalle de performance prédéfini (voir Figure 4.2).

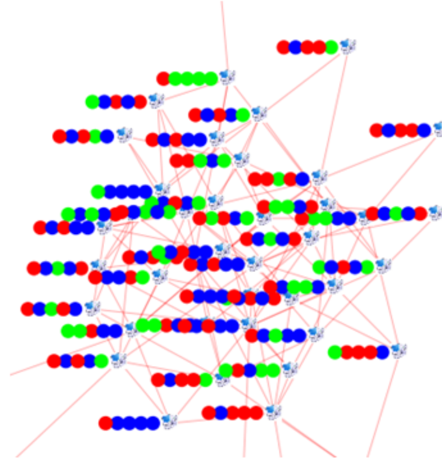

FIGURE 4.2 – Graphe coloré dynamique avec 5 attributs

4.4 Transmission du vecteur d'état

Dans le modèle proposé, chaque sommet x du graphe agrège les vecteurs d'états des tous les sommets du graphe. $Subs_t(x)$ représente l'ensemble des $V_t(y)$ de tous les sommets du graphe au niveau du sommet x, $Subs_t(x) = \bigcup_{(y \in X)} V_t(y)$. La construction de $Subs_t(x)$ est progressive. Au départ, chaque sommet x envoie à tous ses voisins son vecteur d'état $V_t(x)$ et la liste de ses voisins, $SV_t(x) = Voisin_t(x)$, représentant les sommets qui ont déjà reçu $V_t(x)$. Dès qu'un sommet y reçoit $V_t(x)$ et $SV_t(x)$, il met à jour $Subs_t(y)$ par l'insertion de $V_t(x)$ ou sa mise à jour s'il appartient déjà à $Subs_t(y)$. Par la suite, il construit l'ensemble $SV_t(x) = SV_t(x) U \{Voisin_t(y) - S_{t-1}V(x)\}$ et il transmet $V_t(x)$ avec $SV_t(x)$ aux sommets $SS(x) = Voisin_t(y) - SV_t(x)$, qui représente les sommets qui n'ont pas encore reçu $V_t(x)$. Ce processus est réalisé par chaque sommet jusqu'à $SS(x) = \emptyset$.

4.5 Tolérance aux fautes

La technique de tolérance aux fautes, proposée, se charge de trouver, pour chaque nœud défaillant, un ensemble de substituts capables de le remplacer. Nous supposons l'existence d'un détecteur de fautes fiable capable de détecter les fautes dans une grille [2, 220]. La détermination des substituts dépend des besoins de l'utilisateur : il peut, par exemple, s'intéresser au critère de rapidité, i.e., trouver le premier substitut sans prendre en compte d'autres critères. Il peut aussi s'intéresser à trouver des substituts ayant des performances similaires ou supérieures au nœud défaillant. Il peut également s'intéresser uniquement à un sous-ensemble d'attributs et chercher

les substituts qui ont des performances similaires ou supérieures au nœud défaillant, sur la base du sous-ensemble d'attributs qu'il a choisi.

4.5.1 Calcul des substituts

Pour chaque sommet x du graphe, avec un vecteur d'attributs $V_t(x) = [va_1, ..., va_k]$, nous définissons, pour chaque attribut a_i, un coefficient α_i indiquant l'ordre de priorité de cet attribut. Puis, nous calculons sa valeur de performance comme suit :

$$\delta_x = \sum_{i=1}^{k} va_i * \alpha_i. \tag{4.1}$$

Si l'attribut a_j peut être plus performant si ses valeurs sont petites (par exemple l'attribut latence), nous utilisons la notation $1/a_j$ ($a_j \neq 0$). A partir de ce critère de performance, nous définissons trois classes de substituts de chaque nœud défaillant comme suit :

Définition 4 : Deux sommets x et y du graphe sont identiques si et seulement si $\delta_x = \delta_y \mp \epsilon$.

Définition 5 : Un sommet x est plus performant qu'un sommet y si et seulement si $\delta_x > \delta_y + \epsilon$.

Définition 6 : Un sommet x est moins performant qu'un sommet y si et seulement si $\delta_x < \delta_y - \epsilon$.

ϵ est une petite valeur positive, elle permet de déterminer les sommets y identiques au sommet x dans l'intervalle $[\delta_y - \epsilon,\ \delta_y + \epsilon]$. Elle est de même utilisée pour les sommets plus performants et moins performants.

Chaque sommet x classe tous les sommets du graphe en trois ensembles disjoints E_{id}, E_{+perf} et E_{-perf} représentant respectivement, les sommets ayant des performances identiques à x, les sommets ayant des performances supérieures que les siennes, et les sommets ayant des performances inférieures par rapport à ses performances.

4.5.2 Sélection des substituts

Quand un nœud x tombe en panne, la technique de tolérance aux fautes, proposée, tente de trouver un ou plusieurs substituts ($subst_remp$) capables de le remplacer, afin d'assurer la continuité du service assuré par le nœud x. La sélection des substituts dépend de la technique de tolérance adoptée et du nombre de substituts nécessaires pour remplacer le nœud défaillant. La sélection des substituts dépend du critère de performance et du niveau de voisinage, en sélectionnant le substitut le plus proche du nœud défaillant. Supposons que le nœud défaillant nécessite n substituts ; premièrement, nous commençons la recherche des substituts dans l'ensemble E_{id}, qui est composé de substituts ayant tous des performances identiques au nœud défaillant. Dans ce cas, nous choisissons les substituts les plus proches, $subst_remp(x, n) = \{y | y \in E_{id}$ and $Min\ d(x, y)\}$. $R1 = |subst_remp(x, n)|$ représente le nombre de substituts trouvés dans E_{id}. Si $R1 < n$, nous complétons la recherche dans l'ensemble E_{+perf}, et nous sélectionnons les substituts les plus proches, $subst_remp(x, n - R1) = \{y | y \in E_{+perf}$ and $Min\ d(x, y)\}$, ce qui donne $R2 = |subst_remp(x, n - R1)|$. Si $R1 + R2 < n$, nous continuons la recherche dans l'ensemble E_{-perf} du même processus de sélection dans E_{+perf}.

4.5.3 Types de dynamicité

Le modèle que nous avons défini admet deux types de dynamicité : (i) une dynamicité des composants du graphe (sommets et arêtes) ; et, (ii) une dynamicité des vecteurs d'états des sommets qui changent en fonction des valeurs de leurs attributs.

a Dynamicité du graphe : La dynamicité du graphe est caractérisée par l'addition et/ou la suppression d'arêtes et/ou de sommets. Tout d'abord, nous nous intéressons aux modifications du graphe suite à des défaillances de sommets. Pour celà, nous définissons un délai t_{max} où chaque sommet doit transmettre un message de vie à tous ses voisins. Dépassé ce délai, il sera considéré comme un sommet absent du graphe et ses voisins se chargeront de transmettre son absence à tous les autres sommets du graphe. Quand un nouveau sommet s'ajoute dans le graphe, il transmet son vecteur d'état $V_t(x)$ à ses voisins, qui le propagent dans tout le graphe et il demande à un de ses voisins de lui transmettre la liste, $Subs_t(x)$, complète des vecteurs d'états de tous les sommets du graphe pour sa mise à jour. Pour l'adjonction d'arêtes, les deux sommets voisins échangent leurs vecteurs d'états et pour la suppression, elle sera constatée entre les deux sommets.

b Dynamicité du vecteur d'état : Quant un sommet x change son $V_t(x)$, il le transmet à tous ses voisins, qui le propagent dans tout le graphe. Chaque sommet ayant reçu un $V_t(x)$, recalcule son δ_x pour déterminer son ensemble de substituts.

4.6 Discussion

Nous avons présenté, dans les sections précédentes, un modèle transformant toute grille de calcul en un graphe coloré dynamique. Partant de ce graphe, nous avons proposé, dans ce chapitre, un mécanisme de tolérance aux fautes, dans lequel chaque sommet possède un vecteur d'état regroupant ses propriétés qui sont ensuite propagées dans tout le graphe. Chaque sommet classifie les autres sommets du graphe en trois catégories de substituts : identiques, moins performants et plus performants que ses capacités. Ces substituts sont des remplaçants potentiels du nœud en cas de défaillance. Le mécanisme de tolérance aux fautes explore l'ensemble des nœuds en cherchant d'abord dans les nœuds identiques puis dans les nœuds plus performants et enfin dans les nœuds moins performants, afin de déterminer l'ensemble des substituts qui sont en mesure de remplacer le nœud défaillant. Le modèle proposé peut être exploité pour d'autres environnements similaires aux grilles en terme de dynamicité, d'hétérogénéité et de passage à l'échelle comme les réseaux mobiles. Il est possible, avec quelques adaptations, de l'exploiter égalament pour étudier d'autres problématiques des grilles de calcul comme l'équilibrage de charges, la réplication dans les grilles de données et la sécurité.

5 Modèle de degré de tolérance

Dans cette partie, nous modélisons toute grille de calcul par un graphe coloré dynamique et nous proposons un modèle de tolérance aux fautes pour les grilles de calcul basé sur ces graphes, notre contribution est la proposition pour chaque nœud de la grille un ensemble de voisins collaborateurs capables de le remplacer en cas de panne, le nombre de ces voisins collaborateurs est limité par un seuil α. Dans la phase de coloration, nous déterminons pour chaque nœud, à partir de ses nœud voisins (liaison directe), un ensemble des voisins collaborateurs. A partir de cet ensemble, nous classifions les nœuds de la grille en trois catégories (instable, stable et hyperstable). Puis dans la phase de stabilisation, chaque nœud instable tente de se stabiliser par le biais des nœuds hyperstables dans tout le graphe, puis nous proposons un modèle de tolérance aux fautes décentralisé basé sur les graphes colorés dynamiques, s'appuyant sur la coopération des voisins collaborateurs de chaque nœud ou par l'élection d'un nœud remplaçant chargé de tolérer la faute [161].

5.1 Règle de coloration

Chaque sommet x du graphe consulte ses voisins $Voisin_t(x)$ pour une collaboration mutuelle en cas de panne. Ces sommets sont considérés comme des Voisins Collaborateurs de x, noté ($VCol_t(x)$). Nous définissons, pour tout le graphe, un seuil α déterminant le nombre de voisins collaborateurs de x, qui sont suffisants pour tolérer convenablement les fautes d'un sommet x. Ce seuil représente, dans notre modèle, un degré de tolérance. $C_t(x)$ représente la couleur du sommet x à l'instant t ; pour tout sommet x, si $|VCol_t(x)| = \alpha$, il sera coloré en vert ($C_t(x) = V$) et il sera considéré comme sommet **stable** ; si $|VCol_t(x)| < \alpha$, le sommet x sera coloré en rouge ($C_t(x) = R$) et il sera considéré comme sommet **instable** ; finalement, si $|VCol_t(x)| > \alpha$, alors le sommet x sera coloré en bleu ($C_t(x) = B$) et il sera considéré comme sommet **hyperstable** (voir Figure 4.3).

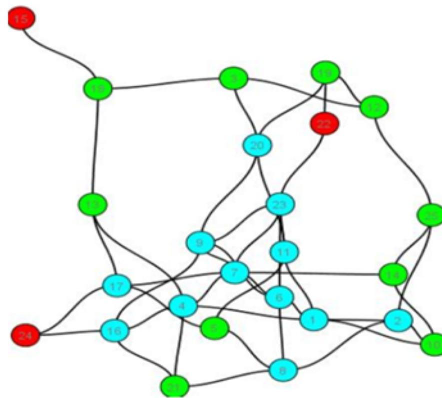

FIGURE 4.3 – Graphe coloré dynamique (25 sommets, 45 arcs avec α=3).

5.2 Calcul du degré de tolérance

Dans notre modèle, la valeur du degré de tolérance α est déterminante pour la coloration des sommets du graphe. Pour celà, nous avons défini 4 méthodes pour calculer ce degré :

1. **M1 : Moyenne des voisins :** Cette méthode consiste à calculer le rapport de la somme des voisins de tous les sommets du graphe sur le nombre de sommets. $\alpha = \frac{\sum_{x \in X} Voisin(x)}{|X|}$
2. **M2 : Modulo :** Cette méthode sélectionne le nombre de voisins le plus répété dans le graphe.
3. **M3 : Méthode itérative :** Dans cette méthode, le degré de tolérance est calculé de manière itérative : Premièrement, α prend comme valeur, pour un sommet x, le plus grand nombre de voisins dans le graphe ($\alpha = Max(Voisin(x))$,$\forall x \in X$). Par la suite, ce seuil est réduit jusqu'à l'obtention d'un nombre de sommets stables supérieur au nombre de sommets instables.
4. **M4 : Méthode directe :** l'utilisateur définit sa propre valeur de α .

5.3 Protocole de stabilisation

Une fois les sommets colorés, les sommets instables (ayant une couleur rouge) tentent de se stabiliser par le biais des sommets hyperstables. Un sommet x ayant une couleur $C_t(x) = R$ explore l'ensemble des $Voisin_t(x)$ en cherchant les sommets hyperstables $Hyp_t(x) = \{y/C_t(y) = B$ and $y \in Voisin_t(x)\}$. Le sommet x envoie alors des requêtes aux sommets y de l'ensemble $Hyp_t(x)$ pour obtenir l'ensemble des sommets qui acceptent de collaborer avec x $\{z/z \in VCol_t(y)$ and $z \notin Voisin_t(x)$ and $z \in Hyp_t(y)\}$. Ceci va réduire le nombre de voisins collaborateurs $VCol_t(y)$ chez le sommet y jusqu'à α (y devient stable à ce moment là) et pour le sommet x, le processus de recherche s'arrête quand il devient stable. Dans le cas contraire, le sommet x cherche dans le graphe le sommet hyperstable le plus proche pour se stabiliser. Ce processus de recherche s'arrête quand il ne sera plus possible de changer les couleurs des sommets dans le graphe (voir Figures 4.4 et 4.5).

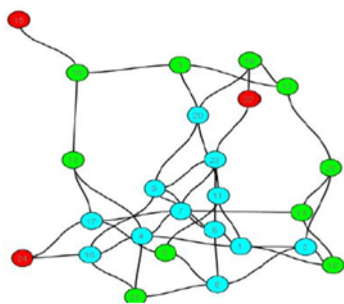

FIGURE 4.4 – Graphe après coloration

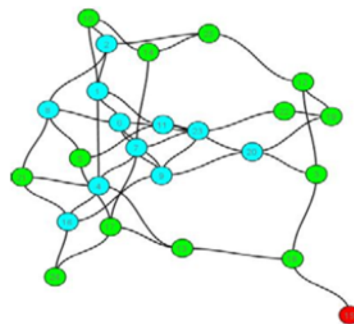

FIGURE 4.5 – Graphe après stabilisation

5.4 Formulation mathématique du modèle

5.4.1 Etat des sommets du graphe

Les sommets du graphe sont colorés durant la phase de coloration par les trois couleurs de base (rouge, vert et bleu). Dans la phase de stabilisation, les sommets changent de couleurs, suivant les lemmes 1 et 2 définis ci-dessous, où chaque sommet de couleur rouge tente de se stabiliser en consommant le surplus de voisins collaborateurs des sommets de couleur bleu.

Lemme 1 :
Un sommet $x \mid C_t(x) = R$ et $|VCol_t(x)| = \beta$ avec $(\beta < \alpha)$ ne peut changer sa couleur qu'en vert si et seulement si $\exists A = \{y|C_t(y) = B$ et $\sum |VCol_t(y)| = \alpha - \beta\}$.

Preuve :
Soit un sommet x ayant un ensemble de voisins collaborateurs dont le nombre est inférieurs à α. Son changement de couleur est exigé par l'existence d'un ou plusieurs sommets ayant un nombre de voisins collaborateurs supérieur à α et qui peuvent compléter les voisins collaborateurs de x jusqu'à ce qu'il atteigne α et sa couleur deviendra verte, sinon sa couleur restera rouge (voir Figure 4.6).

Lemme 2 :
Un sommet $x \mid C_t(x) = B$ et $|VCol_t(x)| = \beta$ $(\beta > \alpha)$ ne peut changer sa couleur qu'en vert si et seulement si $\exists A = \{y|C_t(y) = R$ et $\sum |VCol_t(y)| = \beta - \alpha\}$.

Preuve :
Soit un sommet x avec un nombre de voisins collaborateurs supérieur à α. Son changement de couleur est exigé par l'existence d'un ou plusieurs sommets ayant un nombre de voisins collaborateurs inférieur à α et qui peuvent consommer les voisins collaborateurs de x jusqu'à ce qu'il atteigne le seuil α et sa couleur deviendra vert, sinon sa couleur restera bleu (voir Figure 4.6).

La Figure 4.6 illustre les changements de couleur possibles des sommets du graphe :

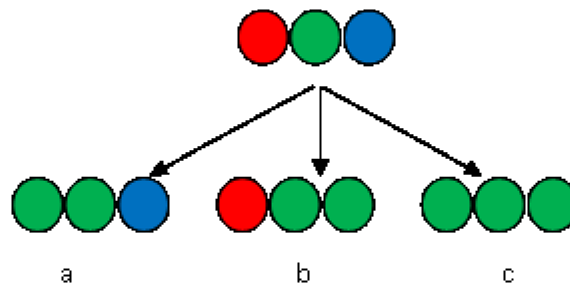

FIGURE 4.6 – Différents états, en terme de couleurs, des des sommets du graphe

- Dans la Figure 4.6 (a), le sommet instable (de couleur rouge) se stabilise (devient vert) par les voisins collaborateuts du sommet hyperstable et ce dernier reste toujours hyperstable.
- Dans la Figure 4.6 (b), le sommet instable n'arrive pas à se stabiliser malgré qu'il a pris tous les voisins collaborateurs du sommet hyperstable qui devient stable.

– Dans la Figure 4.6 (c), le sommet instable se stabilise avec tous les voisins collaborateurs du sommet hyperstable et ils deviennent tous stables.

5.4.2 Etats du graphe

Les phases de coloration et de stabilisation génèrent des graphes colorés de différentes formes. Aussi, nous allons nous intéresser à l'ensemble des couleurs qui peuvent exister dans un graphe à un instant t.

Définition 7 :
L'état du graphe G, noté ($Etat_t(G)$), est l'ensemble des couleurs présentes dans le graphe à un instant t.
$Etat_t(G) = VB$ signifie que l'ensemble des couleurs présentes dans G sont vertes et bleues (voir Figure 4.7). A partir des trois couleurs de base (Rouge, Vert, Bleu),

FIGURE 4.7 – Graphe coloré dynamique (8 sommets, 18 arcs et α=2). $Etat_t(G) = VB$.

l'ensemble des états possibles d'un graphe est défini par $Etat_t(G) = \{V, R, B, VR, VB, RB, RVB\}$. Ceci nous donne l'ensemble des différents états d'un graphe avec le lien entre les phases de coloration et de stabilisation (voir Tableau 4.2)

Etat	Coloration	Stabilisation
R	Oui	Oui
B	Oui	Oui
V	Oui	Oui
VB	Oui	Oui
VR	Oui	Oui
RB	Oui	Non
RVB	Oui	Non

TABLE 4.2 – Différents états d'un graphe avec les phases d'apparition associées

Trois cas sont possibles :

1. $Etat_t(G) = R$ ou B : ces deux états ne peuvent être générés qu'à partir de la phase de coloration et ils sont stables directement ; nous les appelerons les Etats isolés.

2. $Etat_t(G) = V$ ou VB ou VR : ces trois états peuvent être obtenus à partir de la coloration, ou à partir des états du 3eme cas, et ils sont aussi stables. Ils représentent des Etats terminaux.

3. $Etat_t(G) = RB$ ou RVB : ces deux états sont obtenus uniquement à partir de la coloration et ils représentent des états intermédiaires qui convergent toujours vers un état terminal. L'état RB converge vers les états V ou VB et l'état RVB converge vers les états terminaux V, VB ou VR (voir Figure 4.8).

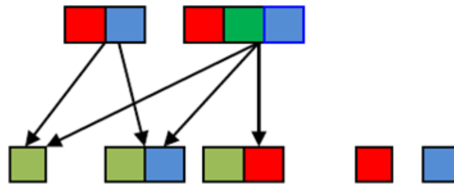

FIGURE 4.8 – Différents états du graphe

Théorème :
Un graphe d'états converge toujours vers un état stable $\{V, R, B, VR, VB\}$.
Preuve :
Pour tout graphe G, nous avons $Etat_t(G) = \{V, R, B, VR, VB, RB, RVB\}$. D'après les lemmes 1 et 2, seuls les sommets de couleurs rouge et bleu peuvent changer de couleur. Les sommets cessent de changer de couleurs s'il n'existe pas simultanément des sommets rouges et bleus. Nous pouvons donc conclure que G converge toujours vers des états isolés et terminaux.

5.5 Tolérance aux fautes

Dans cette section, nous allons nous intéresser au problème de fautes dans les grilles de calcul. En exploitant les graphes colorés dynamiques, nous proposons un modèle de tolérance aux fautes basé sur une approche décentralisée et sur une technique de tolérance aux fautes qui utilise un nombre optimal de voisins collaborateurs, chargés de tolérer les fautes de chaque nœud dans une grille. Après la détermination du degré de tolérance, nous lançons la coloration du graphe. Cette phase de coloration peut donner des sommets instables, stables et hyperstable. Partant de ce résultat, nous tentons de stabiliser le graphe, en utilisant les sommets hyperstables. Pour celà, nous proposons deux modèles de tolérance aux fautes :

1. Distribution sur les voisins collaborateurs

2. Désignation d'un remplaçant pour chaque nœud défaillant.

5.5.1 Type de fautes traitées

Dans le cadre de notre proposition, nous avons traité les types de fautes suivants :
Faute crash d'un nœud : ce type de faute représente l'absence d'un nœud à l'instant t. Tous les jobs et les résultats au niveau de ce nœud sont perdus.
Faute de déconnexion entre deux nœuds : cette faute concerne l'absence de connexion entre deux nœuds à l'instant t.

5.5.2 Modèle de tolérance aux fautes par distribution sur les voisins collaborateurs

Notation :

- Nœud x : ressource qui exécute des jobs.
- Voisin du nœud x : nœud qui a une relation directe avec le nœud x.
- Voisins collaborateurs du nœud x : voisins qui acceptent de collaborer en cas de défaillance du nœud x.

Relations :

- Nœud et voisin collaborateur : un nœud qui exécute des jobs dans une grille, met à jour périodiquement ses voisins collaborateurs en envoyant une partie de ses jobs.
- Voisin collaborateur et nœud : cette relation représente un voisin qui accepte de recevoir une partie des jobs d'un nœud. Il ajoute alors le nœud dans sa liste des voisins.

Description : Dans ce modèle, nous avons deux catégories des nœuds, les voisins et les voisins collaborateurs. En effet, chaque nœud a des voisins et parmi ses voisins il y a des voisins qui acceptent de collaborer. La relation entre ces acteurs doit être toujours vérifiée pour un bon fonctionnement du modèle. Cette relation est maintenue grâce aux mises à jour entre les nœuds et leurs voisins collaborateurs. Le modèle s'appuie sur une technique de tolérance aux fautes décentralisée et auto-organisée. Celà veut dire que chaque nœud est chargé de tolérer les fautes d'un de ses voisins collaborateurs et en même temps il est toléré en cas de détection de fautes par un de ses voisins collaborateurs. Dans notre modèle, il n y a pas de nœud superviseur ou administrateur.
Dans ce qui suit, nous allons détailler le fonctionnement de notre modèle :

1. **Détection des voisins pour chaque nœud :** tous les nœuds qui ont une liaison directe entre eux sont voisins les uns par rapport aux autres.
2. **Calcul du degré de tolérance α :** ce calcul sert à déterminer le nombre optimal de voisins collaborateurs pour chaque nœud.
3. **Coloration des nœuds :** En fonction du degré de tolérance α, nous colorons les nœuds en trois couleurs de base (Rouge, Vert, Bleu).
4. **Stabilisation des nœuds :** les nœuds qui ont un nombre de voisins inférieur à α tentent de se stabiliser c'est-à-dire de trouver d'autres voisins qui acceptent de collaborer avec eux.
5. **Mises à jour :** des mises à jour périodiques entre les nœuds et leurs voisins collaborateurs sont nécessaires (envoi à chaque voisin collaborateur une partie de la liste des jobs).

6. **Détecteur de fautes :** chaque nœud contient un détecteur de fautes qui informe ses voisins de l'état du nœud (en situation de crash ou non).
7. **Tolérance aux fautes :**
   Deux cas sont possibles :
   Cas 1 : le nœud a un seul voisin avant stabilisation : dans ce cas, il est traité comme en état de crash et ses jobs seront tolérés par ses voisins collaborateurs.
   Cas 2 : le nœud a plus d'un voisin avant stabilisation : le détecteur de fautes informe les voisins collaborateurs de ce nœud, du type de faute détecté : crash ou déconnexion :
   Cas2a : Faute crash : les voisins collaborateurs ditribuent les jobs de ce nœud.
   Cas2b : Faute de déconnection : le nœud tente de se stabiliser à nouveau.

