

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Réseaux, Télécommunications, Système et Architecture

Présentée et soutenue par :
Rémi SHARROCK

le : mercredi 8 décembre 2010

Titre :

Gestion autonomique de performance, d'énergie et de qualité de service.
Application aux réseaux filaires, réseaux de capteurs et grilles de calcul.

Ecole doctorale :
Mathématiques Informatique Télécommunications (MITT)

Unité de recherche :
LAAS - CNRS (UPR 8001) et IRIT (UMR 5505)

Directeur(s) de Thèse :

Thierry MONTEIL (LAAS-CNRS) et Patricia STOLF (IRIT)

Rapporteurs :

Pascal BOUVRY (Université de Luxembourg) et Christian PEREZ (ENS Lyon)

Autre(s) membre(s) du jury

Levent GURGEN (ENSIMAG Grenoble), Daniel HAGIMONT (ENSEEIHT Toulouse), Emmanuel JEANNOT (INRIA Bordeaux)

Pour Baba,

Remerciements

Mes remerciements vont tout naturellement aux directeurs des laboratoires qui m'ont accueilli pendant mes trois années de thèse : Raja Chatila pour le LAAS-CNRS et Luis Fariñas del cerro pour l'IRIT. C'est à l'origine Thierry Monteil, mon co-directeur de thèse qui a monté un projet auprès de la région Midi-Pyrénées pour le financement de mes trois années de thèse, c'est donc avec enthousiasme que je remercie Thierry et la région Midi-Pyrénées pour leur soutien financier. Je voudrais noter l'importance de ce financement régional, notamment par la fourniture des équipements réseaux très onéreux, et je les invite à renouveler autant que possible ce soutient qui est très important pour faire avancer la recherche régionale. Mes remerciement vont également à Patricia Stolf, ma co-directrice de thèse et de nouveau à Thierry Monteil pour leur encadrement. Pendant ces trois années, ils m'ont initié à la rigueur scientifique et m'ont réellement transformé que ce soit au niveau de la mise à plat des problèmes, de l'écriture scientifique ou de la communication scientifique. Merci à eux pour leur soutien et leur persévérance. Je remercie Pascal Bouvry et Christian Perez d'avoir rapporté ma thèse, leur nouveau regard a soulevé des questions pertinentes et ils m'ont apporté de nombreux conseils dans la phase finale d'élaboration de ma thèse. Merci également à Daniel Hagimont d'avoir présidé mon jury de thèse ainsi qu'à Emmanuel Jeannot et Levent Gürgen pour leur participation au jury de thèse. Merci aussi aux directeurs d'équipes de m'avoir accueilli et aux personnes de l'administration pour chacune de ces équipes : André Monin et Olivier Brun pour l'équipe MRS du LAAS-CNRS, Zoubir Mammeri et Jean-Marc Pierson pour l'équipe ASTRE de l'IRIT, Shinichi Honiden pour l'équipe du NII à Tokyo et Raymond Namyst pour l'équipe Runtime de l'INRIA Bordeaux.

Je voudrais ensuite remercier les personnes qui m'ont entourées pendant ma thèse, en commençant par mes collègues de travail à Toulouse, au Japon et à Bordeaux. Pour la ville rose : Laurent Broto, Georges da costa, Aeiman Gadafi avec qui nous avons travaillé à l'IRIT, Fadi Khalil Fabio Coccetti Hervé Aubert pour le LAAS, Petr Lorenz pour la société Lorenz Solution basée à Munich mais qui s'est déplacé, Phillipe Combes et toute l'équipe du CICT (Centre Interuniversitaire de Calcul de Toulouse). Pour Tokyo et ses environs : Shinichi Honiden, Akiko Shimazu, Amin Cherbal et bien sûr Michel Dayde et Henri Angelino qui ont pu amorcer ce projet international, en espérant que les relations entre Tokyo et Toulouse s'intensifierons dans l'avenir. Merci aussi aux fabuleux stagiaires : Andreea Simona Anghel, Arnaud Hidoux, Cyril Fantin, Grégoire Linselle, Tarek Saïs et Tom Guérout ; aux co-bureaux et je l'espère, futurs docteurs Ahmad al Asheikh, Jean-Marie Codol, Aurélien Gonzalez, David Flavigné et Mihai Alexandru.

Je terminerai par une mention spéciale à mes proches, famille et amis. Sans eux je n'aurais pas eu le plaisir de répéter ma soutenance et de connaitre le titre de ma thèse par coeur. Après ce dur labeur, me revoilà parmi vous !

Table des matières

1 Introduction **1**
- 1.1 Motivation principale : la gestion de la complexité grandissante 2
  - 1.1.1 Le compliqué, le complexe et le simple 2
  - 1.1.2 Histoire de simplifications 2
  - 1.1.3 La dualité simplicité - complexité 3
- 1.2 Problématiques soulevées par la thèse 4
  - 1.2.1 Les défis de la complexité grandissante 4
  - 1.2.2 Les objectifs de la thèse : l'auto-gestion - "Self-management" . . . 5
  - 1.2.3 L'auto-gestion - "Self-management" 5
  - 1.2.4 Le besoin de généricité 6
- 1.3 Vue globale de la thèse 7

2 État de l'art **9**
- 2.1 L'informatique autonomique ou "Autonomic Computing" 10
  - 2.1.1 Automatique, Autonome et Autonomique 10
    - 2.1.1.1 Différence entre automatique et autonome 10
    - 2.1.1.2 Différence entre autonome et autonomique 10
  - 2.1.2 La vision de l'informatique autonomique : historique et inspiration biologique 11
  - 2.1.3 La gestion autonomique 12
- 2.2 La boucle de contrôle autonomique 13
  - 2.2.1 La boucle MAPE-K 13
    - 2.2.1.1 Le gestionnaire autonomique 14
    - 2.2.1.2 L'observation ou "monitoring" 14
    - 2.2.1.3 L'analyse 15
    - 2.2.1.4 La planification 15
    - 2.2.1.5 L'exécution 15
    - 2.2.1.6 La base de connaissances 15
- 2.3 Contexte de l'étude et définitions 16
  - 2.3.1 Grappes de machines ou "Clusters" 16
  - 2.3.2 Grilles informatiques 16
  - 2.3.3 Centres de calcul et de traitement des données "Data centers" et nuages informatiques "clouds" 17
  - 2.3.4 Réseaux de capteurs 19
    - 2.3.4.1 Exemple : les SunSPOTs 19
  - 2.3.5 L'intergiciel XSStreaMWare de gestion des réseaux de capteur . . 20