5.5.3 Modèle de tolérance aux fautes par désignation d'un remplaçant

Notation :
- Remplaçant : nœud qui a pour rôle de tolérer les jobs d'un nœud défaillant.

Description : Notre modèle comprend des nœuds voisins, des nœuds voisins collaborateurs et des nœuds remplaçants. Les mises à jour se feront uniquement entre le nœud et son remplaçant, et ce dernier se charge de tolérer les fautes du nœud à travers ses voisins collaborateurs.
Les étapes du modèle sont définies comme suit :

1. Calculer le degré de tolérance α,
2. Colorer les sommets du graphe suivant la valeur de α,
3. Stabiliser le graphe,
4. Lancer le détecteur de fautes pour savoir si le nœud est en état crash ou non.
5. Sélectionner un remplaçant : chaque nœud désigne un remplaçant parmi ses voisins collaborateurs ayant une connexion directe avec lui.
6. Mettre à jour périodiquement le nœud et son remplaçant. Cette mise à jour concerne tous les jobs du nœud.
7. En cas de déconnexion entre un nœud et son remplaçant :
   1- Le nœud désigne un autre remplaçant.
   2- Deux cas sont cas possibles :
   Cas 1 : ce nœud a un seul voisin avant la stabilisation : il est traité comme s'il était en état de crash.
   Cas 2 : il a plus qu'un nœud voisin avant stabilisation :
   Cas2a : le détecteur trouve que le nœud est en état de crash. Suite à celà, le nœud qui le remplace partage les jobs de ce nœud sur ses voisins collaborateurs, puis il lance la procédure de tolérance aux fautes du nœud défaillant.
   Cas 2b : le détecteur trouve que le nœud n'est pas en état de crash. Dans ce cas, il y a seulement une déconnexion et le nœud désigne un autre remplaçant.

5.6 Discussion

Nous avons proposé dans cette section, un modèle transformant une grille de calcul en un graphe coloré dynamique et nous avons coloré les sommets selon un

degré de tolérance appelé α. Cette coloration génère des sommets instables, stables et hyperstables suivant leurs nombres de voisin collaborateurs (qui sont en fait leurs substituts potentiels en cas de panne) et la phase de stabilisation donne la priorité aux sommets instables pour se stabiliser par le biais des sommets hyperstables. Le graphe converge toujours vers un état stable. La technique de tolérance aux fautes proposée consiste à distribuer les jobs du nœud défaillant sur ses voisins collaborateurs ou désigner un remplaçant capable de le remplacer en cas de panne.

6 Conclusion

Dans ce chapitre, nous avons proposé un modèle basé sur les graphes colorés dynamiques pour assurer la tolérance aux fautes dans les grilles de calcul. Nous avons exploité ce modèle pour tolérer les défaillances des nœuds d'une grille de calcul. Dans le premier modèle proposé, chaque nœud classifie tous les nœuds de la grille selon un critère de performance basé sur un ensemble de propriétés, les nœuds sont classés en nœuds identiques, plus performants et moins performants. Lors de l'occurrence d'une panne d'un nœud dans la grille, nous cherchons des substituts dans l'ensemble des nœuds identiques, puis dans les plus performants et enfin dans les moins performants. Ce modèle permet également de fournir des informations très importantes sur l'état général de la grille (espace disque disponible, charge des nœuds, etc.). Dans le deuxième modèle, nous avons proposé un modèle de tolérance aux fautes basé sur un degré de tolérance, permettant de transformer la grille en un graphe coloré dynamiques. Par la suite, nous avons proposé deux techniques de tolérance aux fautes dans les grilles de calcul, tout en tenant en compte des caractéristiques des grilles et en particulier surtout la dynamicité et le passage à l'échelle. Les deux modèles proposés associent à chaque sommet du graphe (nœud de la grille), un ensemble de voisins qui sont des substituts potentiels qui pourront être utilisés en cas de panne d'un noeud de la grille. Pour le premier modèle (tolérance par voisins collaborateurs), chaque nœud répartit ses jobs sur ses voisins collaborateurs en cas de défaillance et pour le deuxième modèle (tolérance par remplaçant), chaque nœud désigne un remplaçant qui sera chargé de toute sa tolérance.

Chapitre 5

Modèle fiable de tolérance aux fautes

1 Introduction

Une grille est un système distribué, composé d'un réseau d'ordinateurs hétérogènes, qui fournit des accès uniformes et indépendants à des organisations géographiquement distribuées [60]. Ce type de systèmes fournit un environnement hautement évolutif pour la résolution de problèmes à grande échelle. Cependant, comme un système distribué à grande échelle, les grilles sont sujettes à des défaillances [198]. Ces défaillances peuvent provenir de bogues logiciels, d'erreurs de manipulation humaine, de surcharges de performances, de congestion sévère, de défaillances de composants physiques et/ou logiciels, ou même d'opérations de maintenance. En outre, les catastrophes environnementales telles que les incendies, les inondations, les tremblements de terre, peuvent fermer des parties de systèmes de grille [83]. Dans ce contexte, la tolérance aux fautes apparaît comme l'attribut de survie des grilles de calcul pour atteindre la qualité de service attendue. La migration des processus est une capacité essentielle dans les environnements de grille car elle fournit un support pour la tolérance aux fautes, la maintenance du système à la demande, la gestion des ressources et l'équilibrage de charge. Cependant, la migration des tâches (job ou processus) entre les nœuds de la grille pourrait augmenter le coût global de traitement en raison des coûts en bande passante et de l'état du réseau. Il a été proposé, dans la littérature, comme un mécanisme de tolérance aux fautes proactive pour compléter la technique des points de reprise (*Check-point*) [212]. Par ailleur, plusieurs implémentations de l'environnement MPI ont fourni un appui pour la migration de processus, afin d'assurer la tolérance aux fautes de processus [217]. Dans [11], les auteurs présentent une technique décentralisée de tolérance aux fautes et d'équilibrage de charge, qui prend en compte l'architecture du réseau, l'hétérogénéité des ressources, le retard de communication, la bande passante du réseau, la disponibilité des ressources, l'imprévisibilité des ressources et les caractéristiques de chaque tâche. Ainsi, lors de la sélection d'un site, les ressources disponibles ne peuvent pas être le seul facteur pris en considération. D'autres facteurs doivent être pris en considération pour l'efficacité des sites, tels que l'historique de son niveau de contribution dans l'exécution des jobs, le taux des jobs achevés, etc. Une fonction

objectif a été proposée pour calculer l'efficacité de chaque site. Un ordonnanceur de grille utilise un seuil pour sélectionner le site approprié. Une réplication passive est utilisée pour tolérer les fautes. Sun et al. fournissent des systèmes tolérants aux fautes, comme FARS [123] et FENCE [191], pour augmenter la précision de la prédiction des fautes et d'améliorer la résilience du système à des défaillances par différents mécanismes de tolérance aux fautes, y compris la migration de processus. Ils modélisent également le coût de la migration et introduisent un mécanisme d'ordonnancement dynamique basé sur un système combinant les points de contrôle et la migration [114].
L'évaluation de la fiabilité d'un système est une étape clé dans la conception et l'analyse des systèmes. Parmi les propriétés et les mesures de la sûreté de fonctionnement, nous pouvons citer la fiabilité, la disponibilité, le temps moyen entre défaillances (*Mean Time Between Failures : MTBF*), le temps moyen avant défaillance (*Mean Time To Failure : MTTF*), le temps moyen de réparation (*Mean Time To Repair : MTTR*), le temps médian de réparation (*Median Time To Repair : MDTTR*), l'intensité de défaillance, etc. [112]. Au cours de ces dernières années, plusieurs études ont été réalisées sur les traces de défaillances des grands systèmes de grappes à grande échelle [180]. Ces efforts de recherche ont contribué à une meilleure compréhension des caractéristiques de défaillances, et contribuent à améliorer la fiabilité et la disponibilité des systèmes. Une étude sur l'analyse des données de défaillance a été rendue publique par un des plus grands sites de calcul de haute performance (*High Performance Computing : HPC*) [180]. Les données, relatives à cette étude, ont été recueillies au cours des 9 dernières années à *Los Alamos National Laboratory* et comprend 23000 défaillances enregistrées sur plus de 20 systèmes différents, pour la plupart de grandes grappes de nœuds SMP (symmetric shared memory multiprocessor) et NUMA (*Non Uniform Memory Architecture*). La plupart des charges de travail sont des simulations scientifiques de grande envergure de longue durée. Ces applications fonctionnent sur de longues périodes, souvent, un mois de calcul CPU intensif. À ce jour, c'est le plus grand ensemble de données de défaillances étudié dans la littérature, à la fois en termes de délai et du nombre de systèmes et de processeurs étudiés. Cette étude a permis de relever que le temps entre deux fautes est modélisé par une distribution de Weibull avec la diminution du taux de fautes. En se basant sur un modèle abstrait du système de gestion des ressources d'une grille (*Resource Management System : RMS*), ce chapitre présente une nouvelle approche de tolérance aux fautes dans les grilles de calcul. Quand un nœud tombe en panne, le module de tolérance aux fautes met en œuvre une nouvelle technique de migration de de jobs (tâches, processus), qui combine deux techniques : *Remote Node Fault Recovery* (RNFR) et *Local Node Fault Recovery* (LNFR), en sélectionnant le nœud le plus fiable parmi les voisins collaborateurs d'un nœud défaillant, afin de réduire le coût de la RNFR [165].

2 Migration des tâches pour la tolérance aux fautes

Plusieurs techniques de tolérance aux fautes ont été étudiées dans les systèmes de grille de calcul [104]. Dans la pratique, les techniques de points de reprise (checkpoint) [81] sont aujourd'hui largement utilisées pour permettre à une tâche (pro-

cessus, job, application) de reprendre son exécution à la suite d'une défaillance. Les premiers efforts basés sur les points de reprise (ou de contrôle) ou la migration de processus dans les grands environnements de cluster ont été signalés dans Légion [78], Cactus [76], et Condor [194]. Une image de tous les processus en cours d'exécution dans une application est capturée et utilisée pour reprendre une exécution après une défaillance en revenant au dernier point de contrôle. Dans ce cas, l'ensemble de l'application doit être arrêté et soumis à nouveau à l'ordonnanceur de tâches ou à un serveur de migration comme dans Cactus. Les principaux inconvénients de ce type d'approches sont le coût de migration. La nécessité d'une approche de la migration nomade pour l'exécution des tâches dans un environnement de grille a été initialement discutée dans [115]. Ainsi, Gridway [202] et GrADS [138] fournissent des outils assurant la migration des jobs pour les applications MPI sous le middleware Globus Toolkit [59] et des points de contrôle au niveau de l'application. Le processus d'ordonnancement dans GrADS se compose de deux modes de migration : une migration sur demande (si la dégradation des performances de l'application est inacceptable) et une migration opportuniste (si les ressources ont été libérées par des tâches récemment achevées). Une technique de migration orientée sur la performance pour les grilles de calcul, est décrite dans [202]. Elle tente d'améliorer le temps de réponse pour les applications individuelles. Le système surveille en permanence les applications et évalue le temps d'exécution restant de l'application, et les décisions de migration sont effectuées chaque fois que les applications ne font pas de progrès suffisants. Cependant, le modèle d'évaluation dépend des modèles de performance spécifiques à l'application et la surcharge provoquée par la migration y compris la lecture et l'écriture des points de contrôle est beaucoup plus élevée que le redémarrage d'une tâche. Puis, la migration des tâches est basée sur la fiabilité des services de grille dans [219]. De la même manière, le système Condor/OGRADE [108] est constitué d'un mécanisme de points de reprise pour les applications de PVM (*Parallel Virtual Machine*) et utilise Condor-G [194] pour l'ordonnancement des tâches. Toutefois, la tolérance aux fautes de l'infrastructure de service n'est pas prise en charge. Dans une approche holistique, Kandaswamy et al. [101] décrivent un mécanisme de tolérance aux fautes dans les workflows, en considérant les points de contrôle, la migration et la réplication. Dans le cadre d'applications MPI, l'optimisation du processus de checkpointing [212] et la migration des tâches/processeur [33] sont encore intensivement étudiées. Récemment, Suchang Guo et al., [83] ont introduit un mécanisme de LNFR dans les environnements de grille de calcul à l'encontre de RNFR. Dans LNFR, les modules de tolérance aux fautes sont implémentés dans les ressources de la grille. Cette approche offre la possibilité de reprendre rapidement l'exécution après une défaillance, et donne un coût de migration efficace. En outre, comme la reprise après une défaillance est gérée au niveau des ressources de la grille, les fournisseurs de ressources peuvent définir des contraintes personnalisées pour la tolérance aux fautes, ce qui facilite la mise en œuvre d'une gestion adaptative et distribuée de la tolérance aux fautes. Pour assurer la faisabilité du LNFR, des contraintes sur la durée de vie des tâches et le nombre d'exécutions sur un nœud ont été introduites, ouvrant la voie à la notion de fiabilité d'un service de grille. Dai et al. font usage d'une structure virtuelle [43]. Les nœuds qui représentent les ressources utilisées dans un calcul sont des nœuds virtuels (NV), qui sont reliés les uns aux autres par des liens virtuels (LV's). Les

ensembles de NV et LV, impliqués lors de l'exécution d'un service donné, forment un arbre de recouvrement. Ainsi, un arbre de recouvrement est défini comme un arbre qui relie les NV à d'autres nœuds via des liens, de sorte que ses sommets détiennent toutes les ressources nécessaires pour l'exécution du service de grille. La fiabilité du service de grille est déterminée par la fiabilité de ces arbres. Pour calculer la fiabilité des services de grille, les auteurs mettent en œuvre un algorithme de recherche d'arbre de recouvrement minimal [43]. Dans [42], Dai et al. présentent un modèle hiérarchique pour l'analyse et l'évaluation de la fiabilité du service de grille. Dans cette étude, l'architecture générale du système de service de grille est associée à un modèle hiérarchique. Les auteurs examinent divers types de défaillances, telles que des défaillances de blocage, les défaillances de délai d'attente, les pannes de réseau et les pannes d'exécution pour parvenir à une image complète de la fiabilité du service de la grille. Ils utilisent des modèles de Markov, de la théorie de file d'attente et de la théorie des graphes pour modéliser, évaluer et analyser la fiabilité du service de grille [42]. Ces travaux donnent une analyse intéressante sur la fiabilité du service de la grille, mais ils ne développent pas une technique de tolérance aux fautes.

3 Analyse de fiabilité pour un service de grille

L'Open Grid Services Architecture (OGSA) décrit l'architecture d'un environnement de grille de calcul orientée services à usage professionnel et scientifique [59]. Ainsi, une grille orientée services peut être considérée comme un serveur distribué à large échelle, et l'interaction entre les utilisateurs et la grille est basée sur un ensemble de demandes de services et de réponses. Quand une demande de service arrive au RMS, un service correspondant est lancé pour exécuter une certaine tâche sous le contrôle du RMS [83]. Cependant, dans un tel environnement à grande échelle, les services ne connaissent pas les nœuds et les ressources qui seront regroupés pour traiter la demande. Par conséquent, le RMS joue un rôle capital dans la gestion de l'ensemble des ressources partagées, l'affectation des tâches aux ressources demandées et la supervision de l'exécution des tâches [42]. La structure et les fonctions des RMS dans la grille ont été introduites en détail dans [109].

3.1 Modèle abstrait de RMS

Nous résumons l'architecture d'un RMS en trois couches de base (Figure 5.1) : une couche haute, une couche de matchmaking et une couche basse. La couche haute comprend l'application et les demandes de ressources ; la couche basse contient les informations sur les ressources de la grille et la surveillance des tâches lancées dans la grille ; enfin, la couche de matchmaking sert de lien entre la couche haute et la couche basse pour la recherche de la meilleure ressource pour chaque demande.
Nous allons mettre l'accent sur la couche basse, qui détient toutes les informations sur les ressources disponibles dans la grille et elle supervise l'état des ressources et l'exécution des tâches dans la grille. Ces composantes sont décrites ci-dessous :

1. **Diffusion et découverte de ressources :** les protocoles de diffusion et de découverte de ressources fournissent un moyen pour le RMS pour détermi-

ner l'état des ressources qui sont gérées par lui-même et par d'autres RMS qui interagissent avec lui. Le protocole de diffusion de ressources contient des informations sur les ressources disponibles ou un pointeur vers un serveur d'information. Le protocole de découverte de ressources fournit un mécanisme pour trouver les informations de la ressource demandée. Dans certaines architectures de RMS, aucune distinction n'est faite entre ces deux protocoles. Par exemple, un RMS pourrait utiliser un annuaire réplique qui contient des informations de ressources. Le protocole de diffusion de ressources peut être mis en œuvre en tant que protocole de réplication d'annuaire. La fonction de découverte de ressources consisterait à chercher dans l'annuaire, le plus proche. Alternativement, une grille pourrait maintenir un annuaire central où la diffusion de l'état des ressources est maintenu dans cet annuaire et la découverte consiste à interroger un annuaire central.

2. **Surveillance des ressources et des tâches :** le RMS est également chargé de nommer les ressources du système, surveiller l'état des tâches et le statut des ressources. Le RMS prend en compte l'utilisation des ressources et la mise à jour dynamique de leurs paramètres. Compte tenu des temps d'exécution et de l'intensité des pannes des nœuds et des liens de communication, qui peuvent être estimés par un système de surveillance du réseau, la fiabilité du service de la grille peut être facilement obtenue.
3. **Annuaire des états des ressources et des tâches :** Un RMS utilise une base pour suivre l'état des ressources et le statut des tâches, où il stocke les informations sur l'état des tâches et des ressources. Les modules de diffusion de découverte des ressources et des tâches mettent à jour périodiquement cette base.

3.2 Modèle de grille

Dans les grilles de calcul, une ressource est une entité réutilisable utilisée pour répondre à une demande de tâches ou de ressources. Dans ce chapitre, une ressource peut être une machine ou tout autre service qui est synthétisé en utilisant une combinaison de machines, réseaux et logiciels. Le concept d'organisation virtuelle (OV) d'une grille a été initialement présenté et discuté dans [62]. En fait, une OV se compose d'un ensemble de nœuds virtuels (NVs). Un NV est une unité générale intégrée dans la grille, qui peut exécuter des tâches ou partager des ressources, comme un ordinateur, un élément de traitement, un système embarqué, un produit numérique, etc. Nous considérons un NV comme un nœud qui accueille un ensemble de nœuds de travail (*work node*) identiques ; chaque nœud de travail possède son propre processeur, sa mémoire, son disque, etc. Les nœuds de travail à l'intérieur d'un nœud sont connectés par un réseau d'interconnexion rapide. Une tâche peut être affectée à n'importe quel nœud de travail [51]. Chaque nœud utilise un système de file d'attente pour exécuter ses taches, c'est à dire, quand une tâche est soumise à un nœud, elle est mise dans le système de file d'attente jusqu'à ce qu'elle puisse être soumise à un nœud de travail. Les nœuds sont reliés entre eux par des liens de réseau. Tous les nœuds en ligne ou les ressources sont reliés entre eux par des liens de communication (voir Figure 5.2). Toutefois, seul un sous ensemble de nœuds

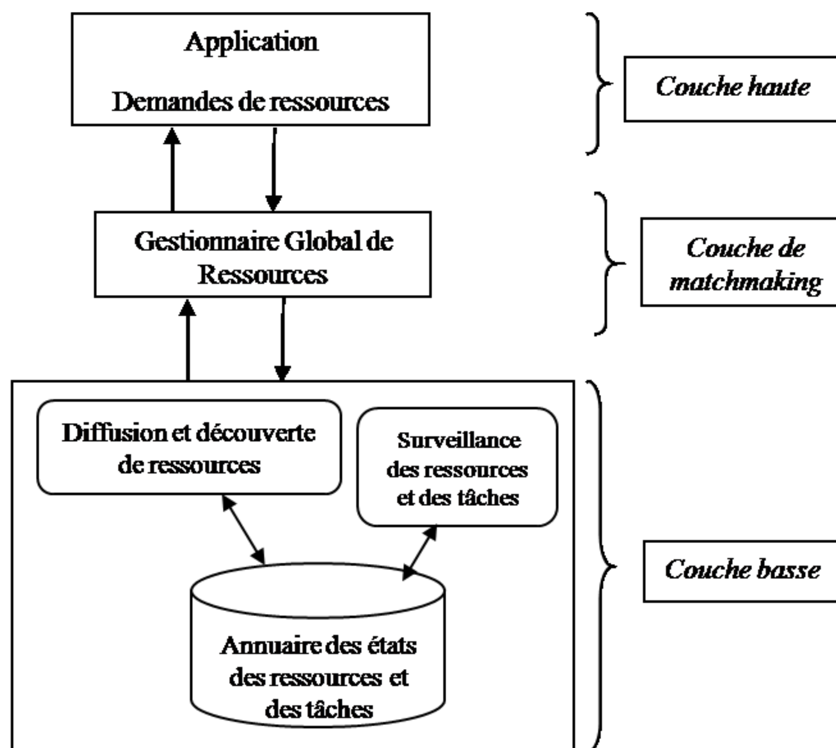

FIGURE 5.1 – Architecture en couches d'un RMS

ou de ressources sont affectés aux RMS pour l'exécution d'un service spécifique de la grille. Néanmoins, les RMS peuvent décider de sélectionner et attribuer plus de ressources que nécessaire pour améliorer la fiabilité d'un service de la grille.

3.3 Modélisation d'une grille de calcul par un graphe

Dans l'architecture d'un RMS (voir Figure 5.1), la couche basse contient le module "Diffusion et découverte des ressources" chargé de détecter de nouvelles ressources dans une grille. Nous supposons que chaque nœud envoie au RMS ses caractéristiques comme sa vitesse de traitement, sa mémoire, son espace disque, sa longueur de file d'attente, etc., ainsi qu'une liste de nœuds qui ont un lien direct avec lui et les caractéristiques des liens (par exemple, la valeur de la bande passante). Ces données sont stockées dans l'annuaire des états des ressources et des tâches. Sur la base de ces données, nous modélisons une grille comme un graphe non orienté $G = (X, U)$, où l'ensemble (des sommets) $X = \{x_1, x_2, ..., x_n\}$ représente les nœuds de la grille, et $U = \{u_1, u_2, ..., u_m\}$ représente les liens entre les nœuds. Nous attribuons à chaque lien $u_i \in U$ un poids non-négatif, pour représenter la valeur de la bande passante.
Le module de surveillance des ressources et des tâches reçoit périodiquement le nouveau statut des nœuds et des liens. Dans l'approche de la tolérance aux fautes proposée, quand un nœud tombe en panne, le RMS fait migrer ses tâches vers un nœud voisin. Nous appelons un nœud qui peut prendre en charge la migration des

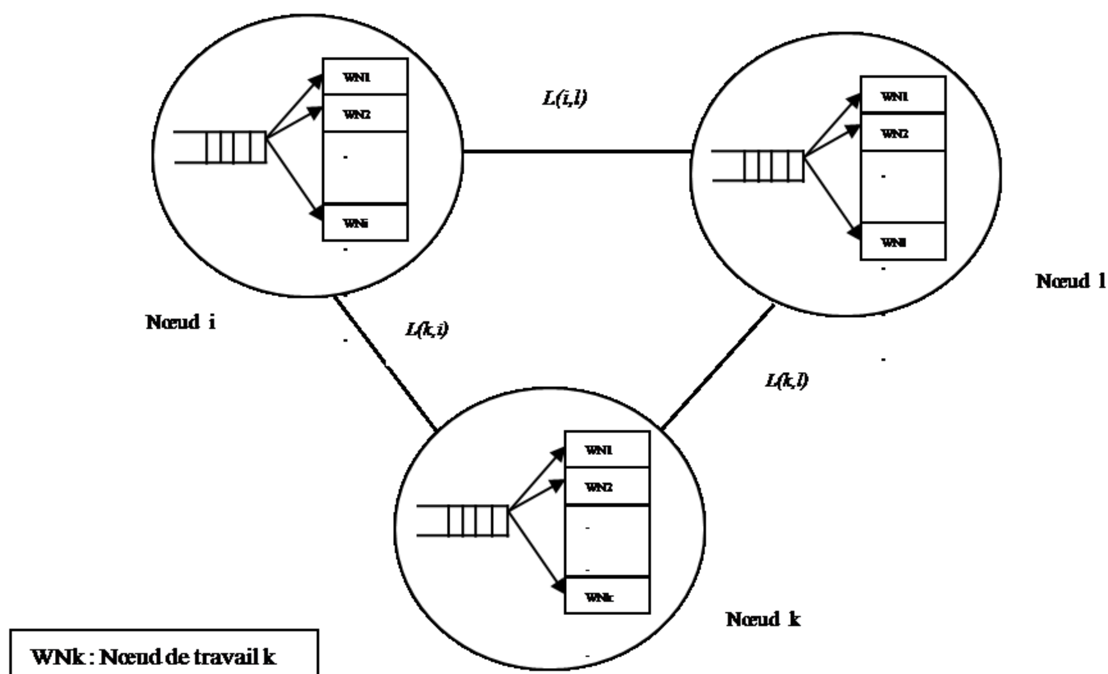

FIGURE 5.2 – Modèle de grille

tâches en provenance d'un nœud, un voisin collaborateur de ce nœud. L'acceptation de ces tâches est conditionnée par l'espace libre dans la file d'attente du voisin collaborateur. En pratique, chaque sommet du graphe qui tombe en panne dispose d'un ensemble de voisins collaborateurs.

3.4 Le temps médian de réparation (MDTTR)

Le temps médian de réparation est le temps d'arrêt durant lequel 50% de toutes les actions de maintenance peuvent être complétées. Nous nous sommes concentrés sur l'influence de MDTTR et temps moyen de réparation (MTTR) sur le processus de migration. Bien qu'il soit difficile de trouver des données de grande ampleur sur les défaillances dans les systèmes distribués à large échelle, nous avons identifié un ensemble de données pertinent du Laboratoire Los Alamos National [180]. Dans cette étude, les chercheurs de ce laboratoire ont constaté que le MDTTR des erreurs humaines varie de moins d'une heure, à cinq heures pour les défaillances dues à des problèmes environnementaux. Le MDTTR pour les autres catégories varie entre 30 minutes et deux heures. Le MDTTR dans toutes les autres pannes est proche d'une heure. Une observation importante est que le temps de réparation pour tous les types de défaillances est extrêmement variable, sauf pour les problèmes environnementaux. Par exemple, le MDTTR des pannes logicielles est d'environ 10 fois plus faible que le MTTR, et dans le cas de défaillances matérielles, il est quatre fois plus faible que la moyenne (voir tableau 5.1). Une autre analyse porte sur la façon dont les temps de réparation varient selon les systèmes. Le MDTTR varie entre 30 minutes et 6 heures mais le MTTR varie entre 2 heures et 4 jours [180]. Le MDTTR est toujours

inférieur au MTTR et ses valeurs sont un bon facteur dans le temps de réparation. Dans ce chapitre, nous proposons une stratégie de tolérance aux fautes basée sur une migration périodique ayant le MDTTR comme période.

	Inconnu	Humain	Environnement	Résau	Logiciel	Matériel	Total
MTTR (min)	398	163	572	247	369	342	355
MDTTR (min)	32	44	269	70	33	64	54

Table 5.1 – Temps de réparation en fonction de la cause de la panne [15]

3.5 Fiabilité des services de grille

Différents types de pannes peuvent se produire, comme l'insuffisance des ressources, perte de messages, bogue dans l'application, etc. Toutes ces pannes rendent un service de grille non fiable. Dans ce chapitre, nous nous concentrons sur les fautes crash des nœuds et les fautes de déconnexion. Nous définissons la fiabilité d'une tâche et la fiabilité d'un service de grille comme suit :
Définition 1 : La fiabilité d'une tâches (job ou processus) est définie comme la probabilité d'une exécution réussie de cette tâche sur une ressource dans une grille. L'exécution réussie comprend la transmission des données et de la tâche, son exécution sur la ressource et la transmission du résultat fourni par cette tâche.
Définition 2 : La fiabilité d'un service de grille est définie comme étant la probabilité que toutes les tâches impliquées dans le service soient exécutés avec succès.
Dans le reste de cette section, nous étudions en détail et discutons ces deux notions.

Hypothèses de travail :

1. Le RMS est entièrement fiable et le temps de traitement d'une demande de tâche par les RMS (réception de la demande, le temps d'attente, temps de matchmaking) est négligeable par rapport au temps de traitement d'une tâche.
2. Chaque nœud est capable d'exécuter n'importe quelle tâche quand il est disponible.
3. Les pannes des différents nœuds et des liens sont indépendantes.
4. Les pannes des nœuds et de liens peuvent être modélisées par un processus de Poisson [43].
5. Il n'y a aucune contrainte de précédence sur l'ordre d'exécution des tâches.
6. Chaque nœud de travail peut exécuter qu'une seule tâche à la fois.
7. Il n'y a aucune faute logicielle.

En résumé, les tâches d'un service de grille envoient leurs demandes de ressources au RMS. Ce dernier met ces demandes dans sa file d'attente. Ensuite, les demandes attendent dans la file d'attente pour une période de temps (appelée délai d'attente).

N^o	Symbole	Description
1	c_i	La complexité de calcul de la tâche T_i
2	s_k	La vitesse de traitement du nœud N_k
3	ζ_{ik}	Le temps de traitement de la tâche T_i sur le nœud N_k
4	λ_k	L'intensité de défaillence du nœud N_k
5	NR_{ik}	La fiabilité du nœud N_k durant l'exécution de la tâche T_i
6	R^h_{ik}	La fiabilité matérielle du nœud N_k durant l'exécution de la tâche T_i
7	R^c_{kl}	La fiabilité de communication entre le nœud N_k et le nœud N_l
8	D^R_{ik}	Le volume de données échangées entre le RMS et le nœud N_k pour exécuter la tâche T_i
9	DR^R_{ik}	Le volume de données de type résultat échangées entre le nœud N_k et le RMS après une exécution réussie de la tâche T_i
10	ϵ^R_k	Le temps de communication entre le RMS et le nœud N_k
11	T^c_{kl}	Le temps de communication entre le nœud N_k et le nœud N_l
12	S_{kl}	La vitesse moyenne de la liaison entre le noeud N_k et le noeud N_l
13	D^e_{kl}	Le volume de données échangées entre le nœud N_k et le nœud N_l
14	λ_{kl}	L'intensité de défaillance de la liaison de communication entre le nœud N_k et le nœud N_l
15	λ^R_k	L'intensité de défaillance de la liaison de communication entre le RMS et le nœud N_k
16	S_{Rk}	La vitesse moyenne de communication entre le RMS et le nœud N_k
17	WT_k	Le temps d'attente moyen dans la file d'attente du nœud N_k
18	$MDTTR_{fi}$	Le temps médian de réparation pour la faute fi
19	RS	La fiabilité d'un service de grille
20	TM^i_{kl}	Le temps nécessaire pour la migration de la tâche T_i du nœud N_k au nœud N_l

TABLE 5.2 – Liste des symboles

Dans le RMS, le service de matchmaking cherchent les ressources demandées parmi les ressources partagées disponibles dans la grille. Une fois les ressources trouvées, le service de matchmaking relie les tâches à leurs ressources séléctionnées. Par la suite, les tâches peuvent atteindre les ressources distantes et échangent des informations avec elles à travers les liens de communication.
L'exécution réussie d'une tâche doit passer par trois étapes :

1. Le RMS envoie la tâche et éventuellement les données au nœud sélectionné.
2. Le nœud exécute la tâche.
3. Le nœud renvoie les résultats au RMS.

Lorsque la tâche T_i est affectée au nœud N_k, le temps nécessaire pour l'exécution de la tâche T_i sur le nœud N_k est défini comme suit :

$$\zeta_{ik} = c_i/s_k \tag{5.1}$$

Le temps de communication T^c_{kl} entre le nœud N_k et le noeud N_l, peut être obtenu en divisant le volume des données échangées entre eux, noté D^e_{kl}, par la vitesse de la bande passante du lien, désigné par S_{kl} et calculé comme suit :

$$T^c_{kl} = D^e_{kl}/S_{kl} \tag{5.2}$$

A partir de l'hypothèse 4, l'apparition des fautes des nœuds et des liens peut être modélisée par un processus de Poisson (La fiabilité $R(t)$ d'un systéme, suivant la loi de Poisson, est la probabilité qu'il soit opérationel pour tout instant $t_i \in [0, t]$. $R(t) = e^{-\lambda t}$ où λ est une constante représentant l'intensité de défaillance du système). Cette hypothèse est justifiée par la phase opérationnelle dans laquelle le logiciel et le matériel ne sont pas modifiés, (i.e le code de la tâche n'est pas modifié après sa soumission et la tâche est exécutée sur la ressource où elle est affectée), de sorte que les intensités de défaillances sont des valeurs constantes. Si une faute se produit pendant l'exécution d'une tâche dans un nœud, elle sera considérée comme une tâche défaillante.
Ainsi, la fiabilité du matériel d'un noeud N_k est caclulée en fonction l'intensité de défaillance λ_k du noeud N_k pendant une période de temps comprenant le temps d'attente WT_k d'une tâche T_i et le temps de l'exécution ζ_{ik} de T_i dans le noeud N_k. Cette fiabilité est donnée par :

$$R^h_{ik} = e^{-\lambda_k(\zeta_{ik} + WT_k)} \tag{5.3}$$

Au cours de la communication entre le nœud N_k et le nœud N_l, en cas d'une faute dans le nœud N_k, le nœud N_l ou dans le lien entre eux, Cette communication est considérée comme défaillante. Ainsi, la probabilité de succès pour la communication entre les nœuds N_l et N_k est calculée en fonction des intensités de défaillance λ_k du noeud N_k, l'intensité de défaillance λ_l du noeud N_l et l'intensité de défaillance λ_{kl} du lien de communication entre les nœuds N_k et N_l pendant le temps T^c_{kl} de communication entre ces nœuds. Elle peut être exprimée par :

$$R^c_{ik} = e^{-(\lambda_k + \lambda_l + \lambda_{kl})T^c_{kl}} \tag{5.4}$$

La fiabilité de la tâche T_i s'exécutant sur le nœud N_k doit inclure la fiabilité des trois étapes mentionnées ci-dessus :

1. D_{ik}^{R} est la taille totale des données échangées entre le RMS et le nœud N_k pour exécuter la tâche T_i.
   En appliquant l'équation (5.4), où le RMS est fiable, donc l'intensité de défaillance sera λ_k du noeud N_k et λ_k^R celle de lien de communication entre le RMS et le noeud N_k. Le temps de communication sera le rapport du volume de données à transférer du RMS vers le noeud N_k sur la vitesse moyenne de communication entre eux, qui est égale à $\epsilon_k^R = D_{ik}^R/S_{Rk}$. Donc la fiabilité de la liaison de communication est :
$$R_{Rk}^{c} = e^{-(\lambda_k+\lambda_k^R)\epsilon_k^R} \tag{5.5}$$
2. La probabilité pour qu'un nœud N_k soit fiable pendant le temps d'exécution ζ_{ik} de la tâche T_i, est donnée par l'équation (5.3).
3. Pour simplifier l'analyse, nous utilisons la même liaison que celle utilisée dans la première étape. Par conséquent, la fiabilité de la transmission d'un résultat au RMS suit le même raisonnement de l'équation (5.5), les intensités de défaillance sont les mêmes et le temps de communication est ϵ_R^k est égal à DR_{ik}^R/S_{Rk} où DR_{ik}^R est le volume des résultats à transmettre et S_{Rk} la vitesse moyenne de communication entre le RMS et le noeud N_k. Donc, cette fiabilité est donnée par :
$$R_{kR}^{c} = e^{-(\lambda_k+\lambda_k^R)\epsilon_R^k} \tag{5.6}$$

Ainsi, la fiabilité de la tâche T_i exécuté sur le noeud N_k est égale à :

$$NR_{ik} = R_{Rk}^{c} R_{ik}^{h} R_{kR}^{c} = e^{\{-(\epsilon_k^R+\zeta_{ik}+WT_k+\epsilon_R^k)\lambda_k-(\epsilon_k^R+\epsilon_R^k)\lambda_k^R\}} \tag{5.7}$$

Nous supposons que les m tâches $T_1, T_2, ..., T_m$, qui compose un service de grille. Le service utilise n nœuds pour l'exécution de ses tâches ; l'échange d'informations entre ces nœuds se fait par les liens du réseau. Ces nœuds sont désignés par $N_1, N_2, ..., N_n$. Par conséquent, à partir de la définition 2, la fiabilité d'un service de grille est définie par l'équation suivante :

$$RS = \prod_{i=1}^{m} NR_{ik} \tag{5.8}$$

où $k \in D(T_i)$ designe le nœud qui exécute la tâche T_i
Ainsi, la fiabilité du service de grille peut être obtenue par l'intensité de la défaillance d'un noeud N_k et l'intensité des défaillances des liaisons de communication entre le RMS et n'importe quel nœud N_K, qui sont évaluées en permanence par le module de surveillance de la ressource. D'autres paramètres peuvent être calculés comme le temps de calcul (voir équation (5.1)), le temps de communication entre le RMS et le noeud N_k (voir équation (5.2)), la vitesse moyenne de communication entre le RMS et le nœud N_k et le temps de transfert de données avant et après l'exécution de la tâche T_i.

4 Technique de tolérance aux fautes

L'approche commune utilisée pour la tolérance aux fautes dans un environnement de grille de calcul est de mettre en œuvre des mécanismes RNFR [212]. Cependant, de

tels mécanismes sont coûteux lorsque les programmes sont complexes et nécessitent beaucoup de temps d'exécution et/ou quand un grand volume de données doit être transmis. Suchang Guo et al. [83] ont introduit un mécanisme de LNFR conçu pour les systèmes de grille qui minimise le coût total et en même temps maximise la fiabilité d'un service de grille tout en satisfaisant toutes les contraintes de ressources, mais le LNFR n'est pas pratique lorsque la faute est irrécupérable. Dans ce chapitre, nous proposons une technique de tolérance aux fautes basé à la fois sur le RNFR et le LNFR de manière complémentaire avec une migration périodique des tâches du nœud défaillant.
Le MDTTR varie considérablement selon la cause de la faute. Nous divisons le temps de réparation d'un nœud en fonction de son MDTTR (voir Figure 5.3).

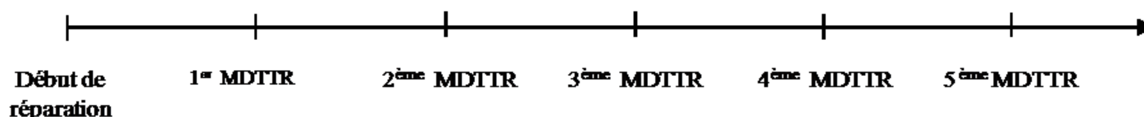

FIGURE 5.3 – Temps de répartition d'un nœud défaillant selon le MDTTR comme période de temps

4.1 Détection de fautes

Si le RMS a correctement alloué les tâches à leurs ressources requises, les tâches sont en mesure de se connecter et d'utiliser ces ressources à travers le réseau. Toutefois, les fautes de réseau peuvent se produire pendant cette période. De même les fautes crash des nœuds peuvent se produire lorsque les nœuds ont exécuté leurs tâches et retournent les résultats au RMS. Deux types de fautes sont tolérés :

1. Le nœud est actif mais il ne peut exécuter aucune tâche (ses nœuds de travail sont défaillants). La faute est détectée par le nœud lui-même.
2. Le nœud est défaillant et il ne peut pas être atteint par le RMS. Cette défaillance est détectée par le module de surveillance de ressources du RMS

Chaque nœud dans la grille accueille les tâches affectées à ses nœuds de travail et des tâches sont dans la file d'attente. Nous supposons que le RMS est capable de générer des points de contrôle, de manière transparente par rapport aux tâches en cours d'exécution dans les nœuds de travail. Ces points de contrôle seront utilisés en cas de défaillance de nœuds et la reprise après incident.

4.2 Stratégie de tolérance aux fautes

Supposons qu'une faute f_i est détectée dans le nœud N_k, alors qu'il accueille un ensemble de tâches mises dans sa file d'attente ou encore en cours d'exécution dans ses nœuds de travail. Les six types de fautes f_i sont présentés dans le tableau 5.1. Avant de décider de migrer une tâche T_i du nœud N_k vers le nœud N_l, nous devons

être sûrs que le temps d'exécution de T_i sur N_l, après la migration, sera inférieur à la durée d'exécution de T_i sur N_k après réparation ; sinon, nous utilisons le mécanisme LNFR. Le temps nécessaire pour la migration d'une tâche T_i de N_k vers N_l est la somme du temps de communication T^c_{kl} entre le nœud N_k et le nœud N_l, du temps d'attente dans la file d'attente de N_l et du temps d'exécution de T_i sur N_l. Ce temps global est calculé comme suit :

$$TM^i_{kl} = T^c_{kl} + WT_l + \zeta_{il} \tag{5.9}$$

Le temps nécessaire pour achever l'exécution de T_i après la réparation de N_k dans la première période de $MDTTR_{fi}$ est égal à $MDTTR_{fi}+\zeta_{ik}$. Par conséquent, nous ferons migrer T_i de N_k vers N_l, dans la première période de $MDTTR_{fi}$, seulement si : $TM^i_{kl}<MDTTR_{fi}+\zeta_{ik}$.
Généralement, nous décidons de migrer une tâche T_i du nœud défaillant N_k vers le nœud N_l dans la n^{ieme} période de $MDTTR_{fi}$, que si :

$$TM^i_{kl} < nMDTTR_{fi} + \zeta_{ik} \tag{5.10}$$

Par conséquent, la stratégie de tolérance aux fautes après la détection de la faute f_i dans le noeud N_k, qui exécute l'ensemble les tâches $\{T_i/i = 1...m\}$ est définie comme suit :

- **Étape 1 :** A chaque période $MDTTR_{fi}$ du temps de réparation du noeud N_k et pour chaque tâche T_i allouée au nœud N_k, nous cherchons l'ensemble des nœuds N_l parmi les voisins collaborateurs du nœud N_k qui satisfont l'équation (5.10).
- **Étape 2 :** Parmi tous les nœuds N_l trouvés, nous identifions le plus fiable (voir équation (5.7)) pour la migration de chaque tâche (RNFR) ; les tâches qui ne répondent pas à l'équation (5.10) ne seront pas migrées (LNFR).
- **Étape 3 :** Les étapes 1 et 2 sont répétées après chaque période $MDTTR_{fi}$.
- **Étape 4 :** Le processus se termine lorsque le nœud N_k est réparé ou s'il ne reste plus de tâches à faire migrer.

La recherche du nœud le plus fiable dépend du type de faute détectée. Dans le cas où les nœuds de travail sont défectueux, le nœud fait migrer directement les tâches aux voisins collaborateurs (voir section 4.3), tandis que dans le cas où le nœud est défaillant, le RMS est responsable de chercher le nœud le plus fiable (voir section 4.4).

4.3 Tolérance aux fautes d'un nœud de travail

Supposons que le nœud N_k veut faire migrer la tâche T_i vers l'un de ses voisins collaborateurs. Cette migration consiste à trouver les voisins collaborateurs les plus fiables parmi les nœuds de travail gérés par ce nœud, tel que :

$$Migr(T_i, N_k, N_l) = Max\{NR_{ikl}\} \tag{5.11}$$

NR_{ikl} est la fiabilité de la migration de la tâche T_i du du nœud N_k vers le nœud N_l. Après avoir reçu la tâche et les données, N_l exécute la tâche T_i, il renvoie les résultats au RMS. Donc NR_{ikl} est calculée à partir de R^c_{kl}, qui représente la fiabilité

de communication entre le nœud N_k et le nœud N_l, et R^h_{il}, la fiabilité matérielle du nœud N_l pendant l'exécution de la tâche T_i et R^c_{lR}, la fiabilité de communication des résultats au RMS. Par conséquent, en utilisant les équations (5.3) et (5.4), nous pouvons calculer NR_{ikl} comme suit :
$NR_{ikl} = R^c_{kl} R^h_{il} R^c_{lR} = e^{-(\lambda_k+\lambda_l+\lambda_k l)T^c_{kl}}.e^{-\lambda_l(\zeta_{ik}+WT_l)}.e^{(-\lambda_l+\lambda^R_l)\epsilon^l_R}$

$$NR_{ikl} = e^{-\lambda_k T^c_{kl} - (T^c_{kl}+\zeta_{il}+WT_l+\epsilon^l_R)\lambda_l - \lambda_{kl}T^c_{kl} - \lambda^R_l \epsilon^l_R} \qquad (5.12)$$

4.4 Tolérance aux fautes du nœud

Dans ce cas, le nœud (différent d'un nœud de travail) n'est plus opérationnel. Le RMS doit réaffecter toutes les tâches qui sont en cours d'exécution ou en attente aux voisins collaborateurs du nœud défaillant. En pratique, le RMS recherche, pour chaque tâche, les voisins collaborateurs les plus fiables. Supposons que le noeud N_k est défaillant et le RMS veut réaffecter la tâche T_i au voisin collaborateur N_l. A partir de l'équation (5.7), le nœud le plus fiable pour l'exécution de la tâche T_i est le noeud N_l qui vérifie l'équation suivante :

$$Migr(T_i, RMS, N_l) = Max\{NR_{il}\} \qquad (5.13)$$

5 Illustration numérique

Nous donnons une illustration numérique de notre modèle de fiabilité du service de la grille. Supposons un service composé de cinq tâches qui peuvent être affectées à 10 nœuds. Les informations relatives aux nœuds de la grille et les liaisons de communication sont montrées dans le Tableau 5.3, et les attributs des cinq tâches sont présentés dans le Tableau 5.4.

Nous avons utilisé les mêmes valeurs d'attributs de nœuds et des tâches comme ceux présentés dans [83], afin d'être en mesure de comparer les résultats. La fiabilité de service de grille dépend non seulement de son temps d'exécution mais aussi du volume de données échangées entre le RMS et les nœuds qui exécutent ce service. Lorsque le volume des données échangées entre les nœuds et le RMS augmente, la valeur de la fiabilité du service de grille est réduite (voir Figure 5.4). Pour un volume de données de 0.01 Go le $RS = 0.9926$ tandis que pour un volume de données de 200 Go le $RS = 0.0643$. Nous pouvons maintenant comparer nos résultats avec les résultats présentés dans [83] pour une tolérance aux fautes, i.e. , $x = 1$, ce qui implique qu'il n'y a aucune défaillance et pour un volume de données de 0.012 Go. Les auteurs de [83] trouvent une fiabilité du service $R = 0.9767$, sans contraintes sur le temps de vie des tâches et sur le nombre de recouvrements effectués. Pour le même volume de données, nous avons trouvé $RS = 0.9916$. Les auteurs de [83] n'ont pas donné une analyse sur l'influence du volume de données échangées sur la fiabilité du service, mais il est clair dans leurs formules que cette influence est négligeable.

Dans la Figure 5.5, nous analysons l'influence du temps d'exécution sur la fiabilité du service. Si le temps d'exécution des tâches est long, la fiabilité du service de grille diminue. En effet, pour $\zeta_{ik} = 500$ sec., $RS = 0.9916$ et pour $\zeta_{ik} = 100000$

Nœud k	S_k	λ_k	WT_k	λ_k^R	S_{Rk}
1	10	0.1	3	0.1	2
2	20	1.8	4	0.2	3
3	15	1.2	10	0.3	4
4	25	1.5	30	0.4	5
5	12	1.9	12	0.1	6
6	20	0.8	5	0.2	6
7	13	0.9	6	0.3	3
8	10	1.2	24	0.1	4
9	29	1.2	15	0.2	3
10	32	0.8	23	0.3	4

TABLE 5.3 – Attributs des nœuds et des liens

Tâche i	c_i (Gops)	D_{ik}^R	DR_{ik}^R
T1	6	12	15
T2	8	16	20
T3	7	14	17
T4	10	20	25
T5	8.5	17	22

TABLE 5.4 – Attributs des tâches

sec (27h 46mn 40s), $RS = 0.2859$. Nous pouvons noter que lorsque la tâche a une complexité de calcul élevée, sa fiabilité est considérablement réduite, ce qui nécessite le développement de techniques de tolérance aux fautes. Nous pouvons comparer nos résultats avec les résultats présentés dans [82]. Pour $x = 1$, et un temps d'exécution $= 2000$ sec., la fiabilité du service est $R = 0.8892$, sans contraintes sur le temps de vie des tâches et sur le nombre de recouvrements effectués et pour une récupération après défaillance, i.e. , $x = 0$, et un temps d'exécution $= 2000$ sec., $R = 0.0832$. Pour les mêmes conditions, nous avons trouvé $RS = 0.9732$. Dans les Figures 5.4 et 5.5, nous pouvons observer l'influence du volume de données échangées et de la complexité de calcul des tâches sur la fiabilité des services de grille.