2.3.5.1 Présentation globale d'XSStreaMWare 20
2.3.5.2 Hiérarchie du modèle générique XSStreaMWare : le DATAMODEL 21
2.3.6 Les logiciels patrimoniaux existants 22
2.3.7 L'application DIET 22
2.3.8 Les applications de simulation électromagnétiques 24
2.3.8.1 Les sciences CEM : TLM et SCT 24
2.3.8.2 L'application YatPac 24
2.4 Positionnement par rapport à l'existant 25
2.4.1 Degré d'autonomicité et degré de généricité 25
2.4.2 Exemples d'approches existantes 27
2.4.2.1 Niveau 1 : Gestion intégrée spécifique à une application 27
2.4.2.2 Niveau 2 : Gestion appliquée à un domaine de compétences 29
2.4.2.3 Niveau 3 : Gestion générique par API avec modification des logiciels 34
2.4.2.4 Niveau 4 : Gestion générique sans modification des logiciels patrimoniaux 35
2.4.2.4.1 Le concept d'encapsulation, modèle à composants Fractal 35
2.4.2.4.2 DeployWare 36
2.4.2.4.3 Adage 37
2.4.2.4.4 JADE 37
2.5 Présentation du système pour l'intégration des contributions 38
2.5.1 Le système de gestion autonomique TUNe : Toulouse University Network 38
2.5.2 Les phases d'administration autonomique dans TUNe 39
2.5.3 Vocabulaire du triangle TUNe et fonctionnement global 40
2.5.4 Etapes de construction d'une gestion autonomique avec TUNe . . 41
2.5.4.1 1ère étape : description de l'infrastructure matérielle . . 41
2.5.4.2 2ème étape : description de l'infrastructure logicielle . . 42
2.5.4.3 3ème étape : définition de l'encapsulation des logiciels . 44
2.5.4.4 4ème étape : définition des règles de configuration 45
2.5.4.5 Vue globale de la construction d'une gestion autonomique avec TUNe 46
2.5.4.6 Introduction d'une dénomination 47
2.6 Etat de l'art : synthèse 48

3 Auto-guérison - "Self-healing" 49
3.1 Introduction 49
3.2 Méta-modélisation des diagrammes de spécification de politiques de gestion - "Policy Description Diagrams" (PDD) 50
3.2.1 Forme MOF "Meta-Object Facility" et forme EBNF "étendue de Backus-Naur" 50
3.2.2 Méta-modèle des PDD 51
3.2.2.1 Méta-modèle de la partie graphique sous forme MOF . . 51
3.2.2.1.1 Le stéréotype Action 51

3.2.2.1.2 Le stéréotype Control . . . 53
3.2.2.2 Méta-modéle de la partie syntaxique sous forme EBNF . 54
3.3 Validation . . . 60
3.3.1 Description des expériences . . . 60
3.3.1.1 Cadre matériel de l'expérience . . . 60
3.3.1.2 Cadre logiciel de l'expérience . . . 61
3.3.1.2.1 Expérience sans agrégation des événements . . . 62
3.3.1.2.2 Expérience avec agrégation des événements . . . 62
3.3.2 PDD utilisés pour les expériences . . . 63
3.3.2.1 PDD utilisés pour la phase de démarrage . . . 63
3.3.2.2 PDD utilisés pour les reconfigurations durant la phase de gestion . . . 64
3.3.2.2.1 PDD pour le "self-healing" des LA . . . 64
3.3.2.2.2 PDD pour le "self-healing" des SEDs . . . 66
3.3.2.3 Temps de réparation du cluster . . . 66
3.4 Conclusion . . . 68

4 Auto-configuration - "Self-configuring" 71
4.1 Introduction et problématique . . . 71
4.1.1 Rappels pour l'auto-configuration . . . 71
4.1.2 Problématique de "self-configuring" des réseaux de capteurs . . . 72
4.1.2.1 Problématique générale . . . 72
4.1.2.2 Problème de gestion manuelle des réseaux de capteurs avec XSStreaMWare . . . 73
4.2 Contributions pour la propriété de "self-configuring" . . . 74
4.2.1 Rajout de la propriété de "self-configuring" pour l'intergiciel XSStreaMWare . . . 74
4.2.1.1 Intégration du moteur de déchiffrage XMI en tant que service ECA . . . 74
4.2.1.2 Architecture globale d'un service ECA . . . 75
4.2.1.3 Cycle ECA . . . 75
4.2.1.4 Types d'événements . . . 76
4.2.1.5 Modélisation de politiques pour la gestion des résaux de capteurs . . . 77
4.2.2 Validation et exemples de politiques de gestion de réseaux de capteurs 79
4.2.2.1 "Self-configuring" des micrologiels ou "firmwares" des capteurs . . . 79
4.2.2.2 "Self-configuring" de paramètres internes des capteurs . . 80
4.3 Conclusion pour l'étude de cas de "self-configuring" appliqué aux réseaux de capteurs . . . 81
4.4 Problématique de "Self-configuring" pour une simulation électromagnétique sur grille . . . 83
4.4.1 Simulation électromagnétique et grille de calcul . . . 84
4.4.1.1 Flot de travail et parallélisation de la simulation . . . 84
4.4.1.2 HDD de la simulation . . . 85
4.4.1.3 SDD de la simulation . . . 86

4.4.1.4 PDD de la simulation . . . 87
4.4.1.5 SWD de la simulation . . . 88
4.4.2 Validation de l'approche de "self-configuring" pour YatPac sur Grid'5000 89
4.4.2.1 Conditions d'expérimentations . . . 89
4.4.2.2 Déploiement . . . 90
4.4.2.3 Démarrage de la simulation . . . 92
4.4.2.4 Arrêt de la simulation . . . 93
4.4.3 Conclusion pour l'étude de cas de "self-configuring" appliqué aux simulations électromagnétiques sur grille de calcul . . . 93
4.5 Conclusion et perspectives . . . 94