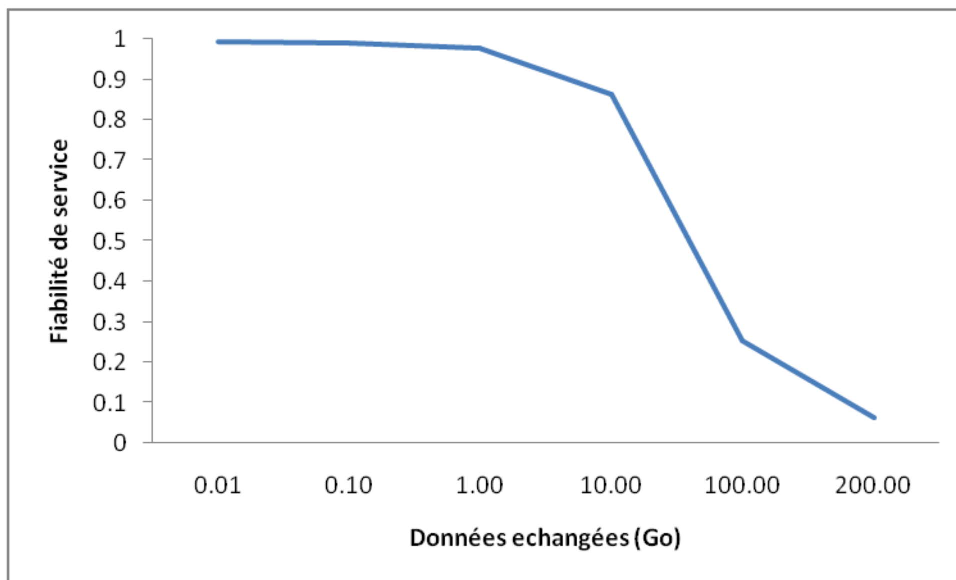

FIGURE 5.4 – Influence du volume de données échangé sur la fiabilité du service de grille

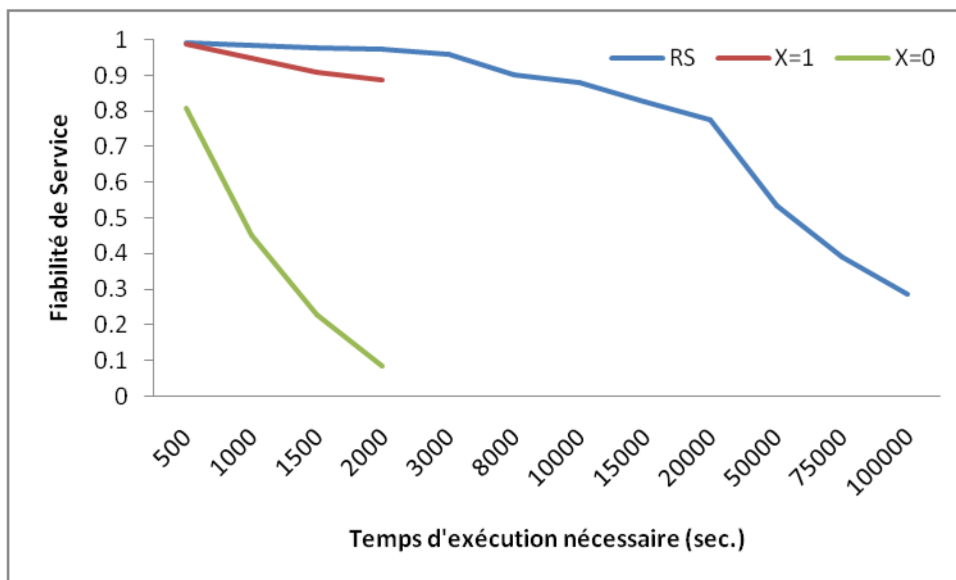

FIGURE 5.5 – Influence du temps d'exécution du processus sur la fiabilité du service de grille

6 Conclusion

L'approche fondamentale dans les recherches sur la tolérance aux fautes dans les grilles de calcul est le mécanisme RNFR. Le LNFR est proposé pour réduire la charge du RNFR. Le LNFR pourrait être plus utile que le RNFR pour redémarrer l'exécution d'une tâche sur le nœud défaillant une fois ce nœud est réparé. Par conséquent, le LNFR peut être utilisé avec le RNFR de manière complémentaire

pour atteindre la tolérance aux fautes dans un environnement de grille. Dans ce chapitre, nous avons proposé une technique de tolérance aux fautes qui combine les deux approches RNFR et LNFR sur la base d'une migration périodique. Dans ce modèle, nous faisons une abstraction de l'architecture de RMS en trois couches de base (couche haute, couche de matchmaking et couche basse). Basé sur cette architecture, un nouveau modèle de fiabilité du service de grille a été présenté. Ce modèle tient en compte tout le processus d'exécution d'une tâche, dès la soumission jusqu'à la récupération des résultats. La technique de tolérance aux fautes met en œuvre une nouvelle technique de migration, qui combine le RNFR et le LNFR, en sélectionnant le nœud le plus fiable parmi l'ensemble des collaborateurs voisins. Le processus de migration est périodique, ayant le MDTTR du nœud défaillant comme période, la migration continue jusqu'à la réparation du nœud défaillant ou la migration de toutes ses tâches.

Chapitre 6

Expérimentations

1 Introduction

Dans ce chapitre, nous allons présenter la mise en œuvre des modèles définis dans les chapitres 3, 4 et 5. Le modèle hiérarchique dynamique est validé sous Globus Toolkit 4.0.1, en exploitant les PC's disponibles à l'Université de Mascara, ce qui nous a permis de créer un Dektop Grid de 15 nœuds. En ce qui concerne le modèle de la grille mobile, les modèles décentralisés basés sur les graphes colorés dynamiques et le modèle fiable de tolèrance aux fautes, nous avons développé, pour chaque modèle, un simulateur pour valider les modèles correspondants.

2 Environnement d'évaluation

Comme environnement de test du modèle hiérarchique dynamique, nous avons utilisé Globus Toolkit [59]. Globus est un projet open source visant à créer des logiciels et des outils nécessaires pour la conception et la mise en œuvre de grilles de calcul. Il est principalement développé aux Etats-Unis dans l'Argonne National laboratory par l'équipe d'Ian Foster. Le travail sur Globus a commencé en 1997 et le projet est toujours actif. Le " Globus Toolkit " est formé d'un ensemble de composants. Son architecture modulaire permet d'apporter les modifications et les améliorations d'une manière rapide et efficace. Globus est devenu le standard utilisé dans les projets grilles de calcul. Ainsi, de nombreuses entreprises l'ont adopté pour servir comme base de leurs produits commerciaux en terme de sécurité, de services d'information, de gestion des communications, de gestion des ressources et de traitement des données.
Fonctionnalités de Globus : Globus fournit les fonctionnalités et les services de base nécessaires pour la construction de grilles de calcul. Ainsi, nous trouvons des services et des mécanismes tels que la sécurité, la localisation et la gestion des ressources, la communication, etc. [55].
Il est composé d'un ensemble de modules ayant chacun une interface, afin que les

services de niveau supérieur puissent les invoquer pour développer leurs propres modules.
Parmi ces modules, nous trouvons :

- **Localisation et allocation des ressources :** Ce composant permet aux applications d'exprimer leurs besoins en ressources et il fournit les mécanismes permettant d'identifier les ressources adéquates.
- **Communications :** Ce composant donne la possibilité aux différentes applications de communiquer entre elles. Un certain nombre de paradigmes de communication sont fournis, comme communication par messages, mémoire distribuée, appel de procédure à distance, etc.
- **Informations sur les ressources :** Ce composant permet d'obtenir des informations sur l'état et la structure globale du système du point de vue ressources.
- **Mécanismes de sécurité :** Ce composant fournit les mécanismes d'authentification et d'autorisation des utilisateurs.
- **Accès aux données :** Ce composant offre un accès consistant aux données stockées dans des fichiers et des bases de données (Voir Tableau 6.1).

Service	Nom	Description
Gestion de ressources	GRAM	Allocation des ressources et gestion des processus
Communications	Nexus	Services de communication unicast et multicast
Sécurité	GSI	Authentification et autorisation
Informations d'état	MDS	Informations sur la structure et l'état de la grille

TABLE 6.1 – Différents services de Globus

3 Modèle hiérarchique dynamique

3.1 Implémentation

Nous avons déployé ce modèle sur des machines Pentium4 sous Linux Fedora 6. Il a été développée en JAVA (JDK), PostgreSQL-8.0.4 sous le Middleware Globus Toolkit GT4.0.1. Notre application, appelé *Dynamic Hierarchical Model Fault Tolerance for the Grid (DHM-FTGrid)*, est un service de grille qui fournit des mécanismes de tolérance aux fautes basé sur deux techniques : la distribution et le remplacement.

3.2 Architecture de service de DHM-FTGrid

La Figure 6.1 illustre les composants principaux du service DHM-FTGrid :

a Gestionnaire de jobs : Les fonctions principales de ce composant est la soumission et la surveillance des jobs.

b Gestionnaire de fautes : Il gère la tolérance aux fautes dans la grille.

c Détecteur de fautes : Ce composant détecte l'existence d'une faute dans l'arborescence correspondante à une grille..

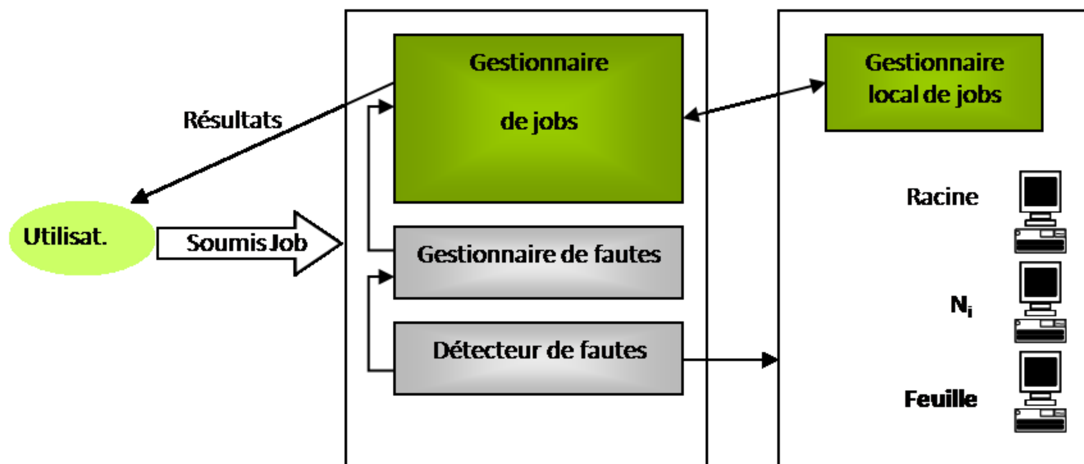

FIGURE 6.1 – Architecture du service DHM-FTGrid

3.3 Architecture de la grille de test

Notre modèle a été expérimenté sur une architecture hiérarchique (grille) en utilisant l'arborescence (8/4/2), qui répartit 15 postes en 4 niveaux (voir Figure 6.2) :

- 8 postes pour le niveau N_0 (niveau feuille).
- 4 postes pour le niveau intermédiaire N_1.,
- 2 postes pour le niveau intermédiaire N_2 avec un poste comme racine dupliquée de la grille.
- Un poste comme racine de la grille.

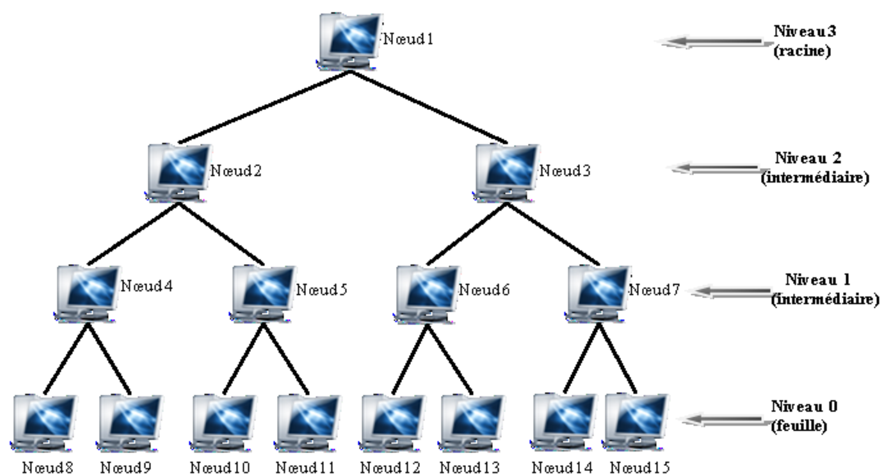

FIGURE 6.2 – Archtecture de la grille de test 8/4/2

3.4 Expérimentations

Le but de la série d'expérimentations, présentée dans cette section, est de mesurer les performances de notre modèle sur les deux schémas de grille. Nous nous intéressons à deux critères de performance. Le premier est le niveau de tolérance qui définit à quel niveau dans l'arborescence, la faute a été tolérée ? Cela, nous permettra de contrôler la propagation de la tolérance aux fautes du niveau où elle est détectée jusqu'au niveau où le DHM-FTGrid arrive à tolérer cette faute. Le deuxième critère, s'intéresse au taux d'utilisation des deux techniques de tolérance aux fautes en concurrence, la distribution et le remplacement. Cela permettra d'évaluer le coût de la tolérance, car la méthode de remplacement consiste à remplacer un nœud défaillant par une feuille, ce qui a pour conséquence de réduire le coût de tolérance aux fautes. Par contre, la méthode de distribution est plus coûteuse, car elle nécessite l'intervention de plusieurs nœuds et plus d'opérations de mise à jour. Durant les d'expérimentations sur la grille de test 8/4/2, nous avons augmenté le nombre de jobs de 5 à 60 par pas de 5, où chaque feuille possède une file d'attente de 5 jobs.
Les résultats obtenus, nous ont permis de relever les constatations suivantes (voir Figures 6.3 et 6.4) :

- Les niveaux de tolérance sont liés aux nombres de jobs dans la grille.
- Les techniques de tolérance (distribution et remplacement) sont correlées aux nombres de niveaux et de fils.
- Dans nos expérimentations, nous avons utilisé beaucoup plus du remplacement que de distribution à cause du nombre réduit de fils.

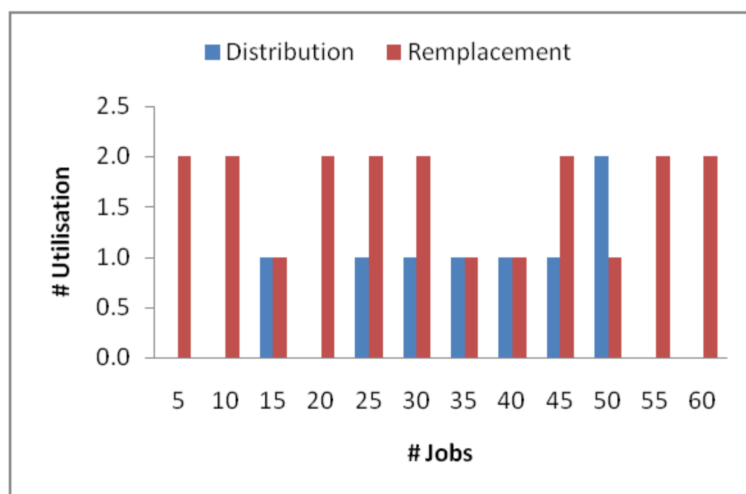

FIGURE 6.3 – Tolérance aux fautes par distribution et remplacement

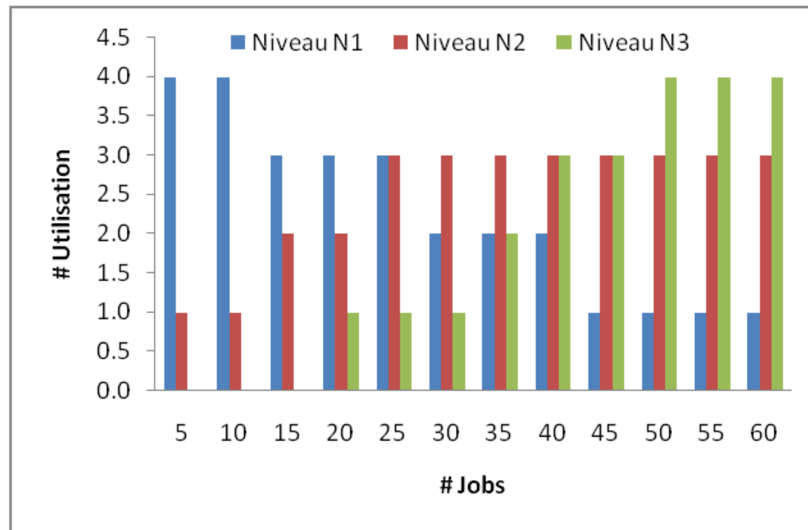

FIGURE 6.4 – Différents niveaux de tolérance

4 Modèle multi-arborescent pour une grille mobile tolérante aux fautes

4.1 Modèle de simulation

Pour valider notre modèle, nous avons développé un simulateur que nous avons appelé *Simobgrid (Simulator for mobile grid).*
Simobgrid est construit autour de cinq modules exploitant une base de données

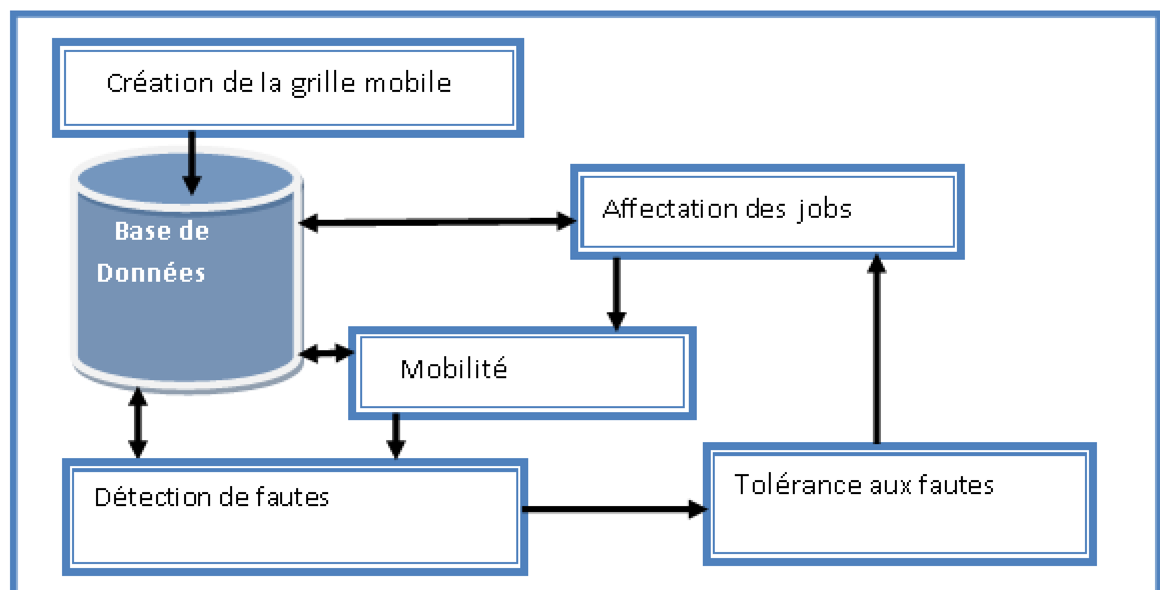

FIGURE 6.5 – Architecture de Simobgrid

pour enregistrer l'état de la grille mobile (voir Figure 6.5).

1. **Création de la grille mobile :** Avant de lancer la simulation, il faut spécifier la superficie de l'environnement de la grille mobile, les ressources de la partie filaire, les ressources mobiles et le nombre de points d'accès. Ce module est chargé de construire l'ensemble des arborescences de la grille mobile en définissant les gestionnaires des groupes à partir des ressources de la grille fixe et l'ensemble des dispositifs mobiles détectés autour des points d'accès en spécifiant les INT, $INTD$ et SUB (voir Figure 6.6).

2. **Affectation des jobs :** L'utilisateur définit le nombre de jobs au niveau de GM, INT et SUB. Simobgrid distribue les jobs à partir du GM équitablement sur ses $INT's$, qui à leurs tours, les distribuent vers les SUB. Chaque job a un temps d'exécution aléatoire permettant d'annoncer sa terminaison.

3. **Mobilité :** L'utilisateur définit un pourcentage de mobilité et Simobgrid sélectionne l'ensemble des SUB et INT à déplacer avec des valeurs de déplacements aléatoires. Cette mobilité change l'état de la grille mobile, et ces changements peuvent être des changements de groupe ou des déconnexions.

4. **Détection de fautes :** Ce module consulte en permanence les SUB de la grille mobile pour détecter les fautes, qui peuvent être des fautes de déconnexion ou de changement de groupe, de déconnexion globale, de déconnexion d'un INT, d'une dégradation de QoS ou la détection d'un nouveau SUB qui s'intègre dans la grille. Une fois la faute détectée, il appelle le service de tolérance aux fautes.

5. **Tolérance aux fautes :** Ce module lance le processus de tolérance aux fautes suivant le type de faute détectée, puis il informe le service d'affectation des jobs pour la prise en compte du nouvel état de la grille.

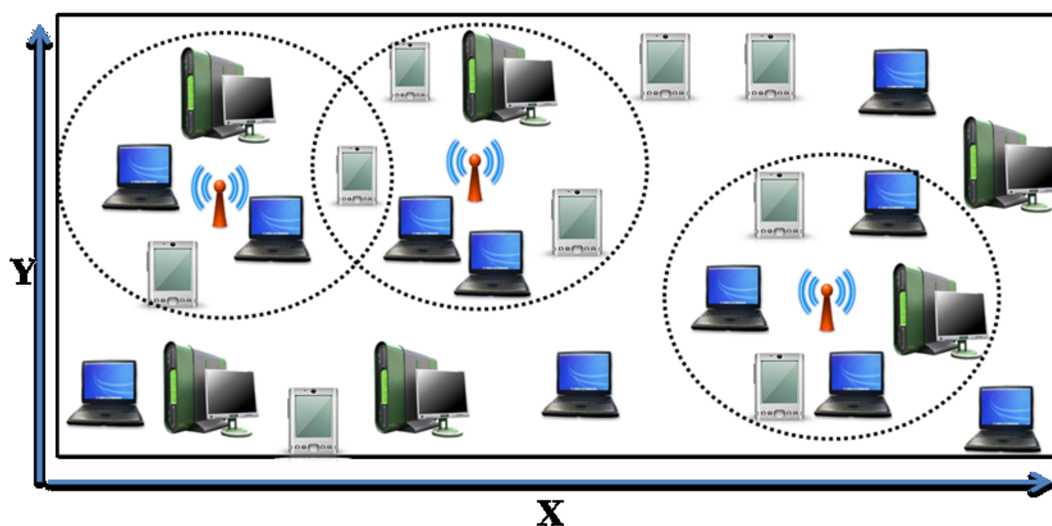

FIGURE 6.6 – Environnement de simulation

4.2 Résultats et interprétations

Les expérimentations ont été effectuées dans les conditions suivantes : le rayon de couverture est de 300 mètres, les files d'attente peuvent contenir 500 jobs par gestionnaire, 200 jobs par interlocuteur et 20 jobs par subordonné. La génération des fautes se fera dans un environnement où le taux de mobilité est égal à 40 %. Ces expérimentations ont été réalisées sur un Pentium 4 avec un processeur de 2.8 Ghz, d'une RAM de 4 Go et avec un disque dur de 80 Go.

4.2.1 Détection des fautes des subordonnés

Nous nous sommes intéressés aux types de fautes des subordonnés à cause de leur nombre dans la grille mobile et à cause de leur rôle (exécution des jobs). Nous avons constaté que les fautes les plus fréquentes sont la dégradation de la QoS et le changement de GM. Au deuxième niveau, surviennent les fautes liées aux changements des INT, la déconnexion totale et enfin les fautes de type crash (voir Tableau 6.2).

# SUB	Déconnex. totale	Change INT	Change GM	Faute crash	Faute Qos de SUB	Total des fautes	% Fautes détect.
776	77	30	126	18	105	356	45.88
2573	67	95	540	37	342	1081	42.01
4625	125	88	1114	38	644	2009	43.44
7168	492	170	1853	77	934	3526	49.19
9966	726	98	1943	452	1322	4541	45.56
10590	749	167	2435	271	1399	5021	47.41
21000	2500	460	3700	320	852	7832	37.30
29000	930	3120	4870	7800	1500	18220	62.83
45000	7900	4320	8700	9100	3700	33720	74.93
58000	5520	9200	10200	10500	6800	41920	72.28

Table 6.2 – Résultats relatifs à la détection des fautes des subordonnés

4.2.2 Tolérance aux fautes des INT

Le Tableau 6.3 représente les différents types de fautes pour les interlocuteurs (INT), et leurs taux de tolérance. Les INT qui changent de groupes viennent en premier lieu, puis les fautes de type crash, les dégradations de la QoS et enfin les fautes de déconnexion. Le taux des fautes des INT est de 48.44% par rapport à

de l'ensemble des INT dans l'environnement de la grille. *Simobgrid* a pu tolérer directement 77.61% des fautes détectés et les autres sont mises en attente jusqu'à la disponibilité des ressources.

# INT	Change groupe	Faute crash	Déconex. totale	Faute QoS	% Faute détect.	# INT tolér.	% Fautes tolér.
6	1	0	1	0	33.33	1	50.00
11	3	0	0	1	36.36	3	75.00
16	5	0	3	3	56.25	5	55.56
22	1	0	1	1	13.64	2	66.67
25	9	3	3	0	60.00	15	100.00
38	11	8	0	2	55.26	20	95.24
42	15	6	1	3	59.52	24	96.00

Table 6.3 – Tolérance aux fautes des interlocuteurs

5 Modèle des substituts

5.1 Evaluation et performances

Dans cette section, nous allons évaluer les performances de notre modèle à travers une simulation axée sur des événements discrets, en utilisant Graphstream [126] et Simgrid [31]. GraphStream est une bibliothèque Java dont le but est de créer et de manipuler des graphes dynamiques et de les utiliser dans les simulations. Simgrid est un outil de simulation qui fournit des fonctionnalités de base pour les différents types de ressources hétérogènes, les services et les types d'application. Toutes nos expérimentations ont été effectuées sur un Dual-Core Intel avec un processeur de 3.06 Ghz et une RAM de 4Go.

5.2 Paramètres observés

Nous observons pour chaque nœud de la grille, 3 classes de paramètres :

- C1 : Paramètres des nœuds.
- C2 : Paramètres de communication.
- C3 : Paramètres liés à la QoS (voir Tableau 6.4).

Classes et attributs	Type	Description	Intervalle de performance
C1 : *Paramètres d'un nœud*			
Mémoire	S		2Go-4Go
Espace disque	D		100Go-500Go
Vitesse CPU	S		2Ghz-3Ghz
Charge CPU	D		0.5-1.0
C2 : *Paramètres de communication*			
Bande passante	S	Fixe	100Mbps, 1Gbps
Latence	D		15ms-20ms
C3 : *Paramètres liés à la QoS*			
Stabilité	D	0 : très instable ; 1 : instable 2 : stable ; 3 : très stable	
Degré de déconnexion	D	0 : très instable ; 1 : instable 2 : stable ; 3 : très stable	
MTBF	D	Mean Time Between Failures	6 heures - 12 heures
MTTR	D	Mean Time To Repair	20 min- 30 min

TABLE 6.4 – Propriétés des nœuds de la grille (S : Statique, D : Dynamique)

5.3 Résultats relatifs au modèle basé sur un graphe coloré dynamique

Dans cette section, nous présentons les résultats obtenus par Graphstream. Nous allons mesurer son comportement en régime transitoire (le nombre de substituts), l'influence du taux de la dynamicité sur la structure du graphe et le changement des classes des substituts. Chaque série de simulation est effectuée sur un graphe non orienté, généré de manière aléatoire comme suit : étant donné un ensemble de nœuds et un nombre d'arêtes ajoutés entre les paires de nœuds de manière aléatoire, les poids des arêtes sont sélectionnés au hasard.

5.3.1 Classes des substituts

Nous faisons varier le nombre de nœuds de 100 jusqu'à 900 par pas de 100, nous ajoutons les arcs, nous créons un graphe pour chaque cas et nous calculons les substituts pour chaque sommet. La moyenne des substituts les plus performants est légèrement supérieure aux substituts moins performants, alors que les substituts identiques sont toujours inférieurs (voir Figure 6.7). Le taux des substituts identiques varie entre 15% et 20%, les substituts plus performants et moins performants se situent entre 40% et 45% avec une légère augmentation des plus performants (1% à 3%).

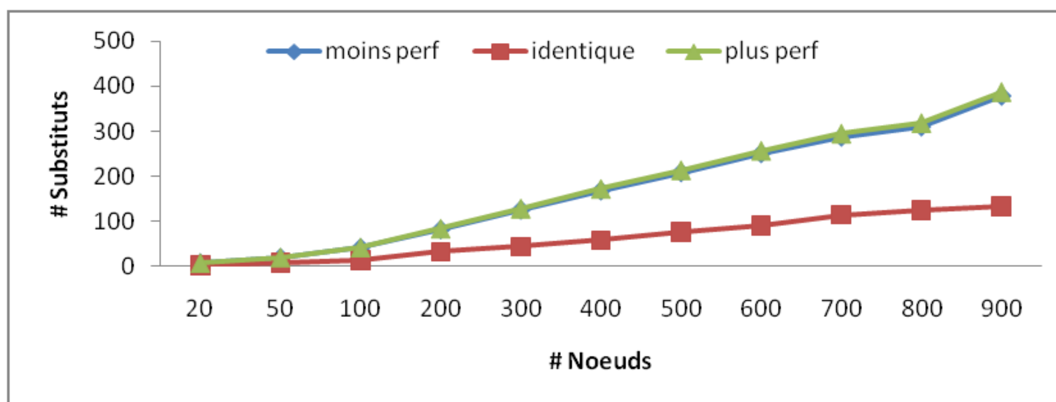

FIGURE 6.7 – Nombre des différents substituts (moins performants, identiques et plus performants) selon le nombre de nœuds

5.3.2 Taux de dynamicité

La dynamicité du graphe est caractérisée par un taux indiquant le nombre de nœuds à ajouter ou à supprimer du graphe. Nous avons défini deux types de dynamicité : (i) une dynamicité discontinue, où le graphe subit une seule dynamicité avec un taux précis. (ce taux varie de 10% à 100%) ; (ii) une dynamicité continue, dans laquelle, le graphe subit une suite de dynamicité qui commence par 10% pour atteindre 100%. Nous avons généré un graphe avec 100 sommets et 400 arêtes. Après une dynamicité continue et discontinue par pas de 10% (voir Figure 6.8), nous avons constaté que le nombre de substituts, dans la dynamicité continue, est plus important que dans la dynamicité discontinue, mais la distribution des classes des substituts (identique, moins performant et plus performant) reste la même.

5.3.3 Transfert du vecteur d'état

Maintenant, nous allons estimer le temps de transfert du vecteur d'état. Pour un graphe de n sommets, chaque sommet x a un vecteur d'état $V_t(x) = [va_1, va_2, ..., va_k]$ représentant les valeurs de ses attributs à l'instant t (voir Tableau 6.4). Supposons que la taille moyenne de chaque vecteur $V_t(x)$ est α Mo, et que la valeur moyenne de la bande passante est de β Mo/sec. Ainsi, le temps de transfert de $V_t(x)$ entre

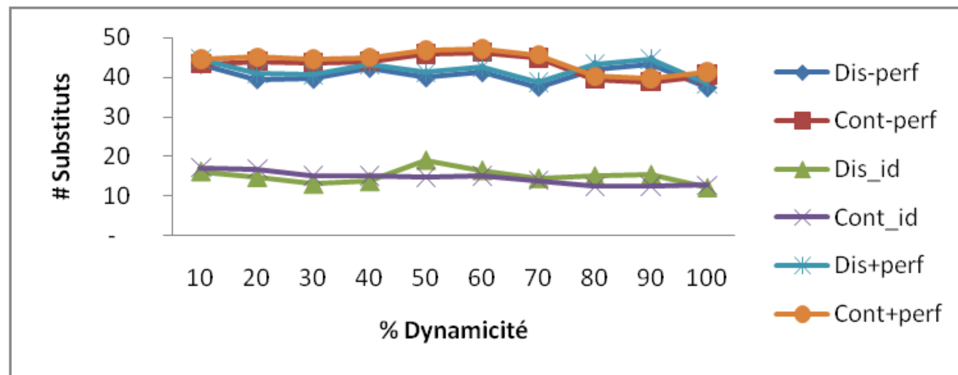

FIGURE 6.8 – Variation des type de substituts pour une dynamicité discontinue et continue

deux sommets est α/β sec. Le nombre de messages émis de tous les sommets vers tous les autres sommets dans un graphe complet est égal à $n(n-1)$ et le temps de transfert de tous les vecteurs d'états $V_t(x)$ est égal à $n(n-1)\alpha/\beta$ sec. Pour réduire le temps de transfert, nous définissons un critère de voisinage comme suit :
Définition 1 : Le sommet x est un voisin du sommet y avec un saut de h, s'il y a un chemin de x vers y en passant par $(h-1)$ sommets (voir Figure 6.9).

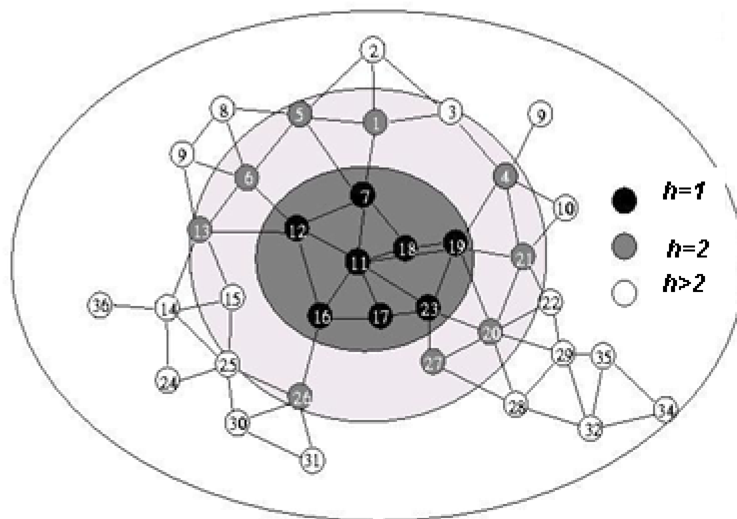

FIGURE 6.9 – Régions pour différentes valeurs du paramètre h (saut)

Chaque sommet x envoie son vecteur d'état $V_t(x)$ à ses voisins avec un saut de h. Nous supposons que chaque zone de périmètre h contient une moyenne de m chemins indépendants à partir du sommet x. Le nombre de messages pour transférer $V_t(x)$ de tous les sommets du graphe est alors égal à $n*h*m$ avec $m \geq 1$ et $h \geq 1$. Son temps de transfert est égal à $n*h*m*\alpha/\beta$ sec. Sous ces hypothèses, nous avons mené des expériences pour estimer le temps de transfert du vecteur d'état. Nous avons défini deux classes de graphes : (i) la classe $C1$ avec des graphes de n sommets (n

variant entre 100 et 900) ; (ii) la classe $C2$: graphes à n sommets variant de 1000 à 4000. Tout d'abord, nous avons fixé les valeurs suivantes : α=0.1Mo, β=100Mo/s, m=10 pour la classe C1 et $m = 40$ pour la classe $C2$. Ensuite, nous avons varié h de 1 à 20. Pour les graphes de classe $C1$, nous notons que le gain devient très élevé à partir d'un nombre de sommets $n = 500$ (10.79 min entre le transfert vers tous les sommets du graphe (*Global*) et $h = 20$) ; d'autre part, cela signifie que la variation de h ne donne pas un gain élevé (2.57 min entre $h = 1$ et $h = 20$) (voir Figure 6.10). Pour les graphes de la classe $C2$, le gain est de l'ordre des heures (3h58 min entre *Global* et $h = 20$) et le gain entre $h = 20$ et $h = 1$ est de 49 min (voir Figure 6.11). Puis, nous fixons $\alpha = 0.1$Mo, $\beta = 100$Mo/s, $h = 10$ et nous varions le paramètre

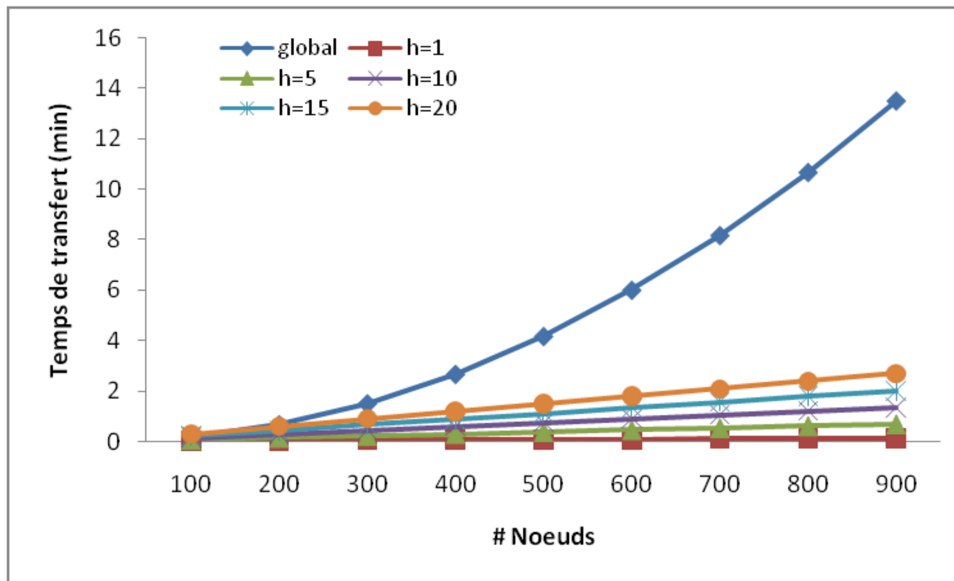

FIGURE 6.10 – Temps de transfert du vecteur d'état pour m=10, h=1, 5, 10, 15 et 20

m de 10 à 30 pour la classe $C1$ et de 20 à 100 pour la classe $C2$. Pour les graphes de la classe $C1$, nous notons que le gain devient très élevé à partir d'un nombre de sommets $n = 500$ (9.14 min entre *Global* et $m = 30$). Ainsi, la variation de m n'a pas donné un gain élevé (3 min entre $m = 10$ et $m = 30$) (voir Figure 6.12). Pour les graphes de la classe $C2$, le gain est de l'ordre des heures (3h34 min entre *Global* et $m = 100$), et le gain entre $m = 20$ et $m = 100$ est de 53 min (voir Figure 6.13).

5.4 Résultats relatifs aux performances de la grille

Pour évaluer notre technique de tolérance aux fautes, nous avons développé, sous Simgrid, un simulateur appelé *DCFT*. Nous utilisons dans notre simulation la configuration d'une grille de calcul composée de 500 sites [11] (voir Tableau 6.5). La grile est constituée de nœuds interconnectés via un réseau rapide. Cependant, les caractéristiques individuelles comme la vitesse du CPU, le nombre de nœuds, etc. varient selon les machines. Nous supposons que tous les transferts de données partagent la bande passante du réseau de manière équitable, et que les conflits de réseau se

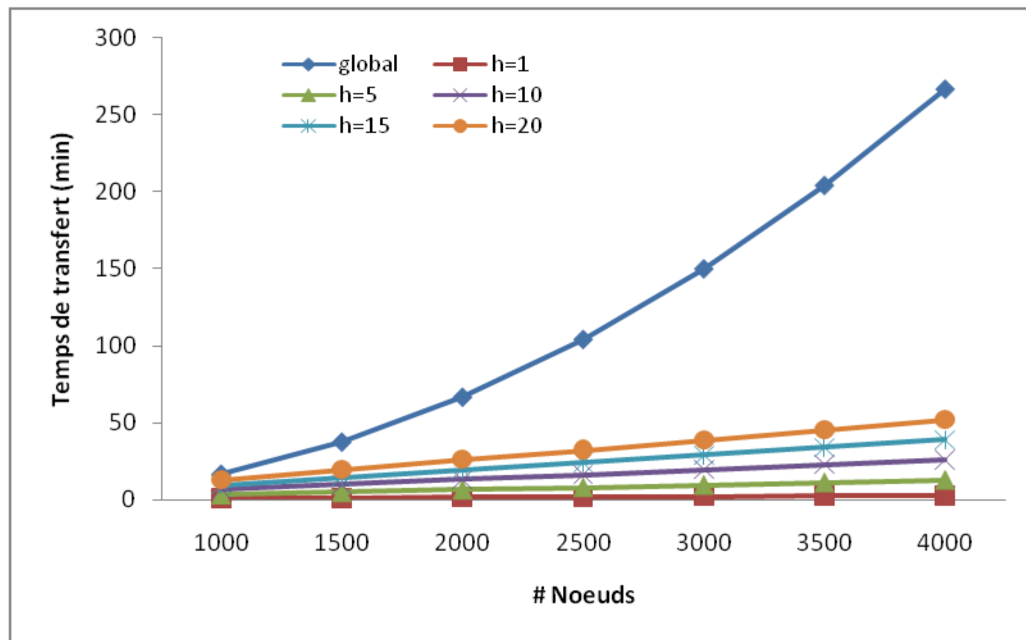

FIGURE 6.11 – Temps de transfert du vecteur d'état pour m=40, h=1, 5, 10, 15 et 20

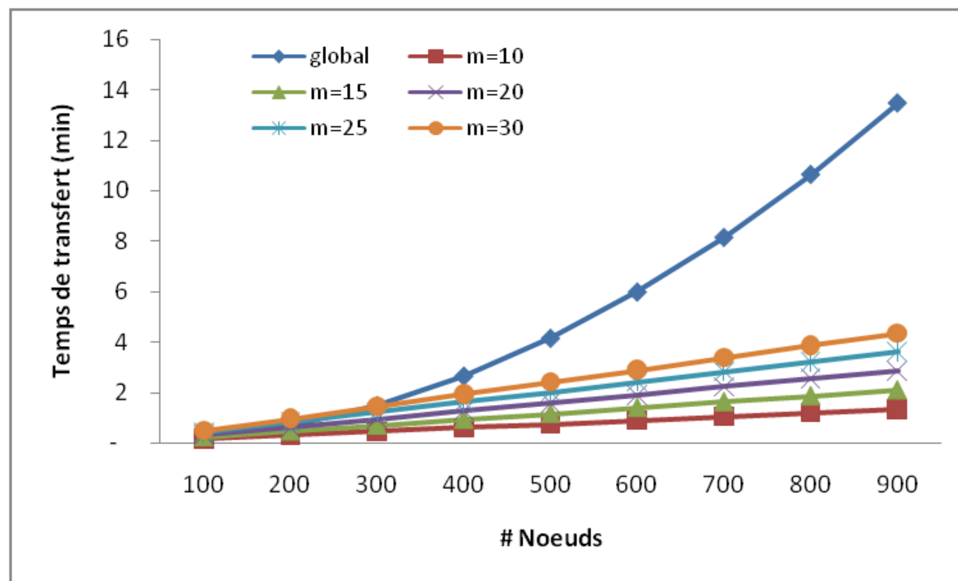

FIGURE 6.12 – Temps de transfert du vecteur d'état pour h=10, m=10, 15, 20, 25 et 30

produisent lorsque les transferts de données multiples utilisent simultanément un chemin de réseau donné. Le modèle de la charge de travail pour les simulations est basé sur *Grid Workload Archive* [90]. Sur la base de ce modèle, un certain nombre de jobs sont générés. Le temps d'arrivée des jobs suit la distribution de Weibull (voir Tableau 6.5). Ainsi, le nombre de jobs augmente de 10 000 (i.e. $\alpha = 9$) à 19 000

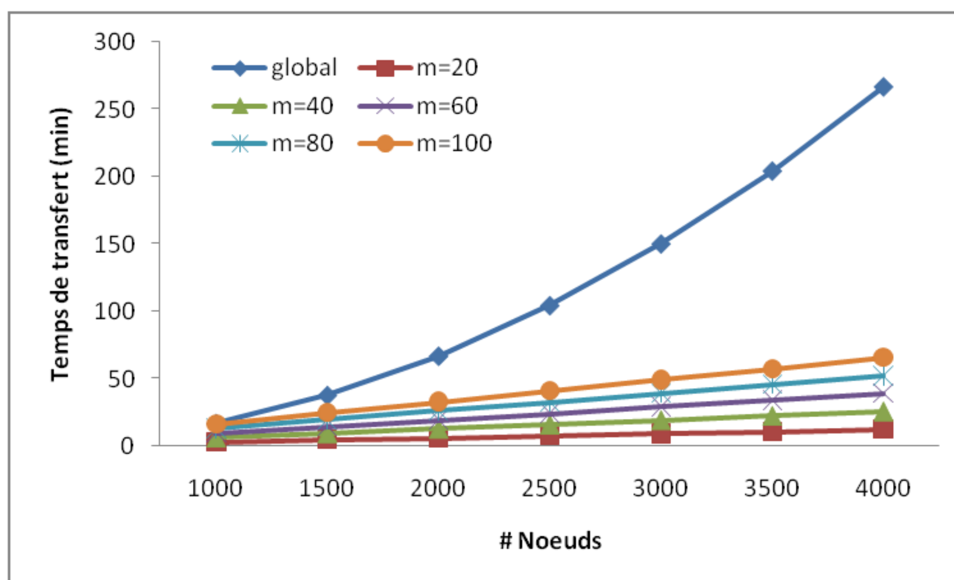

FIGURE 6.13 – Temps de transfert du vecteur d'état pour h=10, m=20, 40, 60, 80 and 100

(i.e. α= 5) [72]. Les temps d'exécution des jobs sont générés de manière aléatoire avec une moyenne de 125 secondes [73]. Pour chaque nœud, les coefficients α_i sont générés de manière aléatoire entre 0 et 1 et leur somme est égale à 1. Le nombre de nœuds est égal à 1000 (avec $h = 5$) et 4000 nœuds (avec $h = 2$) avec des vitesses de calcul différentes. Notre modèle est testé avec des défaillances et l'occurrence des fautes des nœuds suit un processus de Poisson de taux λ_k.

Paramètres	Intervalle de valeurs
Nombre de sites	500
Nombre de nœuds par site	1-8
Nombre de PE par nœud	1-5
Vitesse CPU	10-1.000 Mflops
Bande passante	10 Mbps - 1 Gbps
λ_k $(10^{-4}/s)$	0.1 - 2.0
Temps d'ariivée des jobs	Distribution de Weibull ($5 \leq \alpha \leq 9$, β=4.25) [72]

TABLE 6.5 – Paramètres de la grille de calcul

Pour évaluer l'efficacité de notre technique de tolérance aux fautes, et d'étudier les effets de transfert de données, nous définissons trois métriques, dont deux pour les utilisateurs individuels et l'autre pour les administrateurs du système. Pour les utilisateurs individuels, nous définissons le temps moyen de réponse (*Average Res-*

ponse Time ART) et le temps moyen d'attente (*Average Wait Time* AWT). Ces temps sont calculés comme suit (N est le nombre total de jobs) :

$$ART = \frac{1}{N} \sum_{j \in job} (ET_j - QT_j) \tag{6.1}$$

$$AWT = \frac{1}{N} \sum_{j \in job} (ST_j - QT_j) \tag{6.2}$$

où QT_j, ST_j et ET_j sont les moments où le job j est soumis dans la grille, quand il commence l'exécution, et quand il se termine. Pour l'administrateur du système, nous définissons le coût de migration des données (*Data Migration Overhead* DMOH), qui est calculé comme suit :

$$DMOH = \frac{\text{Temps total de migration des données}}{\sum_j (ET_j - QT_j)} \tag{6.3}$$

En d'autres termes, DMOH est la fraction du volume de données transféré entre les jobs sur le temps de réponse des jobs.

5.4.1 Politiques de comparaison

Nous évaluons le modèle proposé par rapport à deux autres politiques comme suit :

1. **Min-Min :** Cette politique établit le temps d'accomplissement minimum pour chaque job. Elle attribue ensuite le job au nœud qui offre le temps d'accomplissement minimum. Min-min considère le temps d'accomplissement minimum pour tous les jobs, et peut ordonnancer, à chaque itération, le job qui augmentera au minimum la durée d'exécution globale [26].
2. **Decent_MST :** Dans cette politique, les substituts du nœud défaillant sont sélectionnés comme suit : (i) chaque nœud reçoit les vecteurs d'état (avec seulement deux paramètres : vitesse de CPU, bande passante) des voisins au sein de h sauts ; (ii) chaque nœud construit un arbre de recouvrement minimal de tous les nœuds qui ont envoyé leurs vecteurs d'état (nous ne classifions pas les nœuds en identiques, plus performant et moins performant) et nous affectons chaque job vers le nœud le plus proche, qui offre un temps d'accomplissement minimal pour ce job.

5.4.2 Résultats de simulation

Nous avons effectué nos expériences en utilisant les trois cas suivants, qui sont basés sur les ressources de calcul et le volume de données nécessaire à transmettre pour l'exécution des jobs.

Cas 1 Demande de petit volume de données (1 Mo à 10 Mo) avec une charge de travail moyenne ou élevée (10000 à 60000 jobs).

Cas 2 Demande de petit volume données (1 Mo à 10 Mo), charge de travail élevée (60000 jobs) et une durée d'exécution des jobs variable.

Cas 3 Demande de grand volume de données (500 Mo et 10 Go) et une charge de travail élevée (60000 jobs).

Cas 1 : Pour chaque test, nous avons injecté un taux de défaillance allant de 2% à 50% et nous avons calculé son ART et son AWT. Il y a 1.000 nœuds et le nombre de jobs soumis est de 10000 (voir Figures 6.14 et 6.15). Dans les Figures 6.16 et 6.16, il y a 4000 nœuds et le nombre de jobs est de 60000. La durée moyenne d'exécution d'un job est égal à 125 sec.

Le temps de réponse moyen (ART) : Lorsque le volume de données à transmettre est petit, seule la vitesse de CPU aura une incidence sur l'ART. Par conséquent, l'effet du taux de transmission de données sur le temps d'achèvement d'un job n'est pas significatif. Les résultats expérimentaux sont présentés dans les Figures 6.14 et 6.16. Dans l'approche DCFT, l'ART augmente de 115.00 à 139.75 secondes. Le plus haut ART est de 218.63 secondes pour Decent_MST et 234.40 secondes pour la politique Min-Min. La différence de temps de l'ART moyenne entre DCFT et Decent_MST est d'environ 23.21 secondes. (Voir Figure 6.14). Dans le cas de 4000 nœuds et de 60000 jobs, ART atteint 238.25 secondes pour DCFT, 346.38 secondes pour Decent_MST et 363.05 secondes (voir Figure 6.16). Min-Min et Decent_MST sélectionnent les substituts les plus rapides sans prendre en compte la charge du nœud. DCFT tient en compte non seulement de la vitesse de CPU, mais aussi de la charge du nœud. Nous augmentons les coefficients de vitesse de CPU et la charge de travail. DCFT donne le meilleur ART où nous sélectionnons les substituts identiques ou les plus performant que le nœud défaillant.