5 Auto-optimisation - "Self-optimizing" 95
5.1 Introduction . . . 95
5.2 "Self-optimizing" appliqué aux reconfigurations matérielles . . . 96
5.2.1 Rappel sur la Qualité de Service au niveau logiciel . . . 97
5.2.1.1 Différence entre profil de QoS et caractéristique de QoS. 97
5.2.1.2 Expression de la Qualité de Service . . . 97
5.2.1.3 Echelle de temps et granularité de contrôle . . . 98
5.2.1.4 Capacité de gestion QoS des applications . . . 99
5.2.2 Environnement considéré par l'approche . . . 100
5.2.2.1 Le domaine DiffServ . . . 100
5.2.2.2 Le routage du réseau . . . 100
5.3 Modèle mathématique pour la gestion autonomique de l'énergie et de la qualité de service des réseaux filaires. . . . 101
5.3.1 Formalisation pour la partie réseau . . . 102
5.3.1.1 Noeuds et représentation du réseau par un graphe . . . 102
5.3.1.2 Ports, modules et châssis des routeurs . . . 102
5.3.1.3 Expression de la puissance globale . . . 104
5.3.2 Formalisation pour la partie application . . . 104
5.3.2.1 L'ensemble des applications du "datacenter" . . . 104
5.3.2.2 Les besoins en caractéristiques QoS des applications . . 104
5.3.2.3 Expression de la perte de qualité de service . . . 105
5.3.2.4 Propagation des flots des applications . . . 106
5.3.3 Formalisation du critère global de minimisation . . . 107
5.3.4 Validation et expériences . . . 108
5.3.4.1 Problème d'optimisation . . . 108
5.3.4.2 Contexte de résolution . . . 109
5.3.4.3 Résolution dans le cas optimal . . . 110
5.3.4.4 Résolution dans le cas de l'utilisation de l'heuristique . . 112
5.3.5 Comparaison du cas optimal et de l'utilisation de l'heuristique . . 114
5.4 Conclusions du chapitre "self-optimizing" . . . 117

6 Auto-protection - "Self-protecting" 119
6.1 Introduction de la problématique . . . 119
6.1.1 Cohérence, stabilité et contraintes ACID . . . 120
6.1.2 Règles de cohérence internes à TUNe . . . 121

6.1.2.1 Règles de cohérence au SDD . . . 121
6.2 Solutions de récupération de la cohérence . . . 122
6.2.1 Vérification des liaisons utilisées par les SWD et création automatique de liaisons en cours d'exécution d'un PDD . . . 123
6.2.2 Vérification de cohérence des créations automatiques de liaisons et mécanisme de création automatique d'instances en cours d'exécution d'un PDD . . . 125
6.2.3 Interruption automatique en cours d'exécution d'un PDD . . . 127
6.2.4 Vérifications de cohérences finales et procédures de résolution d'incohérences ou d'annulation . . . 129
6.2.5 Vue globale de la gestion des cohérences . . . 130
6.3 Mise en oeuvre du "self-protecting" pour l'architecture DIET . . . 132
6.3.1 Présentation du cadre de mise en oeuvre . . . 132
6.3.1.1 PDD de gestion globale de l'architecture DIET . . . 132
6.3.1.2 Politique de création de nouvelles instances de SEDs . . . 134
6.4 Conclusion du chapitre "self-protecting" . . . 136
6.4.1 Contributions au "self-protecting" des systèmes autonomiques . . . 136
6.4.2 Limites de notre approche . . . 137

7 Conclusions et perspectives 139
7.1 Conclusions . . . 139
7.2 Perspectives . . . 142
7.2.1 Poursuite de l'intégration des contributions au gestionnaire autonomique TUNe. . . . 142
7.2.2 Gestion avancée des événements . . . 143
7.2.3 Vers une gestion autonomique complète des "datacenters" . . . 144

A Fractal, Deployware et Jade 147
A.1 Le modèle à composants Fractal . . . 147
A.2 DeployWare . . . 149
A.3 JADE . . . 149

B Utilisation de la programmation linéaire 153
B.1 Construction d'un Problème Linéaire (PL) avec JOpt . . . 153
B.2 Nommage des variables et des constantes . . . 153
B.3 Algorithmes pour JOpt . . . 154

C DIET et TUNe 155
C.1 SWDL pour DIET . . . 155
C.2 PDD utilisés pour la destruction des SEDs . . . 156
C.3 Expérience d'équilibrage de charge numéraire pour DIET . . . 156
C.4 Expérience d'équilibrage de charge contraint pour DIET . . . 157

D Expériences de mesure de consommation électrique des routeurs de la plateforme Grid'Mip 161

Table des figures

2.1 Boucle de contrôle autonomique MAPE-K . . . 14
2.2 Architecture hiérarchique de XSStreaMWare . . . 20
2.3 Hiérarchie du modèle générique XSStreaMWare : le DATAMODEL . . . 21
2.4 Architecture de DIET . . . 23
2.5 Placement des approches de gestion suivant le degré d'autonomicité et leur généricité . . . 27
2.6 Couches SR et patrimoniale dans TUNe . . . 39
2.7 Vocabulaire du triangle TUNe . . . 40
2.8 Diagramme simple de description d'une infrastructure matérielle dans TUNe 41
2.9 Diagramme simple de description d'une infrastructure logicielle dans TUNe 42
2.10 Exemple de wrapper générique décrit avec le WDL . . . 43
2.11 Exemple de squelette de méthode java utilisé pour l'encapsulation . . . 44
2.12 Exemple de diagramme de démarrage . . . 45
2.13 Vue globale de la construction d'une gestion autonomique avec TUNe . . . 46

3.1 Méta-modèle au format MOF des PDD . . . 52
3.2 Représentation des deux catégories de référence aux PDDs . . . 57
3.3 HDD utilisé pour les expériences . . . 61
3.4 SDD utilisé pour le déploiement de DIET, 1 probe par SED . . . 61
3.5 SDD utilisé pour le déploiement de DIET, 1 probe d'agrégation pour les SEDs . . . 62
3.6 PDD utilisé pour le démarrage de l'architecture DIET . . . 63
3.7 PDD utilisé pour la récupération des PIDs . . . 64
3.8 PDD utilisé pour la réparation d'un LA . . . 65
3.9 PDD utilisé pour la réparation des SEDs . . . 66
3.10 Détail du temps de réparation du LA . . . 67
3.11 Comparaison du temps d'exécution du PDD de réparation des SEDs . . . 67
3.12 Comparaison du temps de réparation du cluster . . . 68

4.1 Intégration de services ECA dans l'architecture hiérarchique de XSStreaMWare . . . 74
4.2 Architecture d'un service ECA . . . 75
4.3 PDD pour le "self-configuring" des capteurs SunSPOT : mise à jour automatique . . . 79
4.4 PDD pour le "self-configuring" de paramètres internes d'un capteur . . . 81
4.5 Flot de travail pour la simulation YatPac . . . 84
4.6 HDD pour la simulation YatPac . . . 86