Temps d'attente moyen (AWT) : Le temps d'attente moyen est affecté par la charge du nœud. Les résultats expérimentaux sont présentés dans les Figures 6.15 et 6.17. Dans l'approche DCFT, AWT varie de 10.00 à 16.50 secondes. L'AWT de Decent_MST et Min-Min sont proches et ils atteignent 27.75 secondes (voir Figure 6.15). Nous remarquons, à partir de la Figure 6.17, que l'AWT de Decent_MST et Min-Min augmentent de manière significative jusqu'à atteindre 63.00 et 66.80 secondes, mais pour DCFT, il est égal à 40.00 secondes. Min-Min et Decent_MST ne tiennent pas compte de la charge du nœud lors de la sélection des substituts. DCFT sélectionne les substituts identiques ou plus performants avec l'augmentation du coefficient de la charge de travail du processeur, ce qui réduit l'AWT.

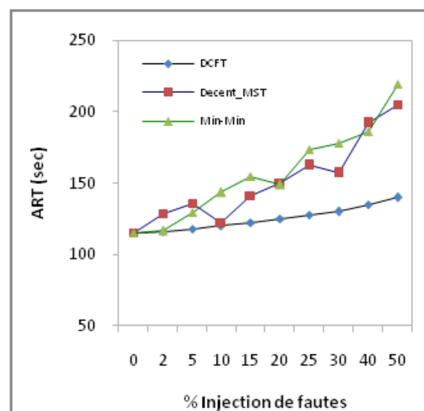

FIGURE 6.14 – Cas 1 : Temps de réponse moyen (1000 nœuds, 10000 jobs)

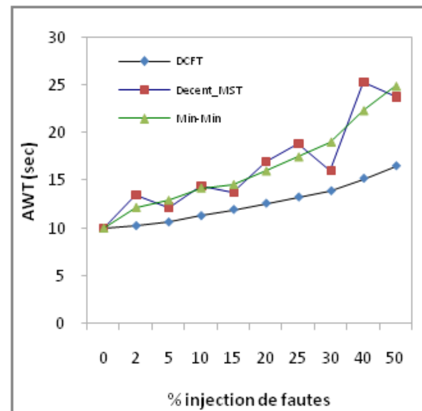

FIGURE 6.15 – Cas 1 : Temps d'attente moyen (1000 nœuds, 10000 jobs)

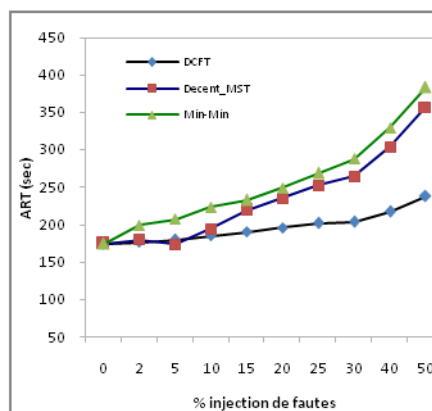

FIGURE 6.16 – Cas 1 : Temps de réponse moyen (4000 nœuds, 60000 jobs)

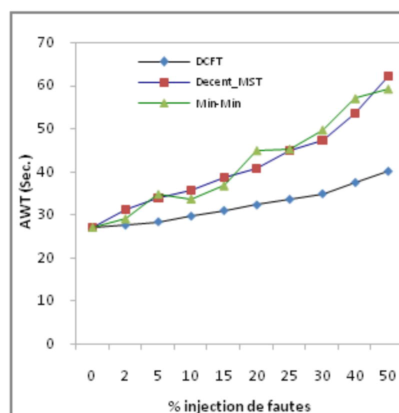

FIGURE 6.17 – Cas 1 : Temps d'attente moyen (4000 nœuds, 60000 jobs)

Cas 2 : Dans ce cas, nous avons 4000 nœuds avec 60000 jobs. Nous faisons varier la durée moyenne d'un job de 50 à 350 secondes. Le taux de défaillance est de 15%. Nous voyons clairement que le DCFT donne le meilleur ART par rapport aux politiques Decent_MST et Min-Min. L'ART de Decent_MST est de 1.8x supérieure à celle de DCFT. Cependant, la différence entre l'ART de Decent_MST et Min-Min

est petite, à savoir près de 6 secondes (voir Figure 6.18). Comme le montre la Figure 6.19 avec la variation de la durée d'exécution des jobs, la plus grande valeur d'AWT de DCFT est de 63.00 secondes. Decent_MST et Min-Min atteignent 104 secondes et la différence d'AWT entre ces deux politiques est d'environ 1.69 secondes. Lorsque nous augmentons la durée des job avec un petit volume de données à transmettre, le temps d'accomplissement d'un job est affecté par la durée de transmission des données des jobs. L'ART et l'AWT de Min-Min et Decent_MST augmentent considérablement avec l'augmentation de la durée d'accomplissement des jobs. DCFT ajoute la charge du nœud pour sélectionner les susbsituts.

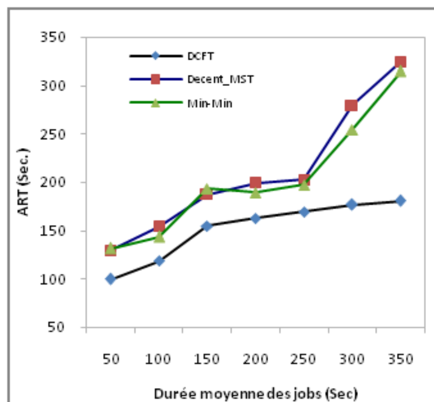

Figure 6.18 – Cas 2 : Temps de réponse moyen (4000 nœuds, 60000 jobs)

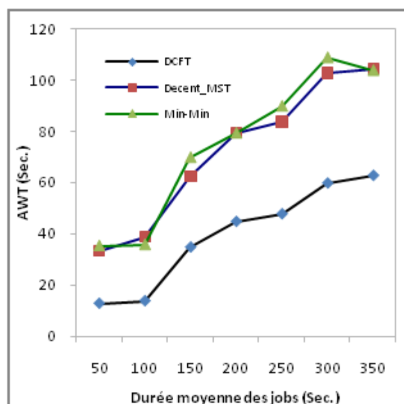

Figure 6.19 – Cas 2 : Temps d'attente moyen (4000 nœuds, 60000 jobs)

Cas 3 : Ce cas concerne un environnement de 4000 nœuds et 60000 jobs avec une durée d'exécution moyenne d'un job est fixée à 125 secondes. Dans ces simulations, nous avons injecté un taux de défaillance allant de 2% à 50%. Nous avons effectué deux tests basés sur deux valeurs de la moyenne des données transférées en entrée/sortie (*Mean Transferred Data in/out* MTD) (MTD = 500 Mo et 10 Go) et nous avons calculé le DMOH pour chaque test. Pour MTD = 500 Mo, le DMOH de DCFT est inférieur à 2.5%. Le DMOH des politiques Decent_MST et Min-Min est toujours le plus élevé. Le DMOH de Min-Min est supérieur à 3.2x DCFT (voir Figure 6.20). Pour une DMT de 10 Go, le DMOH de DCFT est inférieur à 7%.

La politique Min-Min donne les temps les plus élevés avec un DMOH de 32.77%. Nous observons que DMOH des DCFT, pour 500 Mo, est très faible, mais atteint les 7% pour les 10 Go (voir Figure 6.21). Dans le cas où la demande de ressources de calcul et de transmission de données est grande, la vitesse de CPU et la transmission des données disponibles deviennent des paramètres très importants. Le DMOH de Min-Min est toujours le plus élevé et s'approche du point de saturation de façon exponentielle, car Min-Min ne considère pas la transmission de données pour sélectionner un substitut. Decent_MST donne de meilleurs résultats que Min-Min en sélectionnant les substituts les plus proches. DCFT augmente les coefficients de la bande passante et de la latence pour réduire le temps de transmission.

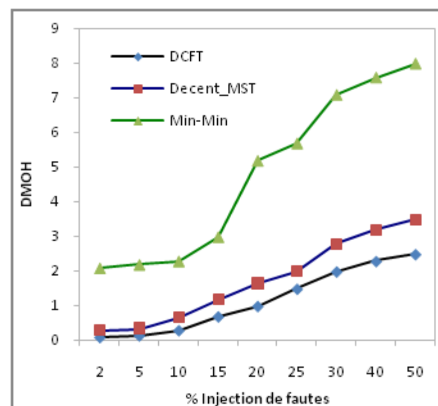

FIGURE 6.20 – Cas 3 : Coût de migration des données (500 Mo)

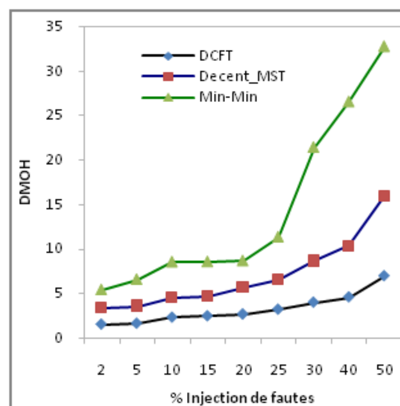

FIGURE 6.21 – Cas 3 : Coût de migration des données (10 Go)

6 Modèle de degré de tolérance

Nous allons, maintenant, évaluer les performances de notre modèle à travers une simulation axée sur des événements discrets, en utilisant Graphstream. Chaque série de simulation est effectuée sur un graphe non orienté, généré de manière aléatoire

	M1 : Moyenne des voisins				M2 : Modulo				M3 : Méthode itérative			
# Sommets	α	R	V	B	α	R	V	B	α	R	V	B
100	60	45	5	50	58	28	10	62	47	0	1	99
200	30	96	16	88	29	79	17	104	22	5	8	187
300	20	140	29	131	20	140	29	131	10	0	3	297
400	15	188	47	165	14	141	47	212	8	4	10	386
500	12	225	66	209	12	225	66	209	6	11	12	477
600	10	277	76	247	8	112	88	400	5	16	23	561
700	8	262	89	349	9	351	99	250	4	26	24	650
800	7	301	114	385	6	178	123	499	4	47	56	697
900	6	316	139	445	6	316	139	445	4	87	95	718
1000	6	464	142	394	5	294	170	536	3	54	95	851

TABLE 6.6 – Coloration des sommets par trois méthodes de calcul du degré de tolérance.

comme suit : étant donné un ensemble de nœuds et un nombre d'arêtes ajoutées entre les paires de nœuds de manière aléatoire, les poids des arêtes sont sélectionnés au hasard [126].

6.1 Influence du degré de tolérance

6.1.1 Phase de coloration

Nous faisons varier le nombre de nœuds de 100 jusqu'à 1000 par pas de 100 et nous fixons le nombre d'arcs à 3000. Nous générons un graphe pour chaque cas et nous testons la coloration du graphe pour chaque méthode de calcul du degré de tolérance (voir Tableau 6.6).

La méthode M3 (qui est une méthode itérative) donne toujours un degré de tolérance plus faible par rapport aux autres méthodes ; ses couleurs rouges et vertes sont très proches avec une valeur moyenne de 3.30% pour le rouge, 4.52% pour le vert et 92.18% pour la couleur bleu. Les degrés de tolérance de la méthode M1 (qui prend en compte la moyenne des voisins d'un nœud) sont toujours supérieurs ou égaux à ceux de M2 (qui utilise la fonction modulo) avec une différence maximale égale à 2. Les deux méthodes donnent des sommets de couleur verte très faibles par

	M1 : Moyenne des voisins				M2 : Modulo				M3 : Méthode itérative			
# Sommets	α	R	V	B	α	R	V	B	α	R	V	B
100	60	0	100	0	58	0	53	47	47	0	1	99
200	30	0	200	0	29	0	151	49	22	0	13	187
300	20	0	300	0	20	0	300	0	10	0	3	297
400	15	0	400	0	14	0	274	126	8	0	14	386
500	12	0	500	0	12	0	500	0	6	0	23	477
600	10	0	600	0	8	0	242	358	5	0	40	560
700	8	0	700	0	9	107	593	0	4	0	51	649
800	7	0	800	0	6	0	383	417	4	0	111	689
900	6	0	900	0	6	0	656	244	4	0	201	699
1000	6	0	1000	0	5	0	604	396	3	0	160	840

TABLE 6.7 – Phase de stabilisation par trois méthodes (M1, M2 et M3).

rapport aux autres couleurs.

6.1.2 Phase de stabilisation

Après la phase de stabilisation des graphes colorés précédemment, tous les sommets colorés par la méthode M1 convergent vers l'état $Etat(G) = V$; par contre, ceux colorés par les méthodes M2 et M3 convergent vers l'état $Etat(G) = VB$. Les sommets de couleur bleu atteignent jusqu'à 96.12% dans la méthode M3. La méthode M1 donne toujours la meilleure valeur du degré de tolérance α, parce qu'elle fait converger tous les sommets vers un état stable (voir Tableau 6.7).

6.2 Tolérance aux fautes

Notre modèle a montré sa capacité à tolérer les fautes de type crash des nœuds. Pour un graphe de 600 sommets et 3000 arcs, nous avons pu tolérer toutes les fautes des nœuds, mais ce taux commence à diminuer quand le taux des nœuds défaillants dépasse 66% des nœuds de la grille (voir Figure 6.22).

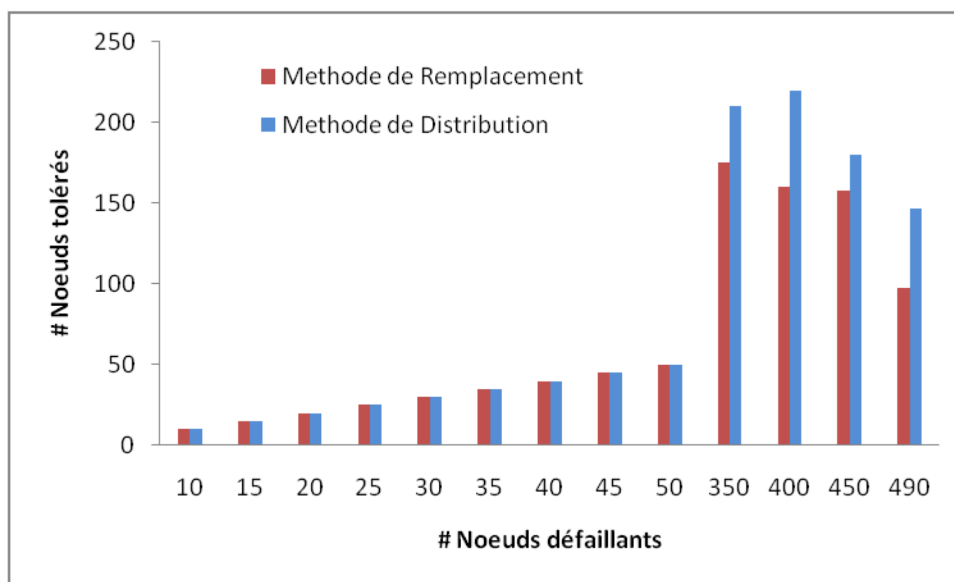

FIGURE 6.22 – Tolérance aux fautes des nœuds par deux méthodes (Remplacement et Distribution)

7 Modèle fiable de tolérance aux fautes

Dans cette section, nous allons évaluer les performances de notre technique de tolérance aux fautes par des simulations. Ces simulations utilisent une configuration d'une grille de calcul composée de six clusters (voir Tableau 6.8). Chaque cluster est architecturalement semblable aux autres ; il se compose de nœuds SMP interconnectés via un réseau exclusif rapide. Cependant, les caractéristiques individuelles comme la vitesse du CPU, la taille de SMP, le nombre de nœuds, etc. varient selon les machines. Il n'est pas nécessaire que tous les nœuds soient opérationnels dans la grille. Chaque nœud de travail ne peut exécuter qu'une seule tâche à la fois. Nous supposons que tous les transferts de données partagent équitablement la bande passante du réseau, et que les conflits de réseau se produisent lorsque les transferts de données multiples utilisent simultanément un chemin de réseau donné. Dans notre modèle, chaque réseau du cluster a une bande passante maximale de 800 Mo/s, tandis que 100 Mo/s sont disponibles entre les réseaux de clusters. Le nombre de liens de la grille est de quatre fois le nombre de nœuds. Chaque lien a une bande passante de liaison entre 100 et 800 Mo/s et l'intensité de rupture comprise entre 0.1 à 2 ($10^{-5}/s$). Le nombre de tâches dans la grille est dix fois les nœuds présents dans la grille, avec une variation aléatoire de $\mp$ 20%. Chaque tâche a une complexité de calcul entre 5 à 10 Giga opérations et la quantité de données échangées pour le traitement d'une tâche se situe entre 0.01 à 200 Go.

Notre technique est testée en présence de défaillances. Des défaillances se produisent dans chaque nœud suivant un processus de Poisson de taux λ_k. Les défaillances sont instantanées, de sorte que les nœuds peuvent être réparés après un temps de réparation de n^{ieme} MDTTR, n étant une valeur positive aléatoire inférieure à 10. Pendant le temps de la réparation, nous tolérons les fautes du nœud défaillant. Notre infrastructure de simulation a été créée sous Simgrid [31].

Cluster Id.	# Nœuds	# Nœuds de travail	Vitesse CPU	S_k	λ_k	WT_k	λ_k^R	S_{Rk}
M1	192	16	375	150	0.1	5	0.1	2
M2	305	4	332	100	1.8	15	0.2	3
M3	144	8	375	150	1.2	10	0.3	2
M4	8	16	1300	250	1.5	5	0.4	5
M5	74	4	375	150	1.9	15	0.1	4
M6	180	4	375	150	0.8	15	0.2	3

TABLE 6.8 – Paramètres de configuration de la grille de calcul

7.1 Métriques de performance

Dans cette section, nous allons procéder à l'évaluation de l'efficacité de notre technique de tolérance aux fautes et du protocole de migration des tâches. Nous définissons trois indicateurs clés, dont deux pour les utilisateurs individuels et l'autre pour les administrateurs système [181]. Pour les utilisateurs individuels, nous définissons le Temps de réponse moyen (ART) et le temps d'attente moyen (AWT). Concernant l'administrateur du système, nous définissons une métrique appelée Fraction des programmes migrés (*Fraction of Programs Migrated* FOPM). Cette métrique nous permet de déterminer le taux des tâches migrées par rapport au total des tâches dans la grille. Elle est définie comme suit :

$$FOPM = \frac{\text{Nombre de tâches migrées}}{\text{Nombre total de tâches défaillantes}} \tag{6.4}$$

7.2 Algorithmes de référence

Nous utilisons trois algorithmes en tant que références de base pour la comparaison. Ces 3 algorithmes sont associés aux stratégies centralisée, décentralisée et à l'heuristique Max-Min qui combinent le RNFR et le LNFR.

1. **Stratégie Centralisée** : Dans cette Stratégie, toutes les tâches sont soumises au RMS, qui est responsable de la prise de décisions globales et de l'allocation de chaque tâche à un nœud spécifique. L'algorithme relatif à cette stratégie suit l'état de chaque tâche et met à jour l'information sur tous les nœuds disponibles. Quand un nœud tombe en panne, le RMS fait migrer toutes ses tâches vers les nœuds les plus fiables dans la grille.
2. **Stratégie Décentralisée :** Dans l'algorithme associé à cette Stratégie, chaque nœud est responsable de tolérer ses fautes. Toutes les tâches sont soumises au RMS, qui affecte les tâches aux nœuds disponibles dans la grille. Quand un nœud tombe en panne, toutes ses tâches sont migrées vers les nœuds voisins les plus fiables.

3. **Heuristique Max-Min** : Dans ce cette heuristique, nous nous concentrons sur RNFR et LNFR. RNFR est basé sur l'heuristique Max-Min. Sa métrique est temps d'achèvement ou de terminaison minimum (*Minimum Completion Time,* MCT). L'algorithme commence avec l'ensemble de toutes les tâches non affectées du nœud défaillant. Puis, il calcule l'ensemble des temps d'achèvement minimaux, $M = \{Min(completion_time(T_i, N_j))\}$. Ensuite, la tâche qui a un temps d'accomplissement égal à M est sélectionnée et puis elle est transférée vers le nœud correspondant.

7.3 Résultats expérimentaux

Dans cette section, nous allons présenter et discuter les résultats des expérimentations que nous avons effectuées pour évaluer les métriques définies dans les sections précédentes.

Temps de réponse moyen : Dans les simulations relatives aux métriques ART et AWT, nous avons utilisé le modèle de la grille présenté dans le Tableau 6.8. Pour chaque test, nous avons injecté un taux de défaillance allant de 2% à 50% et nous avons calculé l'ART. Dans l'approche de LRNFR, l'ART augmente de 135.60 à 183.82 secondes. Nous comparons les performances de notre technique de tolérance aux fautes avec les approches centralisées, décentralisées et Max-Min. La différence moyenne du temps de l'ART entre LRNFR et l'approche décentralisée est d'environ 140 secondes, alors qu'avec l'approche centralisée, il est d'environ 107 secondes. Ainsi, notre technique de tolérance aux fautes est plus performante que l'approche décentralisée et est presque aussi efficace que l'approche centralisée. L'approche Max-Min donne une différence de 34 secondes. Avec un taux d'injection de fautes moins de 10%, Max-Min donne une valeur ART proche de LRNFR. Pour les taux d'injection de défaillance élevés, la différence de l'ART devient plus grande. Comme Max-Min ne tient pas compte de la fiabilité des nœuds et du volume de données échangées, ceci affecte son ART (voir Figure 6.23).

Temps d'attente moyen : L'AWT de LRNFR passe de 17.22 à 25.04 secondes.

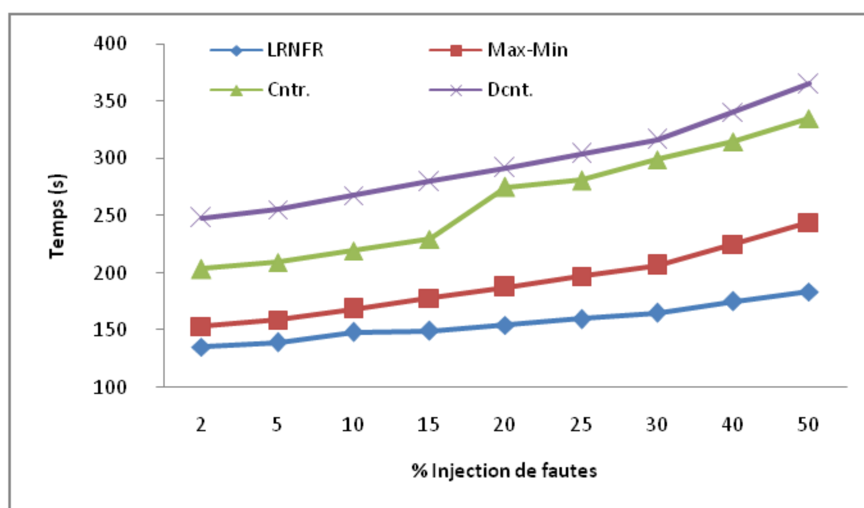

FIGURE 6.23 – Le temps de réponse moyen

Quand nous comparons la différence moyenne du temps AWT entre LRNFR et l'approche décentralisée, nous trouvons qu'elle est de l'ordre de 8 secondes, avec l'approche centralisée, elle est de 5 secondes et avec l'approche Max-Min, elle est de l'ordre de 3 secondes. En ce qui concerne la métrique AWT, la différence n'est pas significative (voir Figure 6.24).