4.7 SDD pour la simulation YatPac . . . 86
4.8 PDD pour la simulation YatPac . . . 87
4.9 SWDL utilisés pour la simulation YatPac . . . 88
4.10 Temps de création du SR interne à TUNe . . . 91
4.11 Temps de déploiement de la simulation yatpac sur Grid'5000 . . . 91
4.12 Temps pour démarrer la simulation yatpac sur les noeuds Grid'5000 . . . 92
4.13 Temps pour arrêter et nettoyer la simulation yatpac sur les noeuds Grid'5000 93

5.1 Topologie de l'architecture Grid'MIP . . . 109
5.2 Puissance en fonction du nombre d'applications et d'α dans le cas de self-optimization optimal . . . 111
5.3 Qualité de service perdue en fonction du nombre d'applications et d'α dans le cas de self-optimization optimal . . . 112
5.4 Puissance en fonction du nombre d'applications et d'α dans le cas de routage OSPF avec utilisation de l'heuristique . . . 115
5.5 Qualité de service perdue en fonction du nombre d'applications et d'α dans le cas de routage OSPF avec utilisation de l'heuristique . . . 115
5.6 Temps de résolution pour le cas optimal . . . 116
5.7 Temps de résolution pour le cas de l'heuristique . . . 116

6.1 SDD pour la création automatique de liaisons en cours d'exécution d'un PDD . . . 123
6.2 SWDL utilisé pour la classe B . . . 123
6.3 PDD avec les actions manuelles - démonstration de la création automatique de liaisons . . . 124
6.4 PDD avec les actions manuelles et automatiques - démonstration de la création automatique de liaisons . . . 124
6.5 PDD avec les actions manuelles - démonstration de la création automatique d'instances . . . 125
6.6 PDD avec les actions manuelles et automatiques - démonstration de la création automatique d'instances . . . 127
6.7 PDD avec les actions manuelles - démonstration de l'interruption automatique . . . 128
6.8 PDD avec les actions automatiques - démonstration de l'interruption automatique . . . 128
6.9 Tableaux de transformation des actions en actions d'annulation. A gauche pour les actions de modification structurelle, à droite pour les appels de méthode SWD. . . . 129
6.10 Vue globale de la résolution d'incohérences . . . 131
6.11 SDD simplifié de DIET pour le self-protecting . . . 133
6.12 PDD global pour l'architecture DIET. . . . 133
6.13 PDD avec les actions automatiques en gris pour la création d'un nouveau SED . . . 134
6.14 Résultat de l'expérience pour (a) gauche : la création de nouvelles instances de SEDs (b) droite : la destructions d'instances de SEDs. . . . 136

A.1 Modèle à composants Fractal : composant client/serveur . . . 148

A.2 Transformation d'un fichier Fractal en langage DeployWare 149
A.3 Architecture de JADE 149
A.4 Architecture de JADE 150

C.1 SWDL utilisé pour un SED 155
C.2 PDD de destruction des SEDs 156
C.3 Equilibrage de charge numéraire (a) gauche : Local Agent 1, machine puissante, (b) droite : Local Agent 2, machine lente. 157
C.4 Equilibrage de charge contraint (a) gauche : Local Agent 1, machine puissante, (b) droite : Local Agent 2, machine lente. 158
C.5 PDD avec les actions automatiques pour la création d'un nouveau LA . . 158
C.6 PDD avec les actions automatiques pour la création d'un nouveau MA . 159

Liste des tableaux

3.1 Cardinalités en fonction du type de noeud 53
3.2 Caractères utilisés pour la méta-modélisation EBNF 55
3.3 Paramètres, références et appels de méthodes en EBNF 56
3.4 Modifications structurelles en EBNF 58
3.5 Modifications et sélections de composants en EBNF 59
3.6 Noeuds de contrôle en EBNF 60

4.1 Attributs des événements de gestion 76
4.2 Expressions des opérations XSStreaMWare en EBNF 78
4.3 Expression d'accès au DATAMODEL en EBNF 78
4.4 Localisation des noeuds pour les expériences 90

Glossaire des abréviations

DS, domaine DS DiffServ, domaine DiffServ
EBNF forme étendue de Backus-Naur
ECA Evénement Condition Action
HDD Hardware Description Diagram
MAPE_K Monitoring Analyzing Planning Executing - Knowledge
ME Management Event
MOF Meta-Object Facility
PDD Policy Description Diagram
SDD Software Description Diagram
SE System Event
SR System Representation
SWD Software Wrapper Description
SWDL Software Wrapper Description Language

Chapitre 1

Introduction

Sommaire

1.1 Motivation principale : la gestion de la complexité grandissante 2

1.1.1 Le compliqué, le complexe et le simple 2

1.1.2 Histoire de simplifications 2

1.1.3 La dualité simplicité - complexité 3

1.2 Problématiques soulevées par la thèse 4

1.2.1 Les défis de la complexité grandissante 4

1.2.2 Les objectifs de la thèse : l'auto-gestion - "Self-management" . 5

1.2.3 L'auto-gestion - "Self-management" 5

1.2.4 Le besoin de généricité 6

1.3 Vue globale de la thèse 7

Les contributions de cette thèse s'inscrivent dans la continuité des travaux effectués au sein de l'équipe ASTRE [1] de l'IRIT [2], et notamment de la thèse de Laurent Broto, en faisant intervenir de nouvelles problématiques de performances issues de l'équipe MRS [3] du LAAS-CNRS [4].

Cette thèse nous plonge dans le contexte de la complexité grandissante des systèmes informatiques. Un des axes thématiques de recherche émergeant dans ce contexte est l'**informatique autonomique** [5], qui vise à étudier les possibilités de gestion autonomique de tels systèmes. Cette thèse fournit un aperçu des problématiques et des défis soulevés par cette complexité grandissante. Elle met l'accent sur la diversité des systèmes et la nécessité de prendre en compte cette diversité.

1. Architecture, Systèmes, Temps-Réel, Embarqués
2. Institut de Recherche en Informatique de Toulouse
3. Modélisation des réseaux et des signaux
4. Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS à Toulouse
5. En anglais : autonomic computing. L'autonomique est un mot inspiré de la biologie.

1.1 Motivation principale : la gestion de la complexité grandissante

Dans cette section, nous soulevons la motivation principale de cette thèse. Il s'agit de faire face à l'accroissement de la complexité des systèmes informatiques, qui dans un futur proche (de l'ordre de quelques années) risque fort d'être le principal frein à l'évolution et au développement de ces systèmes. En premier lieu, nous faisons une brève distinction entre le compliqué, le complexe et le simple. Nous nous penchons ensuite sur des exemples historiques de technologies dont la complexité a été maitrisée, avant d'introduire la dualité entre la simplicité et la complexité.

1.1.1 Le compliqué, le complexe et le simple

"Les choses devraient être faites aussi simples que possible, mais pas plus simples." - Albert Einstein.

Levons en premier l'ambiguïté entre compliqué et complexe car on mélange parfois à tort ces deux adjectifs. Une chose compliquée est embrouillée, difficile à faire ou à comprendre parce qu'elle a été conçue comme telle, alors qu'une chose complexe comporte en soi plusieurs éléments imbriqués, ce qui peut la rendre difficile à saisir. Une chose est compliquée parce qu'on l'a rendue compliquée, alors qu'elle aurait pu être simple. On dit souvent qu'on complique une chose pour rien. A l'inverse, une chose est intrinsèquement complexe, c'est à dire qu'elle est complexe en soi en raison de sa nature même et non par la faute de quelqu'un.

Lorsqu'on s'apprête à étudier un système de grande taille, il faut procéder par étapes. Si le système est déjà existant, il faut d'abord faire la part entre le compliqué et la complexité du système. Améliorer ensuite le compliqué et le rendre plus simple. Puis définir la complexité du système résultant, c'est à dire définir tout ce qui englobe les différents éléments le composant et comment ils sont reliés entre eux. C'est à partir de cette définition que l'on peut dégager une simplicité d'utilisation. Autrement dit, la simplicité d'utilisation d'un système consiste à cacher la complexité de tout ce qui n'est pas nécessaire à l'utilisation de ce système. Dans cette thèse, nous nous intéressons aux systèmes informatiques actuels, qui, pour "compliquer" la chose, mélangent parfois le compliqué et la complexité.

1.1.2 Histoire de simplifications

Prenons quelques exemples historiques pour illustrer des étapes de simplification technologiques. Le premier concerne l'histoire de l'automobile. Ne serait-ce que le mot automobile contient déjà une étymologie qui rassemble "auto" (du grec autos qui veut dire soi-même) et "mobile" (du latin qui veut dire qui se meut), signifiant une simplification d'utilisation par automatisation du mouvement. Dans les années 1900, les automobiles devaient être manoeuvrées par des opérateurs-chauffeurs qualifiés pour contrôler la transmission manuelle à l'huile, ajuster les bougies et régler les différentes pressions. Un mécanicien compétent était requis car les pannes étaient très fréquentes. A partir des années 1930, les automobiles deviennent "utilisables" par des personnes non qualifiées et deviennent populaires, les infrastructures en réseaux de routes et en stations-service se

développent fortement. C'est parce que l'interface d'utilisation a été grandement simplifiée et parce que la complexité de gestion a été maitrisée, que l'automobile devient de plus en plus sûre à utiliser et que la technologie se développe.

En continuant dans les technologies du 19ème siècle, les premiers réseaux électriques incluaient leur propre générateur et s'étalaient sur quelques kilomètres. Par exemple, le premier réseau électrique de New York avait essentiellement pour but l'éclairage, découpé par quartiers, mais souffrait de nombreuses pannes. Ces réseaux sensibles et notamment ceux implantés dans de grandes entreprises, étaient à l'origine gérés par des "vice-président pour l'électricité", ces personnes étaient en général des séniors ayant une longue expérience dans le management. Une génération plus tard, l'accès au réseau électrique [6] est largement simplifié et automatisé et l'interface d'utilisation, la prise électrique, est omniprésente.

Enfin, dans un passé plus proche, l'histoire des communications téléphoniques dans les années 1920 suit le même schéma. La demande pour des opérateurs de plus en plus formés dans les centres de commutation explose en même temps que l'utilisation de la téléphonie fixe. Ces opérateurs humains sont ensuite remplacés par des commutateurs automatiques et l'interface d'utilisation est elle aussi simplifiée, avec une numérotation standardisée pour joindre un correspondant plus facilement.

1.1.3 La dualité simplicité - complexité

Le point commun de ces histoires de simplifications vient du fait qu'à l'origine, toutes les sortes de technologies ont un besoin en investissement humain important. Les nouvelles technologies ou les technologies émergentes sont typiquement destinées à des spécialistes expérimentés. Elles font appel à un besoin d'expertise pointu pour l'installation, le démarrage et la maintenance des nouveaux systèmes. En général, le travail humain est considéré comme requis par défaut au premier abord, notamment pour ses qualités de flexibilité et d'adaptabilité. Au démarrage de ces nouvelles technologies, ce travail humain est donc toujours supérieur aux éventuelles alternatives.

Au final, lorsque la technologie arrive à maturation, les humains sont complètement supplantés ou en grande partie remplacés par une technologie devenue simple (pour les humains) et standardisée. Cela permet notamment d'élargir l'adoption de cette technologie (électricité, automobiles), de réduire les coûts et d'augmenter les ventes (révolution industrielle, agriculture) ou de permettre une performance accrue dépassant les capacités humaines actuelles (conquête spatiale). La simplification de l'utilisation de la technologie est très souvent liée à une complexification globale du système. En effet, les différentes tâches humaines nécessaires au commencement d'une technologie sont remplacées par des petits systèmes de contrôle, se rajoutant au système global. Il reste à savoir si c'est une règle universelle : c'est à dire si la simplicité d'usage engendre inévitablement un système complexe sur lequel s'appuyer.

6. le réseau électrique est aussi appelé "power grid" en anglais, dans cette thèse, nous faisons une relation avec les grilles informatiques ou "grid computing" en anglais

1.2 Problématiques soulevées par la thèse

Dans cette section, nous introduisons l'objectif de la thèse qui se décompose en deux thèmes principaux :

1. comment **gérer** des systèmes informatiques devenant de plus en plus complexes ?
2. comment assurer et maintenir une **généricité** vis-à-vis des systèmes déjà existants ?

Après avoir évoqué les statistiques de la complexité grandissante dans le domaine informatique, nous verrons les quatre problématiques soulevées par la gestion de cette complexité : l'auto-configuration "self-configuration", l'auto-optimisation "self-optimization", l'auto-guérison "self-healing" et l'auto-protection "self-protection". Ces quatre problématiques sont regroupées dans un objectif global surnommé l'auto-gestion "self-management" ou encore le "self-*" [7]. Le deuxième objectif concerne la prise en compte des systèmes déjà existants, que nous développerons en montrant le besoin de généricité. La problématique principale traitée dans cette thèse est l'application des propriétés fondamentales des "self-*" pour les systèmes informatiques patrimoniaux (propriétaires) déjà existants.