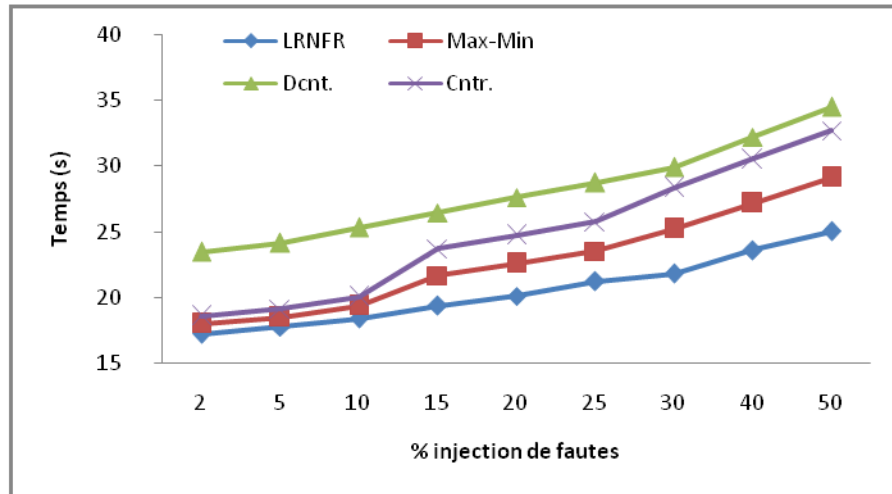

Figure 6.24 – Evaluation du temps d'attente moyen

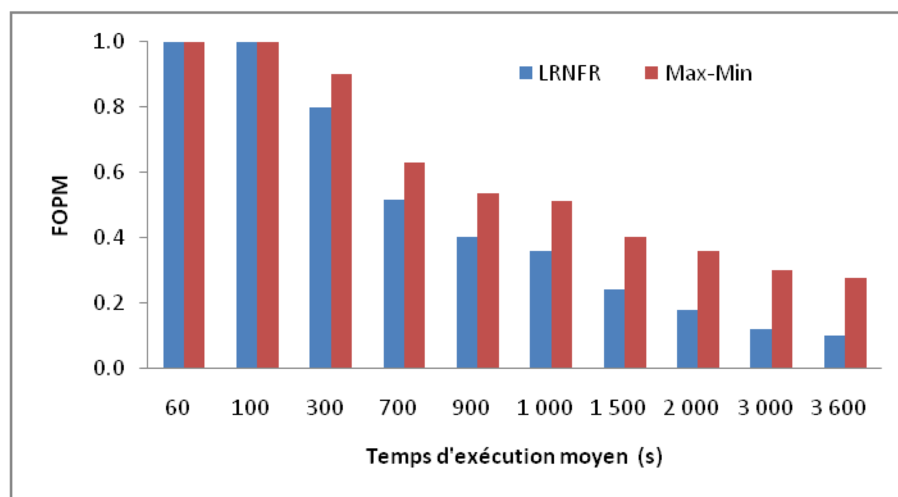

Figure 6.25 – Fraction des tâches migrées

Fraction des tâches migrées : Dans cette simulation, nous avons injecté 10% des défaillances, nous avons fait varier le temps d'exécution moyen (*Mean Execution Time* MET) des tâches soumises dans la grille de 60 secondes à 3600 secondes, la taille des données échangées est de 500 Mo et nous avons calculé FOPM pour chaque cas. La fraction FOPM des approches décentralisées et centralisées est de 100%, parce que ces approches migrent toutes les tâches du nœud défaillant. La Figue 6.25 présente le FOPM de LRNFR et Max-Min. Pour MET inférieur à 700

secondes, FOPM est supérieure à 50% pour les deux approches, ce qui augmente le nombre de tâches migrées. Quand nous augmentons la valeur de MET, FOPM diminue de manière significative. Pour une valeur de MET égale à 3600 seconds, le FOPM de Max-Min atteint 28% et 10% pour RLFNR. Min-Max migre plus de tâches, car elle ne repose que sur le MCT.

8 Conclusion

Les expérimentations présentées dans ce chapitre nous ont permis de mettre en valeur la praticabilité de nos modèles, leurs avantages ainsi que leurs limites. Les modèles centralisés hiérarchiques supportent mal le passage à l'échelle. Les modèles décentralisés résistent au problème du gestionnaire et supportent facilement le passage à l'échelle ainsi que la dynamicité. Par ailleur, l'introduction de la fiabilité des services améliore énormément les techniques de tolérance aux fautes dans les grilles de calcul en général.

Conclusion et Perspectives

Les travaux présentés dans cette thèse portent sur la tolérance aux fautes dans les grilles de calcul. Les grilles de calcul représentent des infrastructures très intéressantes pour répondre à la demande croissante de puissance de calcul, en ce qui concerne des applications de calcul de haute performance. Ces infrastructures permettent d'agréger et de mutualiser des ressources informatiques, distribuées sur une échelle très large, pour mettre en place une puissance de calcul impossible à obtenir de manière indivudelle. Toutefois, les grilles de calcul ne sont pas encore arrivées à un niveau de maturité qui leur permet d'être généralisées et exploitable par un large public. Les principales difficultés proviennent de la nature même des grilles de calcul, qui sont des architectures très hétérogènes et fortement dynamqiues. De plus leur exploitation nécessite une technicité qui n'est pas forcément à la portée de tout utilisateur. De nombreux travaux de recherche ont proposé des solutions permettant de faire face à une partie de ces difficultés alors que d'autres sont encore en cours. En ce qui nous concerne, nous nous sommes intéressés au problème de tolérance aux fautes dans les grilles de calcul. Dans ces architectures, le risque de défaillance est très important, car le nombre de ressources (physiques et logiques) est à la fois très important, mais également ces ressources sont dispersés sur une échelle très large et sont très diverses les unes par rapport aux autres. Il est donc nécessaire de gérer ce risque élevé de défaillances, en proposant des techniques de tolérance aux fautes qui puissent prendre en compte cet aspect dans la gestion globale d'une grille, tant au niveau infrastructure qu'au niveau applicatifs. Vu l'importance et l'intérêt réel d'une telle problématique, de nombreux travaux ont été menés sur le thème général de sûreté de fonctionnement dans les systèmes distribués à large échelle, et en particulier les grilles. En ce qui concerne, nos travaux, nous nous sommes orientés vers trois axes complémentaires pour la gestion de la tolérance aux fautes dans les grilles. Cette gestion sera d'autant plus facile, si cette tolérance aux fautes se base sur des modèles théoriques représentant les grilles. Pour cela, nous avons concentré nos réflexions sur trois types de modèles.

1. Les modèles hiérarchiques de tolérance aux fautes sous Globus.
2. Les modèles décentralisés de tolérance aux fautes, basés sur les graphes colorés dynamiques.
3. Les modèles de tolérance aux fautes qui prend en compte la fiabilité des services des grilles de calcul.

Contributions :

Les travaux que nous avons mené dans ce contexte, ainsi que les résultats que nous avons obtenus, permettent de situer nos contributions comme suit :

- Partant d'un modèle abstrait de grille, appelé G/S/M, nous avons procédé à son amélioration par un modèle hiérarchique dynamique. Ce modèle est basé sur une arborescence de niveaux dynamiques, en fonction du nombre de nœuds disponibles dans une grille. Selon cette nouvelle représentation, les techniques de tolérance aux fautes que nous avons proposées ont permis de réduire sensiblement la migration de jobs dans les niveaux de l'arborescence, dans le cas de pannes. Cette réduction est très appréciable car elle réduit le taux de communication dans une grille.
- Comme dexuième contribution significative, nous nous sommes intéressés aux grilles mobiles, où nous avons proposé un modèle multi-arborescent pour les représenter. Ce modèle, en plus de la structure intéressante qu'il offre pour la gestion de la tolérance aux fautes dans le cas mobile, permet de prendre en charge certains aspects spécifiques de ces grilles, comme la disponibilité limitée, le taux élevé de mobilité élevé, la fréquence des opérations de déconnexion fréquente,etc. Le modèle de tolérance aux fautes proposé est dérivé du modèle aborescent associé à une grille mobile.
- La troisième contribution, qui représentent le noyau de cette thèse, porte sur la définition de graphes colorés dynamiques pour capter les deux caractéristiques d'une grille, à savoir l'hétérogénéité et la dynamicité. La richesse de ces graphes, notamment grâce aux couleurs, nous a permis de mettre en place deux techniques très intéressantes de tolérance aux fautes, la première se base sur le niveau de performance des substituts qui remplacent un nœud défaillant, et la deuxième se base sur le voisinage d'un nœud défaillant plutôt que sur tous les nœuds d'une grille.
- La quatrième contribution investit dans la prise en compte de la fiablité des services de grilles, comme un critére de base pour la tolérance aux fautes dans les grilles de calcul. Elle tente de réduire le nombre des tâches migrées en appliquant une technique de migration périodique basée sur le MDTTR comme période.

L'ensemble des propositions qui ont été faites dans cette thèse ont été validées expérimentalement soit à partir de simulateurs existants, soit à partir de simulateurs que nous avons développés pour nos propres besoins.

Perspectives :

Les résultats théoriques et pratiques que nous avons obtenu dans le cadre de cette thèse, permettent d'envisager un certain nombre de perspectives de recherche fort intéressantes. Trois perspectives nous semblent très pertinentes : (i) une étude théorique plus poussée sur les graphes colorés dynamiques ; nous pensons, en effet, que ces graphes renferment beaucoup de propriétés intéressantes qui peuvent améliorer la prise en charge de la tolérance aux fautes dans les grilles ; (ii) l'utilisation des graphes colorés dynamiques pour traiter d'autres problématiques, comme l'équilibrage de charge, l'ordonnancement des tâches ou des jobs ; et, (iii) l'adaptation des modèles proposés dans cette thèse, à des environnements de type Cloud qui sont considérés, dans certains cas, comme une couche logicielle au dessus des grilles.

Publications de l'auteur

1. Journal international

1. M. Rebbah, Y. Slimani, A. Benyettou, and L. Brunie. : Reliable fault tolerant model for grid computing environments. Multiagent and Grid Systems 10(4) : 213-232 (2014)
2. M. Rebbah, Y. Slimani, and A. Benyettou : A decentralized fault tolerant model for grid computing. International Journal of Computer Science Issues (IJCSI), 11(1) :123-130, January 2014.
3. M. Rebbah, Y. Slimani, and A. Bakbak : Multi-tree model for fault tolerant mobile grid. Asian Journal of Applied Sciences, 4(2) :155-165, 2011.
4. M. Rebbah, Y. Slimani, A. Benyettou, and L. Brunie : Dynamic hierarchical model for fault tolerant grid computing. World Applied Programming Journal, 1(5) :309-321, December 2011.

2. Conférences internationales avec actes et comité de lecture

1. M. Rebbah, Y. Slimani, and A. Benyettou. Decentralized fault tolerant model for p2p grid. In International Conference on Artificial Intelligence and Information Technology (ICA2IT 2014), Ouragla (Algérie), 10-12 Mars 2014.
2. M. Rebbah, Y. Slimani, A. Benyettou, and L. Brunie. Modèle décentralisé pour la tolérance aux fautes dans les grilles de calcul. 6eme Conférence francophone sur les architectures logicielles, CAL 2012, Montpellier, France, 9-11 Mars 2012.
3. M. Rebbah,C. Mokhtari, Y. Slimani, A. Benyettou. "PFT-Grid : Pervasive Fault Tolerant Grid". Conférence Internationale sur le Traitement de l'Information Multimédia, CITIM 2012, Université de Mascara, Algérie, 9-11 Avril 2012.
4. M. Rebbah, C. Mokhtari, M.F. Khaldi, M. Bourassi, and O. Smail. Hierarchical model for fault tolerant grid computing over Globus toolkit. In International Congress on Models, Optimization and Security of Systems (ICMOSS-2010), Tiaret, 29-31 mai 2010.
5. M. Rebbah. "Service de tolérance aux fautes pour systèmes pervasifs". 10th African Conference on Research in Computer Science and Applied Mathematics (CARI'2010), Côte d'Ivoire, 18-21 Octobre 2010.

6. M. Rebbah, C. Mokhtari. " Service de tolérance aux fautes pour les systèmes pervasifs ". International Conference on Systems and Information Processing (ICSIO 09), Guelma, Algérie, 2-4, Mai 2009.
7. M. Rebbah, C. Mokhtari, S. Abid, H. Bouferra. " Distribution verticale et horizontale tolérante aux fautes pour le datamining distribué ". Conférence Internationale des Technologies de l'Information et de la Communication (CI-TIC 09),Sétif, Algérie, 4-5 Mai 2009.
8. M. Rebbah, C. Mokhtari, N. Souane, K. Ghezal. "PGS : Pervasive Grid Simulator Library". In Proceedings of AMS 2009, Third Asia International Conference on Modelling and Simulation, Bandung, Bali, Indonesia, 25-29 May 2009.
9. M. Rebbah, C. Mokhtari, S. Abid, H. Sadjal. " Développement d'une distribution verticale tolérante aux fautes pour le datamining distribué ". 10th Maghrebian Conference on Information Technologies (MCSEAI 08), Oran, Algérie, 28-30 Avril 2008.

3. Séminaires nationaux avec actes et sans comité de lecture

1. M. Rebbah. Tolérance aux pannes dans les grilles de calcul. Journées d'études : Grille de Calcul et Intelligence Artificielle, GCIA06, Université de Mascara, Algérie, 2006.

Bibliographie

[1] J. H. Abawajy. Fault-tolerant scheduling policy for grid computing systems. In *IPDPS 2004*, pages 3289–3295, USA, 2004. IEEE Computer Society.

[2] J. Ahn. Efficient failure detection and recovery scheme for hierarchical distributed monitoring. *Future Generation Communication and Networking*, 2 :510–515, 2007.

[3] J. Almond and D. F. Snelling. UNICORE : uniform access to supercomputing as an element of electronic commerce. *Future Generation Comp. Syst.*, 15(5-6) :539–548, 1999.

[4] A. Andrzejak, S. Graupner, V. Kotov, and H. Trinks. Algorithms for self-organization and adaptive service placement in dynamic distributed systems. Technical report, HP Laboratories Palo Alto, 2002.

[5] G. Antoniu, P. Hatcher, M. Jan, and D. A. Noblet. Performance evaluation of jxta communication layers. CCGRID '05, pages 251–258, USA, 2005. IEEE Computer Society.

[6] B. Arash, K. Mehmet, M. K. Zvi, and P. Wijckoff. Charlotte : Metacomputing on the web. *Future Generation Comp. Syst.*, 15(5-6) :559–570, 1999.

[7] A. Avizienis, J. C. Laprie, and B. Randell. Fundamental concepts of dependability. Technical report, 01145, LAAS-CNRS, Toulouse, France, 2001.

[8] A. Avizienis, J. C. Laprie, and B. Randell. Dependability and its threats - A taxonomy. In *Building the Information Society, IFIP 18th World Computer Congress, Toulouse, France*, pages 91–120, 2004.

[9] M. Baker, R. Buyya, and D. Laforenza. Grids and grid technologies for wide-area distributed computing. *Softw. Pract. Exper.*, 32(15) :1437–1466, 2002.

[10] N. Balani and R. Hathi. *Apache CXF Web Service Development.* Packt Publishing, 2009.

[11] J. Balasangameshwara and N. Raju. A hybrid policy for fault tolerant load balancing in grid computing environments. *J. Netw. Comput. Appl.*, 35(1) :412–422, 2012.

[12] A. Barak. The mosix multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13 :4–5, 1998.

[13] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A user"s guide to pvm parallel virtual machine. Technical report, University of Tennessee, USA, 1991.

[14] W.H. Bell, D.G. Cameron, A.P. Millar, L. Capozza, K. Stockinger, and F. Zini. Optorsim : A grid simulator for studying dynamic data replication strategies. *IJHPCA*, 17(4) :403–416, 2003.

[15] F. Berman, G. Fox, and A. J. G. Hey. *Grid Computing : Making the Global Infrastructure a Reality.* John Wiley & Sons, Inc., USA, 2003.

[16] T. Berners-Lee, R. T. Fielding, and H. Frystyk Nielsen. Hypertext transfer protocol - http/1.0. Technical report, 1994.

[17] H. Bersini. And the winner is...not the fittest [coevolution]. In *In Evolutionary Computation, Proceedings of the 2002 Congress on CEC*, (CEC '02), pages 1510–1515, 2002.

[18] H. Bersini. *des réseaux et des sciences - Biologie, informatique, sociologie : l'omniprésence des réseaux.* Vuibert informatique, vuibert edition, 2005.

[19] M. Bertier. *Service de détection de défaillances hiérarchique.* Thèse de doctorat, Université de Paris 6, 2004.

[20] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. DSN 02, pages 354–363, USA, 2002. IEEE Computer Society.

[21] M. Bertier, O. Marin, and P. Sens. Performance analysis of a hierarchical failure detector. In *In Proceedings of the International Conference on Dependable Systems and Networks*, pages 635–644, 2003.

[22] F. Boem, R.M. G. Ferrari, T. Parisini, and M. M. Polycarpou. A distributed fault detection methodology for a class of large-scale uncertain input-output discrete-time nonlinear systems. In *CDC-ECE*, pages 897–902. IEEE, 2011.

[23] J. A. Bondy. *Graph Theory With Applications.* Elsevier Science Ltd., UK, 1976.

[24] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. Using group replication for resilience on exascale systems. *International Journal of High Performance Computing Applications*, 28(2) :210–224, 2014.

[25] A. Bouteiller, T. Hérault, G. Krawezik, et al. Mpich-v project : A multiprotocol automatic fault-tolerant mpi. *IJHPCA*, 20(3) :319–333, 2006.

[26] T.D. Braun, H.J. Siegel, N. Beck, L. L. Bölöni, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J. Parallel Distrib. Comput.*, 61(6) :810–837, 2001.

[27] K. Budati, J.D. Sonnek, A. Chandra, and J.B. Weissman. Ridge : combining reliability and performance in open grid platforms. In *HPDC*, pages 55–64, 2007.

[28] B. Buyya and M. Murshed. Gridsim : A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation : Practice and Experience (CCPE)*, 14(13) :1175–1220, 2002.

[29] R.Y. Camargo, R. Cerqueira, and F. Kon. Strategies for storage of checkpointing data using non-dedicated repositories on grid systems. In *In Proc. 3rd International Workshop on Middleware for Grid Computing*, pages 1–6. ACM Press, 2005.

[30] F. Cappello, P. Primet, O. Richard, C. Cérin, and P. Sens. Data gridexplorer : une plate-forme d'émulation de grilles. In *ACI Masse de Données / DGDX Paristic*, pages 506–520, 2005.

[31] H. Casanova. Simgrid : A toolkit for the simulation of application scheduling. CCGRID '01, pages 430–437, USA, 2001. IEEE Computer Society.

[32] A. Chakrabarti and S. Sengupta. Scalable and distributed mechanisms for integrated scheduling and replication in data grids. In *Distributed Computing and Networking*, volume 4904 of *Lecture Notes in Computer Science*, pages 227–238. Springer Berlin Heidelberg, 2008.

[33] S. Chakravorty, C.L. Mendes, and L.V. Kalé. Proactive fault tolerance in MPI applications via task migration. In *HiPC 2006*, pages 485–496, 2006.

[34] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2) :225–267, 1996.

[35] K. M. Chandy and L. Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1) :63–75, 1985.

[36] H. Chen, H. Cheng, G. Jiang, and K. Yoshihira. Exploiting local and global invariants for the management of large scale information systems. In *Proceedings of the 8th IEEE (ICDM 2008)*, pages 113–122. IEEE Computer Society, Italy, 2008.

[37] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Proceedings. International Conference on Autonomic Computing*, pages 36–43, 2004.

[38] S. Choi, M. Baik, C. Hwang, J. Gil, and H. Yu. Volunteer availability based fault tolerant scheduling mechanism in desktop grid computing environment. NCA '04, pages 366–371, USA, 2004. IEEE Computer Society.

[39] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J.S. Chase. Correlating instrumentation data to system states : A building block for automated diagnosis and control. OSDI'04, pages 16–32, USA, 2004. USENIX Association.

[40] Platform Computing Corporation. Administering platform process manager, version 3.0, March 2005.

[41] P. Crosby. Edgsim. a simulation of the european data grid project. *http ://www.hep.ucl.ac.uk/ pac/EDGSim/.*, 2003.

[42] Y. Dai, Y. Pan, and X. Zou. A hierarchical modeling and analysis for grid service reliability. *IEEE Trans. Computers*, 56(5) :681–691, 2007.

[43] Y. Dai, M. Xie, and K. Poh. Reliability of grid service systems. *Computers & Industrial Engineering*, 50(1-2) :130–147, 2006.

[44] A. Das, A. Das, I. Gupta, and A. Motivala. Swim : Scalable weakly-consistent infection-style process group membership protocol. In *In Proc. (DSN 02)*, pages 303–312, 2002.

[45] A. Dennis, B.H. Wixom, and D. Tegarden. *Systems Analysis and Design with Uml (Wie).* John Wiley & Sons, Incorporated, 2002.

[46] R. Diestel. *Graph Theory.* Springer-verlag edition, 2010.

[47] S. Djilali, T. Hérault, O. Lodygensky, T. Morlier, and G. Fedak. Rpc-v : Toward fault-tolerant rpc for internet connected desktop grids with volatile nodes. In *in Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.

[48] C. L. Dumitrescu and I. Foster. Gangsim : a simulator for grid scheduling studies. CCGRID '05, pages 1151–1158, USA, 2005. IEEE Computer Society.

[49] M. el Mehdi Diouri, O. Gluck, L. Lefevre, and F. Cappello. Energy considerations in checkpointing and fault tolerance protocols. In *Dependable Systems and Networks Workshops (DSN-W)*, pages 1–6, 2012.

[50] E. N. M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3) :375–408, 2002.

[51] C. Ernemann, V. Hamscher, U. Schwiegelshohn, et al. On advantages of grid computing for parallel job scheduling. In *(CCGrid 2002), Germany*, 2002.

[52] P. Felber, X. Défago, P. Eugster, and A. Schiper. Replicating CORBA Objects : a marriage between active and passive replication. In *(DAIS'99)*, pages 375–387, 1999.

[53] R. Fernandes, K. Pingali, and P. Stodghill. Mobile mpi programs in computational grids. PPoPP '06, pages 22–31, USA, 2006. ACM.

[54] R. M.G. Ferrari, T. Parisini, and M.M. Polycarpou. Distributed fault diagnosis of large-scale discrete-time nonlinear systems : New results on the isolation problem. In *(CDC)*, pages 1619–1626. IEEE, 2010.

[55] L. Ferreira, V. Berstis, J. Armstrong, M. Kendzierski, et al. *Introduction to grid computing with globus*. IBM Corp., USA, 2003.

[56] R. Finkel and U. Manber. Dib : a distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst.*, 9(2) :235–256, 1987.

[57] G. Florin. La tolérance aux pannes dans les systèmes répartis. Technical report, Laboratoire CEDRIC CNAM, 1996.

[58] Open Grid Forum. *https ://www.ogf.org/ogf/doku.php.*, 2006.

[59] I. Foster and C. Kesselman. Globus : A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11 :115–128, 1996.

[60] I. Foster and C. Kesselman. *The Grid 2 : Blueprint for a New Computing Infrastructure*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Elsevier Science, 2003.

[61] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid : An open grid services architecture for distributed systems integration. http ://www.globus.org/alliance/publications/papers/ogsa.pdf, 2002.

[62] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid : Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3) :200–222, 2001.

[63] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[64] Paul M. Frank. Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy—a survey and some new results. *Automatica*, 26(3) :459–474, 1990.

[65] M. Gabel, A. Schuster, R. Gilad-Bachrach, and N. Bjørner. Latent fault detection in large scale services. In *IEEE/IFIP DSN 2012*, pages 1–12. IEEE Computer Society, USA, 2012.

[66] E. Gabriel, G. E. Fagg, A. Bukovsky, T. Angskun, and J. J. Dongarra. A fault-tolerant communication library for grid environments. In *In Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS'03)*, 2003.

[67] S. Genaud. *Exécutions de programmes parallèles à passage de messages sur grille de calcul*. Habilitation á diriger des recherches, Université Henri Poincaré - Nancy 1, Dec 2009.

[68] S. Genaud, E. Jeannot, and C. Rattanapoka. Fault-management in p2p-mpi. *International Journal of Parallel Programming*, 37(5) :433–461, 2009.

[69] S. Genaud and C. Rattanapoka. P2p-mpi : A peer-to-peer framework for robust execution of message passing parallel programs on grids. *J. Grid Comput.*, 5(1) :27–42, 2007.

[70] C. Germain, J. Nauroy, and K. Rafes. The grid observatory 3.0 - towards reproducible research and open collaborations using semantic technologies. In *EGI Community Forum 2014*, Finland, 2015.

[71] C. Germain-Renaud, A. Cady, P. Gauron, M. Jouvin, C. Loomis, et al. The Grid Observatory. In IEEE Computer Society Press, editor, *11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 114–123, 2011.

[72] T. Ghafarian-M., H. Deldari, B. Javadi and, et al. Cycloidgrid : A proximity-aware p2p-based resource discovery architecture in volunteer computing systems. *Future Generation Comp. Syst.*, 29(6) :1583–1595, 2013.

[73] T. Ghafarian-M., H. Deldari, H. Mohhamad, and M.H. Yaghmaee-M. Proximity-aware resource discovery architecture in peer-to-peer based volunteer computing system. In *IEEE International Conference on Computer and Information Technology*, pages 83–90, 2011.

[74] A. Girault, C. Lavarenne, Y. Sorel, and M. Sighireanu. Fault-tolerant static scheduling for real-time distributed embedded systems. In *ICDCS*, pages 695–698, 2001.

[75] M. Godwin. The free network. *https ://freenetproject.org/*, 1996.

[76] T. Goodale, G. Allen, G. Lanfermann, et al. The cactus framework and toolkit : Design and applications. In *In Vector and Parallel Processing - VECPAR 2002, 5th International Conference*. Springer, 2003.

[77] J. Goux, S. Kulkarni, M. Yoder, and J. Linderoth. An enabling framework for master-worker applications on the computational grid. HPDC '00, USA, 2000. IEEE Computer Society.

[78] Andrew S. Grimshaw, Wm. A. Wulf, and CORPORATE The Legion Team. The legion vision of a worldwide virtual computer. *Commun. ACM*, 40(1) :39–45, 1997.

[79] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6) :789–828, 1996.

[80] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Ada-Europe*, pages 38–57, 1996.

[81] F.P. Guimaraes, P. Célestin, D.M. Batista, G.N. Rodrigues, et al. A framework for adaptive fault-tolerant execution of workflows in the grid : Empirical and theoretical analysis. *J. Grid Comput.*, 12(1) :127–151, 2014.

[82] S. Guo, H. Huang, and Y. Liu. Modeling and analysis of grid service reliability considering fault recovery. *New Generation Comput.*, 29(4) :345–364, 2011.

[83] S. Guo, H. Huang, Z. Wang, and M. Xie. Grid service reliability modeling and optimal task scheduling considering fault recovery. *IEEE Transactions on Reliability*, 60(1) :263–274, 2011.

[84] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. PODC '01, pages 170–179, USA, 2001. ACM.

[85] J. Hagel and A. G. Armstrong. *Net Gain : Expanding Markets Through Virtual Communities*. Harvard Business School Press, 1997.

[86] S. Helal, R. Bose, and W. Li. *Mobile Platforms and Development Environments*. Synthesis Lectures on Mobile and Pervasive Computing. Morgan & Claypool Publishers, 2012.

[87] Y. Horita and K. Taura. A scalable and efficient self-organizing failure detector for grid applications. In *(Grid 05)*, pages 202–210. Society Press, 2005.

[88] L. Hui, G. David, W. Lex, and T. Jeff. Job Failure Analysis and Its Implications in a Large-Scale Production Grid. *e-science*, 0 :27, 2006.

[89] S. Hwang and C. Kesselman. A flexible framework for fault tolerance in the grid. *J. Grid Comput.*, 1(3) :251–272, 2003.

[90] A. Iosup, O. Sonmez, S. Anoep, and D. Epema. The performance of bags-of-tasks in large-scale distributed systems. HPDC '08, pages 97–108, USA, 2008. ACM.

[91] S. Jafar. *Programmation des systèmes parallèles distribués : tolérance aux pannes, résilience et adaptabilité*. These, Institut National Polytechnique de Grenoble - INPG, 2006.

[92] A. Jain and R. K. Shyamasundar. Failure detection and membership management in grid environments. GRID '04, pages 44–52, USA, 2004. IEEE Computer Society.

[93] T. R. Jensen and B. Toft. *Graph coloring problems*. Wiley interscience series in discrete mathematics and optimization. John Wiley and sons, New York, 1995.

[94] G. Jiang, H. Chen, and K. Yoshihira. Discovering likely invariants of distributed transaction systems for autonomic system management. *Cluster Computing*, 9(4) :385–399, 2006.

[95] Y. Jiang and W. Chen. Task scheduling in grid computing environments. In *Genetic and Evolutionary Computing*, volume 238, pages 23–32. 2014.

[96] H. Jin, X. Shi, W. Qiang, and D. Zou. Dric : Dependable grid computing framework. *IEICE Transactions*, 89-D(2) :612–623, 2006.

[97] H. Jitsumoto, T. Endo, and S. Matsuoka. Abaris : An adaptable fault detection/recovery component framework for mpis. In *IPDPS*, pages 1–8, 2007.

[98] J. Joshy and C. Fellenstein. *Grid Computing*. Prentice Hall PTR, USA, 2003.

[99] A. Kalakech. *Etalonnage de la sûreté de fonctionnement des systèmes d'exploitation. Spécifications et mise en oeuvre*. These, Institut National Polytechnique de Toulouse - INPT, 2005.

[100] H. Kalla. *Génération automatique de distributions/ordonnancements temps réel, fiables et tolérants aux fautes*. These, Institut National Polytechnique de Grenoble - INPG, 2004.

[101] G. Kandaswamy, A. Mandal, and D.A. Reed. Fault tolerance and recovery of scientific workflows on computational grids. In *(CCGrid 2008), France*, pages 777–782, 2008.

[102] L.M. Khanli, M.E. Far, and A.M. Rahmani. Rfoh : A new fault tolerant job scheduler in grid computing. In *(ICCEA)*, volume 1, pages 422–425, 2010.

[103] T. Kohonen, M. R. Schroeder, and T. S. Huang, editors. *Self-Organizing Maps*. Springer-Verlag New York, Inc., USA, 3rd edition, 2001.

[104] G. Kola, T. Kosar, and M. Livny. Phoenix : Making data-intensive grid applications fault-tolerant. In *GRID*, pages 251–258, 2004.

[105] G. Kola, T. Kosar, and M. Livny. Faults in large distributed systems and what we can do about them. In *(Europar 2005)*, pages 442–453. Springer-Verlag, 2005.

[106] D. Kondo, G. Fedak, F. Cappello, A.A. Chien, and H. Casanova. Characterizing resource availability in enterprise desktop grids. *Future Gener. Comput. Syst.*, 23(7) :888–903, 2007.

[107] D. Kondo, B. Javadi, A. Iosup, and D. Epema. The failure trace archive : Enabling comparative analysis of failures in diverse distributed systems. CCGRID '10, pages 398–407, USA, 2010. IEEE Computer Society.

[108] J. Kovács and P. Kacsuk. A migration framework for executing parallel programs in the grid. In *AxGrids 2004*, pages 80–89, 2004.

[109] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Softw., Pract. Exper.*, 32(2) :135–164, 2002.

[110] M. Kubale. *Graph Colorings*. American Mathematical Society, contemporary mathematics edition, 2004.

[111] S. Kurkovsky, B. Bhagyavati, and A. Ray. A collaborative problem-solving framework for mobile devices. In *Proceedings of the 42nd Annual Southeast Regional Conference*, pages 5–10, USA, 2004. ACM.

[112] S. M. Lambert. Réplication et durabilité dans les systèmes répartis. Technical report, Université de Lausanne, 2001.

[113] L. Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, pages 7–9, 2002.

[114] Z. Lan and Y. Li. Adaptive fault management of parallel applications for high-performance computing. *IEEE Trans. Computers*, 57(12) :1647–1660, 2008.

[115] G. Lanfermann, G. Allen, T. Radke, and E. Seidel. Nomadic migration : Fault tolerance in a disruptive grid environment. CCGRID '02, pages 280–281, USA, 2002. IEEE Computer Society.

[116] J. C. Laprie. Dependable computing : Concepts, challenges, directions. In *COMPSAC*, 2004.

[117] J. C. Laprie, K. Kanoun, and M. Kaâniche. Modelling interdependencies between the electricity and information infrastructures. SAFECOMP'07, pages 54–67, Berlin, Heidelberg, 2007. Springer-Verlag.

[118] L.F. Lau, A.L. Ananda, G. Tan, and W.F. Wong. Gucha : Internet-based parallel computing using java. ICA3PP 2000.

[119] The lcg real time monitor. *http ://gridportal.hep. ph.ic.ac.uk/rtm/.*, 2015.

[120] C. Lee, S. Matsuoka, and D.and ithers Talia. A grid programming primer, 2001.

[121] The lhc computing grid (lcg) project. *http ://wlcg.web.cern.ch/.*, 2015.

[122] H. Li, L. Sun, and E.C. Ifeachor. Challenges of mobile ad-hoc grids and their applications in e-healthcare. In *CIMED2005*, 2005.

[123] Y. Li, Z. Lan, P. Gujrati, and X. Sun. Fault-aware runtime strategies for high-performance computing. *IEEE Trans. Parallel Distrib. Syst.*, 20(4) :460–473, 2009.

[124] J. Liang, R. Kumar, and K. W. Ross. Understanding kazaa, 2004. Polytechnic University Brooklyn, NY, USA, Submitted for publication.

[125] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Observations on the dynamic evolution of peer-to-peer networks. IPTPS '01, pages 22–33, UK, 2002. Springer-Verlag.

[126] GraphStream A Dynamic Graph Library. *http ://graphstream.sourceforge.net/.* Accessed : 30/03/2015.

[127] A. Litke, D. Skoutas, and T. Varvarigou. Mobile grid computing : Changes and challenges of resource management in a mobile grid environment. In *European Across Grids Confernece*, Spain,, 2003. Lecture Notes in Computer Science.

[128] B. Marin, M. Olivier, and S. Pierre. Performance analysis of a hierarchical failure detector. (DSN '03), pages 635–644, 2003.

[129] A. N. Mario. Peer-to-peer : harnessing the power of disruptive technologies. *SIGMOD Record*, 32(2) :57–58, 2003.

[130] M.L. Massie, B.N. Chun, and D.E. Culler. The ganglia distributed monitoring system : design, implementation, and experience. *Parallel Computing*, 30(5-6) :817–840, 2004.

[131] M. Migliardi, M. Maheswaran, B. Maniymaran, P. Card, and F. Azzedin. Mobile interfaces to computational, data, and service grid systems. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4) :71–73, 2002.

[132] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Comput. Surv.*, 32(3) :241–299, 2000.

[133] J. Mincer-Daszkiewicz. *A Service for Reliable Execution of Grid Applications.* These, Warsaw University in Warsaw, Poland, 2006.

[134] D. Mogilevsky, G. A. Koenig, and W. Yurcik. Byzantine anomaly testing for charm++ : Providing fault tolerance and survivability for charm++ empowered clusters. In *(CCGrid 2006), Singapore*, page 30, 2006.

[135] A. S. Mohd Noor, M. Mat Deris, M.Y. Bin Mohd. Saman, and E. A. Sirajudin. Distributed dynamic failure detection. *JSW*, 9(5) :1342–1347, 2014.

[136] A. Mohsin. *Conception d'une architecture journalisée tolérante aux fautes pour un processeur de données.* These, Université Paul Verlaine - Metz, 2011.

[137] S. Monnet. *Gestion des données dans les grilles de calcul : support pour la tolérance aux fautes et la cohérence des données.* Thèse de doctorat, IRISA, Rennes, France, 2006.

[138] R. S. Montero, E. Huedo, and I. M. Llorente. Grid resource selection for opportunistic job migration. In *Euro-Par 2003*, pages 366–373, 2003.

[139] C. Morin. *Architectures et systèmes distribués tolérants aux fautes.* Habilitation á diriger des recherches, Université Rennes 1, March 1998.

[140] M. Nandagopal and V.R. Uthariaraj. Fault tolerant scheduling strategy for computational grid environment. *International Journal of Engineering Science and Technology*, 2(9) :4361–4372, 2010.

[141] A. Natrajan, J. Karpovich, M. Humphrey, et al. Capacity and capability computing using legion. In *(ICCS)*, pages 273–283, 2001.

[142] M.O. Neary, S.P. Brydon, Kmiec; P.K., et al. Javelin++ : scalability issues in global computing. *Concurrency - Practice and Experience*, 12(8) :727–753, 2000.

[143] S. Olivier. *Stockage dans les systèmes pair à pair.* Phd's thesis, Université Rennes 1, France, 2005.

[144] N. Padoy. Redistribution de données entre deux grappes d'ordinateurs. Technical report, RESEDAS - INRIA Lorraine - LORIA, 2002.

[145] J. Pankaj. *Fault tolerance in distributed systems.* Prentice Hall, 1994.

[146] M. Parashar and J. M. Pierson. When the grid becomes pervasive : A vision on pervasive grids. In *(HERCMA).* LEA Publishers,Greece, 2007.

[147] D. Pelleg, M. Ben-Yehuda, R. Harper, L. Spainhower, and T. Adeshiyan. Vigilant : out-of-band detection of failures in virtual machines. *SIGOPS Oper. Syst. Rev.*, 42(1) :26–31, 2008.

[148] T. Pepper and J. Frankel. Rfc-gnutella. *http ://sourceforge.net/projects/rfc-gnutella*, 2000.

[149] T. Phan, L. Huang, and C. Dulan. Challenge : Integrating mobile wireless devices into the computational grid. MobiCom '02, pages 271–278, USA, 2002. ACM.

[150] J. M. Pierson. Data management concerns in a pervasive grid. In *VECPAR*, pages 506–520, 2008.

[151] Y. Pigné. *Modélisation et Traitement Décentralisé des Graphes Dynamiques - Application Aux Réseaux Mobiles Ad Hoc*. Phd thesis, L'Université du Havre, 2008.

[152] E. Pitoura and B. Bhargava. Building information systems for mobile environments. CIKM '94, pages 371–378, USA, 1994. ACM.

[153] J. Postel and J. Reynolds. File transfer protocol (ftp), 1985.

[154] R. Presuhn. Version 2 of the protocol operations for the simple network management protocol (snmp). Technical report, USA, 2002.

[155] L. Qian-mu, X. Man-wu, and Z. Hong. A root-fault detection system of grid based on immunology. GCC '06, pages 369–373, USA, 2006. IEEE Computer Society.

[156] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogeneous architectures. In *In Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.

[157] B. Randell. System structure for software fault tolerance. In *IEEE Transactions on Software Engineering*, volume SE-1, pages 220–232, 1975.

[158] M. Rebbah. Tolérance aux pannes dans les grilles de calcul. In *Journées d'études Grille de calcul et Intelligence Artificielle, GCIA06*, Université de Mascara, Algérie, 2006.

[159] M. Rebbah, C. Mokhtari, M.F. Khaldi, M.and Bourassi, and O. Smail. Hierarchical model for fault tolerant grid computing over globus toolkit. In *International Congress on Models, Optimization and Security of Systems (ICMOSS2010)*, 2010.

[160] M. Rebbah, Y. Slimani, and A. Bakbak. Multi-tree model for fault tolerant mobile grid. *Asian Journal of Applied Sciences*, 4(2) :155–165, 2011.

[161] M. Rebbah, Y. Slimani, and A. Benyettou. A decentralized fault tolerant model for grid computing. *IJCSI International Journal of Computer Science Issues*, 11(1) :123–130, January 2014.

[162] M. Rebbah, Y. Slimani, and A. Benyettou. Decentralized fault tolerant model for p2p grid. In *International Conference on Artificial Intelligence and Information Technology (ICA2IT'14)*, 2014.

[163] M. Rebbah, Y. Slimani, A. Benyettou, and L. Brunie. Dynamic hierarchical model for fault tolerant grid computing. *World Applied Programming Journal*, 1(5) :309–321, December 2011.

[164] M. Rebbah, Y. Slimani, A. Benyettou, and L. Brunie. Modèle décentralisé pour la tolérance aux fautes dans les grilles de calcul. In 6me *Conférence francophone sur les architectures logicielles, CAL 2012,*, Montpellier, France, 2012.

[165] M. Rebbah, Y. Slimani, A. Benyettou, and L. Brunie. Reliable fault tolerant model for grid computing environments. *Multiagent and Grid Systems*, 10(4) :213–232, 2014.

[166] X. Ren, R. Eigenmann, and S. Bagchi. Failure-aware checkpointing in fine-grained cycle sharing systems. HPDC '07, pages 33–42, USA, 2007. ACM.

[167] T. Röblitz, F. Schintke, A. Reinefeld, O. Bärring, et al. Autonomic management of large clusters and their integration into the grid. *J. Grid Comput.*, 2(3) :247–260, 2004.

[168] Antony I. T. Rowstron and P. Druschel. Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. Middleware '01, pages 329–350, UK, 2001. Springer-Verlag.

[169] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, et al. Critical event prediction for proactive management in large-scale computer clusters. KDD '03, pages 426–435, USA, 2003. ACM.

[170] J. Salmon, C. Stein, and T. Sterling. Scaling of beowulf-class distributed systems. Supercomputing '98, pages 1–13, USA, 1998. IEEE Computer Society.

[171] P. Sang-Min, K. Young-Bae, and K. Jai-Hoon. Disconnected operation service in mobile grid computing. In *ICSOC 2003,*, pages 499–513, 2003.

[172] S. Sankaran, J. M. Squyres, M. Barrett, and A. Lumsdaine. The lam/mpi checkpoint/restart framework : System-initiated checkpointing. In *in Proceedings, LACSI Symposium, Sante Fe*, pages 479–493, 2003.

[173] L.F.G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18 :561–572, 2002.

[174] S. Saroiu, P.K. Gummadi, and S.D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Syst.*, 9(2) :170–184, 2003.

[175] K. Sato, A. Moody, K. Mohror, et al. Fmi : Fault tolerant messaging interface for fast and transparent recovery. In *28th International Parallel and Distributed Processing Symposium, 2014 IEEE*, pages 1225–1234, 2014.

[176] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2 : Patterns for Concurrent and Networked Objects.* Wiley, 2000.

[177] C. Schneider, A. Barker, and S. Dobson. Autonomous fault detection in self-healing systems : Comparing hidden markov models and artificial neural networks. ADAPT '14, pages 24–31, USA, 2014. ACM.

[178] C. Schneider, A. Barker, and S. Dobson. Autonomous fault detection in self-healing systems using restricted boltzmann machines. *CoRR*, abs/1501.01501, 2015.

[179] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. P2P '01, page 101, USA, 2001. IEEE Computer Society.

[180] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. DSN '06, pages 249–258, USA, 2006. IEEE Computer Society.

[181] H. Shan and L. Oliker. Scheduling in heterogeneous grid environments : The effects of data migration. In *ADCOM2004*, 2004.

[182] AB. Sharma, H. Chen, M. Ding, K. Yoshihira, and G. Jiang. Fault detection and localization in distributed systems using invariant relationships. In *2013 (DSN), Hungary*, pages 1–8, 2013.

[183] X. Shen, H. Yu, J. Buford, and M. Akon. *Handbook of Peer-to-Peer Networking.* Springer Publishing Company, Incorporated, 2009.

[184] S. Shukla and A. Deshpande. Ldap directory services- just another database application ? (tutorial session). *SIGMOD Rec.*, 29(2) :580, 2000.

[185] D. Siewiorek and R. Swarz. *Reliable Computer Systems : Design and Evaluation.* Digital Press, 1992.

[186] S. H. Srinivasan. Pervasive wireless grid architecture. In *(WONS 2005)*, pages 83–88, 2005.

[187] P. Stelling, C. DeMatteis, I. Foster, C. Kesselman, L. Craig, and G. von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2) :117–128, 1999.

[188] R. Stewart, P. Trinder, and P. Maier. Supervised workpools for reliable massively parallel computing. In *Trends in Functional Programming*, volume 7829, pages 247–262. Springer Berlin Heidelberg, 2013.

[189] E. Strohmaier and J. D. Meuer. Top500 supercomputer sites. Technical report, University of Tennessee, USA, 1997.

[190] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3 :204–226, 1985.

[191] X.-H. Sun, Z. Lan, Y. Li, H. Jin, and Z. Zheng. Towards a fault-aware computing environment. In *Proceedings of the High Availability and Performance Computing Workshop (HAPCW)*, 2008.

[192] A. Tanenbaum. *Systèmes d'exploitation. Systèmes centralisés, systèmes distribués.* Dunod Informatique, dunod edition, 1999.

[193] L. Temal and D. Conan. Détections de défaillances, de connectivité et de déconnexions. In *Actes de la 1ère Conférence Francophone Mobilité et Ubiquité.*, pages 90–97, 2004.

[194] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In Fran Berman, Anthony Hey, and Geoffrey Fox, editors, *Grid Computing : Making the Global Infrastructure a Reality.* John Wiley, 2003.

[195] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. A grid monitoring architecture, 2002.

[196] O. Togni. *Coloration de graphes : quelques contributions.* Habilitation à diriger des recherches, université de Bourgogne, 2006.

[197] P. Townend, P. Groth, N. Looker, and J. Xu. Ft-grid : A fault-tolerance system for e-science. In *AHM05*, 2005.

[198] P. Townend, P. Townend, and J. Xu. Fault tolerance within a grid environment. In *In Proceedings of the UK e-Science All Hands Meeting 2003*, pages 272–275, 2003.

[199] F. Tronel. *Application des problèmes d'accord à la tolérance aux défaillances dans les systèmes distribués asynchrones.* PhD thesis, 2003.

[200] S. Tuecke, K. Czajkowski, I. Foster, et al. Open Grid Services Infrastructure (OGSI). http ://www.globus.org/alliance/publications/papers/ Final_OGSI_Specification_V1.0.pdf, 2003.

[201] J. D. Ullman and J. Widom. *A first course in database systems (3. ed.)*. Prentice Hall, 2007.

[202] S. Vadhiyar and J. Dongarra. A performance oriented migration framework for the grid. In *(CCGrid 2003)*, pages 130–137. IEEE Computer Society, 2003.

[203] S. S. Vadhiyar and J.J. Dongarra. Srs - a framework for developing malleable and migratable parallel applications for distributed systems. In *In : Parallel Processing Letters.*, pages 291–312, 2002.

[204] L. Valcarenghi and P. Castoldi. Qos-aware connection resilience for network-aware grid computing fault tolerance. In *In : Intl. Conf. on Transparent Optical Networks*, pages 417–422, 2005.

[205] P. D. V. van der Stok, M. M. M. P. J. Claessen, and D. Alstein. A hierarchical membership protocol for synchronous distributed systems. In *(EDCC-1)*, volume 852 of *Lecture Notes in Computer Science*, pages 599–616. Springer, 1994.

[206] R. Van Nieuwpoort, J. Maassen, T. Kielmann, and H.E. Bal. Satin : Simple and efficient java-based grid programming. *Scalable Computing : Practice and Experience*, 6(3), 2005.

[207] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Middleware '98, pages 55–70, UK, 1998. Springer-Verlag.

[208] V. Venkatasubramanian, R. Rengaswamy, K. Yin, and S. N. Kavuri. A review of process fault detection and diagnosis : Part i : Quantitative model-based methods. *Computers and Chemical Engineering*, 27(3) :293–311, 2003.

[209] D.C Verma, S. Sahu, S.B. Calo, A. Shaikh, et al. Sriram : A scalable resilient autonomic mesh. *IBM Systems Journal*, 42(1) :19–28, 2003.

[210] M. Voelter, M. Kircher, and Z. Uwe. *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware.* J. Wiley & Sons, Hoboken, NJ, USA, October 2004.

[211] Q.H. Vu, M. Lupu, and B.C. Ooi. *Peer-to-Peer Computing - Principles and Applications.* Springer, 2010.

[212] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in hpc environments. SC '08, pages 43 :1–43 :12, USA, 2008. IEEE Press.

[213] X. Wang, Y. Zhuang, and H. Hou. Byzantine fault tolerance in mds of grid system. In *Machine Learning and Cybernetics*, 2006.

[214] P. Watson. Databases and the grid. Technical report, Grid Computing : Making The Global Infrastructure a Reality, 2001.

[215] P. Watson. Databases and the grid. Technical report, Grid Computing : Making The Global Infrastructure a Reality, 2002.

[216] J.B. Weissman and B. Lee. The virtual service grid : An architecture for delivering high-end network services. concurrency : Practice and experience. In *In Concurrency : Practice and Experience*, pages 287–319, 2002.

[217] N. Woo, S. Choi, H. Jung, et al. Mpich-gf : Providing fault tolerance on grid environments. In *On Cluster Computing and the Grid (CC-Grid2003)*, 2003.

[218] N. Woo, H.Y. Yeom, and T. Park. Mpich-gf : Transparent checkpointing and rollback-recovery for grid-enabled mpi processes. *IEICE TRANSACTIONS on Information and Systems*, 87 :1820–1828, 2004.

[219] G. Wrzesinska, R. V. Van Nieuwpoort, J. Maassen, and H.E. Bal. Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid. In *In Proc. of 19th International Parallel and Distributed Processing Symposium*, 2005.

[220] Z. Han W. Qiang S. Wu D. Zou X. Shi, H. Jin. Alter : Adaptive failure detection services for grids. In *IEEE International Conference on Services Computing (SCC'05)*, pages 355–358, 2005.

[221] B. Yagoubi and Y. Slimani. Dynamic load balancing strategy for grid computing. *Int.Journ. of transaction on Engineering, Computing and technology*, 13, 2006.

[222] C.S. Yeo, B. Buyya, H. Pourreza, R. Eskicioglu, P. Graham, and P. Sommers. Cluster computing : High-performance, high-availability, and high-throughput processing on a network of computers. In *Handbook of Nature-Inspired and Innovative Computing : Integrating Classical Models with Emerging Technologies, chapter 16*, pages 521–551, 2006.

[223] V.C. Zandy, B.P. Miller, and M. Livny. Process hijacking. In *Proceedings of the Eighth International Symposium on High Performance Distributed Computing*, 1999.

[224] S. Zanikolas and R. Sakellariou. A taxonomy of grid monitoring systems. *Future Generation Comp. Syst.*, 21(1) :163–188, 2005.

[225] X. Zhang, F. Junqueira, M.A. Hiltunen, et al. Replicating nondeterministic services on grid environments. In *HPDC-15, France, 2006*, pages 105–116. IEEE, 2006.

[226] X. Zhang, D. Zagorodnov, M. Hiltunen, et al. Fault-tolerant grid services using primary-backup : Feasibility and performance. In *IEEE International Conference on Cluster Computing*, pages 105–114, 2004.