1.2.1 Les défis de la complexité grandissante

A partir des années 2000, en parallèle de l'explosion des technologies de l'information et de la communication, on assiste à une forte demande en administrateurs humains dans les systèmes informatiques. On parle souvent de "responsable des technologies de l'information et de la communication", ce qui fait penser aux "vice-président pour l'électricité" évoqués précédemment. Encore aujourd'hui, cette demande en administrateurs humains ne semble pas faiblir. C'est en 2001 que le responsable des laboratoires de recherche d'IBM, Paul Horn, a déclaré : "le niveau de complexité atteint par l'informatique a engendré une crise qu'il n'est plus possible de surmonter qu'en réalisant un immense travail collectif de recherche" [1]. En effet, les systèmes informatiques modernes se complexifient par l'ajout récurrent de nouvelles briques technologiques et l'émergence de grands systèmes imbriquant de multiples éléments communicants. L'administration de ces systèmes est d'autant plus problématique que l'adoption de ces nouvelles technologies par le grand public est beaucoup plus rapide (de l'ordre de quelques années) que les technologies évoquées précédemment (automobile, électricité sur plusieurs dizaines d'années...).

Le nombre d'outils et de matériels communicants ne cesse de croître et l'informatique mobile augmente en étendant ses technologies de communication. Pour les particuliers, les ordinateurs portables, PDAs, smartphones ou téléphones portables, ainsi que l'évantail des technologies sans fils utilisées rend la gestion et le contrôle manuels des systèmes de plus en plus difficiles. Pour les industriels, les nouveaux serveurs et services connectés ne cessent de se diversifier. Ce volume et cette complexité sont actuellement gérés par des humains hautement qualifiés, augmentant ainsi le coût de gestion. De plus, le contrôle manuel de ces systèmes prend beaucoup de temps et engendre un grand risque en terme d'erreurs humaines. Le problème général de ces systèmes informatiques modernes est que leur développement est freiné à cause de leur complexité et en particulier à cause de la complexité de leur gestion.

7. En anglais, le "self-*" se prononce "self-star"

D'après une étude sur les tendances technologiques menée à l'université de Berkeley en 2002 [2], les industries du domaine de l'information et de la technologie sont appelées à délivrer des services de plus en plus rapidement et à un coût moindre. Ces services doivent être intégrés dans des infrastructures déjà existantes, augmentant ainsi la complexité résultante. Le coût de gestion humain de ces infrastructures dépasse maintenant le coût des équipements matériels et logiciels. Ce coût multiple par un facteur allant de 3 à 18 le coût global d'investissement dans l'infrastructure informatique. De plus, le budget dépensé pour prévenir ou réparer les pannes varie dans une proportion allant d'1/3 à 1/2 du budget total du domaine, dépendant du type de système utilisé.

1.2.2 Les objectifs de la thèse : l'auto-gestion - "Self-management"

En anglais, le préfixe **"self"** s'applique aux actions s'effectuant elles-mêmes. On parle de système informatique **"self-*"** ou encore de **"selfware"** [8] à l'image des "software ou "hardware". C'est en 1995 que Wooldridge et Jennings [3] identifient les propriétés fondamentales des "self-*" en introduisant les premiers systèmes multi-agents :

- **autonomie** : les agents opèrent sans intervention directe humaine et ont un contrôle interne de leurs actions pour mener à bien une tâche définie.
- **abilité sociale** : les agents interagissent entre eux (et avec les humains) via des langages et des protocoles de communication inter-agents.
- **réactivité** : les agents perçoivent l'environnement qui les entoure et répondent aux changements de cet environnement.
- **proactivité** : les agents ne répondent pas seulement aux changements de l'environnement mais peuvent aussi prendre des initiatives suivant des politiques ou des stratégies.

Cette approche est une des solutions à la gestion de la complexité.

1.2.3 L'auto-gestion - "Self-management"

C'est en appliquant les propriétés des "self-*" pour les systèmes informatiques que Kephart et Chess [4] ainsi que Brantz [5] définissent en 2003 les quatre paradigmes devant être implémentés au minimum dans de tels systèmes pour qu'ils deviennent auto-gérés ou "self-managed" : **"self-configuration"**, **"self-optimization"**, **"self-healing"** et **"self-protection"**. Dans cette thèse, nous suivons une démarche qui apporte des contributions dans ces quatre problématiques, chaque aspect étant nécessaire pour la création d'un système informatique qui fait face à la gestion de la complexité grandissante :

1. **Auto-guérison - self-healing** : Un système est qualifié de "self-healing" [9] quand il peut maintenir son activité en surmontant les événements routiniers ou extraordinaires dus aux dysfonctionnements de ses éléments. Il peut se surveiller, détecter les erreurs ou les situations qui peuvent se manifester plus tardivement comme des erreurs. Il peut initier une réparation par lui-même et ce de manière à ce que son activité soit la moins perturbée possible. On parle aussi de la **gestion des pannes** : les pannes sont très fréquentes dans un environnement composé de centaines voire

8. On pourrait traduire **selfware** en français par auto-logiciel ou de manière plus générale auto-informatique.

9. En français cela peut se traduire par s'auto-guérir ou bien s'auto-réparer

de milliers de machines et d'entités logicielles. Il existe deux types de panne : la panne matérielle et la panne logicielle. La gestion des pannes consiste à détecter la défaillance d'une machine ou d'un logiciel en vue de permettre aux administrateurs de ramener le système dans un état opérationnel.

2. **Auto-protection - self-protection** : la "self-protection" d'un système lui permet d'ajuster son comportement pour assurer sa survie, ainsi que pour la confidentialité et la protection de ses données. Cet "instinct de conservation" doit aussi lui permettre de réagir à des attaques de toutes sortes, en anticiper les menaces et lui permettre de prendre les mesures adéquates. On parle aussi de la **gestion de la sécurité** : cette gestion peut être décomposée en deux vues : la protection interne et la protection externe. La protection interne apporte une cohérence globale au système à tout instant (c'est à dire maintient le système dans un état cohérent) et la protection externe contre les événements dit exogènes gère les attaques malicieuses, les infiltrations et autres infections par des virus.
3. **Auto-configuration - "self-configuration"** : La propriété de "self-configuration" permet à un système de s'installer, de se paramétrer et de se mettre à jour tout seul. La manière d'installer et de paramétrer les éléments dépend des besoins du système mais aussi des utilisateurs. Si certains éléments sont configurés ou installés manuellement par un humain, le système en comprend les implications et s'y adapte. On parle aussi de la **gestion du déploiement** : cette fonction est subdivisée en plusieurs tâches à savoir : la configuration, l'installation, le démarrage, la désinstallation, la mise à jour et l'arrêt des éléments d'un système ;
4. **Auto-optimisation - "self-optimisation"** : La propriété de "self-optimization" permet au système d'utiliser au mieux et en toutes circonstances toutes ses ressources. Cette optimisation doit s'effectuer dans le respect des critères définis dans la politique de gestion. Par exemple, une politique de gestion avec un compromis entre énergie électrique consommée et performance. On parle aussi de la **gestion de performance** : cette fonction consiste à assurer que le système reste dans un état dans lequel sa performance est la plus proche de l'optimal. L'environnement à administrer doit être adapté aux différentes variations de performance et aux pics de charge. Cela peut être, par exemple dans le cas d'un serveur, l'augmentation du nombre de requêtes clients à traiter.

Nous développerons à travers quatre chapitres distincts les contributions sur ces quatre propriétés fondamentales du "self-management".

1.2.4 Le besoin de généricité

Dans le domaine informatique, le concept de généricité est utilisé pour les langages de haut niveau (par exemple pour les langages dit *objets*) et pour la conception de systèmes complexes. Pour les langages informatiques, la programmation générique consiste à séparer le programme en un ensemble de descriptions (de fonctionnement, de cohérence) de haut niveau d'une part et de leur implémentation d'autre part. Cela permet notamment de construire des algorithmes au dessus de ces descriptions, en toute indépendance par rapport à l'implémentation de bas niveau. Pour la conception de système complexes, la généricité permet d'abstraire un ensemble de concepts, d'aspects ou d'éléments cohérents

pour décrire une base de travail. Ce socle de travail permet de construire un nouveau système par dessus, indépendamment du fonctionnement sous-jacent. Dans les deux cas, la généricité est issue d'une description des interfaces entre les différents niveaux d'abstraction.

Ce concept permet de séparer un système complexe en différentes couches avec une description générique des relations inter-couches. Les avantages sont que les différentes briques ou couches du système deviennent simples à comprendre. Une fois qu'une confiance s'installe pour de nouvelles briques (c'est à dire que la fiabilité des nouvelles briques est reconnue), elles sont utilisées pour construire de nouveaux systèmes. Du point de vue de l'utilisateur du système, le concept de généricité peut se décliner en différentes couches d'utilisation, c'est à dire que l'utilisateur a le choix de la couche sur laquelle il souhaite contrôler le système, chaque couche laissant son degré de liberté d'utilisation et de complexité.

Dans notre approche, nous utilisons le concept de généricité pour cacher la complexité de bas niveau et ainsi permettre de décrire des politiques de gestion avec un haut niveau d'abstraction.

1.3 Vue globale de la thèse

Ce manuscrit de thèse est structuré en sept chapitres : les deux premiers chapitres sont l'**introduction** et l'**état de l'art**. Ils permettent de définir les concepts utilisés tout le long de la thèse, notamment celui de l'administration autonomique et d'avoir un aperçu sur les approches proposées dans la littérature. Les chapitres **Auto-guérison "self-healing"**, **Auto-configuration - "self-configuring"**, **Auto-optimisation - "self-optimizing"** et **Auto-protection - "self-protecting"** mettent en valeur les contributions dans les quatre facettes de l'administration autonomique et positionnent notre approche par rapport aux approches existantes. Enfin, les chapitres conclusions, perspectives, puis les annexes permettent d'avoir des éléments pour les futurs travaux de recherche, ainsi que certains détails d'intégration et d'implémentation techniques, pouvant aider pour la reprise des concepts développés.

Chapitre 2

État de l'art

Sommaire

2.1 L'informatique autonomique ou "Autonomic Computing" . 10
2.1.1 Automatique, Autonome et Autonomique 10
2.1.2 La vision de l'informatique autonomique : historique et inspiration biologique 11
2.1.3 La gestion autonomique 12
2.2 La boucle de contrôle autonomique 13
2.2.1 La boucle MAPE-K 13
2.3 Contexte de l'étude et définitions 16
2.3.1 Grappes de machines ou "Clusters" 16
2.3.2 Grilles informatiques 16
2.3.3 Centres de calcul et de traitement des données "Data centers" et nuages informatiques "clouds" 17
2.3.4 Réseaux de capteurs 19
2.3.5 L'intergiciel XSStreaMWare de gestion des réseaux de capteur 20
2.3.6 Les logiciels patrimoniaux existants 22
2.3.7 L'application DIET 22
2.3.8 Les applications de simulation électromagnétiques 24
2.4 Positionnement par rapport à l'existant 25
2.4.1 Degré d'autonomicité et degré de généricité 25
2.4.2 Exemples d'approches existantes 27
2.5 Présentation du système pour l'intégration des contributions 38
2.5.1 Le système de gestion autonomique TUNe : Toulouse University Network 38
2.5.2 Les phases d'administration autonomique dans TUNe 39
2.5.3 Vocabulaire du triangle TUNe et fonctionnement global 40
2.5.4 Etapes de construction d'une gestion autonomique avec TUNe 41
2.6 Etat de l'art : synthèse 48

Dans ce chapitre, nous nous positionnons par rapport à l'état de l'art et nous introduisons l'outil sur lequel nos contributions sont appliquées. Une première partie définit les concepts de l'informatique autonomique et de la boucle de contrôle autonomique. La

partie suivante présente le contexte de notre étude, des définitions et des exemples de plateformes que nous utilisons pour nos validations. Nous nous positionnons ensuite par rapport à l'existant en faisant un état de l'art original qui introduit un positionnement par rapport aux degrés d'autonomicité et de généricité des approches. Enfin, nous terminons par la présentation du système pour l'intégration des contributions et faisons une synthèse du chapitre.

2.1 L'informatique autonomique ou "Autonomic Computing"

Dans cette section, nous définissons le mot autonomique en le comparant aux mots automatique et autonome. Nous introduisons ensuite le concept d'informatique autonomique issu d'une inspiration des systèmes biologiques tels que le système nerveux qui est qualifié d'autonomique.

2.1.1 Automatique, Autonome et Autonomique

2.1.1.1 Différence entre automatique et autonome

Les termes **automatique** et **autonome** se réfèrent à un même but : celui de soulager ou de restreindre au strict minimum une intervention humaine. Dans le domaine de l'informatique, ces termes concernent des processus qui peuvent être exécutés indépendamment, du début à la fin, en évitant au maximum l'intervention humaine. La différence réside dans le fait que les processus automatiques remplacent simplement une routine manuelle contre une routine codée (de manière logicielle ou matérielle) qui suit une séquence décrite étape par étape. Cette séquence peut éventuellement faire intervenir de nouveau l'humain à une certaine étape. Un processus autonome quant à lui, émule un comportement humain dans le sens où il englobe un ensemble de tâches. Un système autonome peut être vu comme une imbrication de systèmes automatiques en y ajoutant la communication entre ces systèmes automatiques.

2.1.1.2 Différence entre autonome et autonomique

L'**autonomie** est le fait de déléguer de la responsabilité au système autonome pour atteindre des objectifs définis. C'est à dire que l'autonomie est l'automatisation de la prise de responsabilité et de la prise de décision pour que les différentes tâches du système soient réussies. L'**autonomique**, quant à lui, est le fait que le système se contrôle lui-même, sa spontanéité résulte en général de stimulus. C'est à dire que l'autonomique est l'automatisation de la prise de responsabilité et de la prise de décision pour que tout le processus d'opération du système fonctionne. Ainsi, en fonction du degré d'automatisation, on peut imaginer qu'un système autonomique puisse créer ses propres buts.

Par exemple, un système qui a pour tâche de détecter un phénomène particulier peut utiliser un ensemble d'éléments prédéfinis. Ce système peut être autonome et décider par lui-même de quels éléments il a besoin pour détecter ce phénomène, c'est à dire qu'il choisit ses paramètres internes afin d'accomplir ses tâches de détection. Par contre, tout ce qui assure que le système continue d'opérer au mieux, même quand plusieurs

éléments tombent en panne, ne fait pas directement partie des buts premiers du système, c'est à dire de son autonomie de détection. Un système autonome peut donc échouer dans son but premier s'il ne gère pas les changements dynamiques, les comportements incertains dans son propre environnement. Dans cette optique, l'autonomique peut se définir comme une spécialisation de l'autonomie, c'est à dire une autonomie vis-à-vis de la gestion du système lui-même. Le système autonomique peut en effet être considéré comme un système autonome qui inclue nécessairement ses propres systèmes de contrôle, c'est à dire qu'il s'auto-contrôle.

2.1.2 La vision de l'informatique autonomique : historique et inspiration biologique

A la fin des années 90, la DARPA [1] (agence initiatrice du développement d'Internet) développe une des premières implémentations du concept de "self-management" au sein de son projet militaire SAS [2]. Le but était de créer un réseau de petits équipements que portent les soldats sur le terrain pour transmettre des informations sous forme de texte ou de petits messages vocaux. La particularité de ces équipements est leur capacité de pouvoir se configurer et de se reconfigurer automatiquement en cours de fonctionnement. Les paramètres modifiables sont par exemple le débit de transmission, la fréquence de la porteuse du signal émis, qui sont adaptés en fonction de l'environnement (bas débit et basse fréquence pour les grandes distances, haute fréquence et haut débit pour les courtes distances). Chaque équipement communique avec ses voisins qui retransmet à son tour l'information aux voisins suivants. Les équipements sont aussi capables de se reconfigurer en cas de brouillage du signal par des ennemis, ainsi que pour minimiser l'interception d'information. Cette **reconfiguration dynamique** en fonction de l'environnement et du maillage formé par les voisins relève de plusieurs problématiques [6] comme le routage adaptatif multi-sauts (pour n'envoyer les données qu'aux voisins les plus proches), le facteur d'échelle (pour gérer jusqu'à 10000 équipements sur le terrain), les contraintes temporelles (pas plus de 200ms pour l'adaptation dynamique et le début de la transmission du message avec le risque de perdre le début du message).

Au début des années 2000, la DARPA se penche ensuite sur une approche architecturale nommée DASADA [3][7] qui introduit systématiquement des **sondes spécialisées** renvoyant des informations sur l'environnement. Ces informations sont ensuite utilisées pour la reconfiguration dynamique et automatique en cours d'exécution.

C'est en 2003 que Kephart expose une vision pour l'**informatique autonomique** dans son article fondateur "A vision for autonomic computing" [4]. Il s'inspire d'une métaphore biologique du système nerveux humain qui permet à chacun de faire des efforts cognitifs, physiques ou de prendre des décisions sans avoir à se soucier du calcul du rythme cardiaque, de la pression sanguine ou de la quantité d'adrénaline à produire. En transposant délibérément le mot "autonomique" utilisé dans le domaine de la biologie vers le domaine informatique, il affirme que ces futurs systèmes informatiques autonomiques

1. Defense Advanced Research Projects Agency, soit en français : agence pour les projets de recherche avancée de défense

2. Situation Awareness System

3. Dynamic Assembly for System Adaptability, Dependability, and Assurance ; soit en français : Assemblage dynamique pour l'adaptabilité, la vérification d'intégrité et la confiance dans les systèmes

agiraient et réagiraient de même, permettant aux utilisateurs et aux administrateurs de se focaliser sur des problématiques de plus haut niveau. Kephart s'inspire aussi de la conviction exprimée il y a presque un siècle par le grand mathématicien Alfred North Whitehead qui déclarait "La civilisation progresse par l'automatisation croissante des tâches et fonctions ce qui permet de libérer l'homme qui peut ainsi consacrer de plus en plus de temps à la réflexion théorique et conceptuelle".

2.1.3 La gestion autonomique

Le but de l'informatique autonomique est de **surpasser la complexité des systèmes** logiciels et matériels utilisés dans l'informatique d'aujourd'hui et de demain en leur procurant des capacités d'auto-gestion [4]. Nous pouvons soulever à nouveau le paradoxe de la complexité des systèmes (voir section 1.1.3) : pour arriver à intégrer des capacités d'auto-gestion, le système global a besoin de devenir encore plus complexe. Ce point n'est pas négligeable car très souvent, une complexification du système est encore synonyme de plus de problèmes et plus de maintenance. Cette complexification peut donc rebuter au premier abord. C'est au domaine de recherche en autonomic computing de convaincre de la nécessité d'une complexification et de démontrer l'efficacité finalement apportée au système global.

Paul Horn définit les "7 commandements" d'un système informatique autonomique dans son manifesto de l'autonomic computing [1] :

1. Apprendre à **se connaître lui-même** - Une parfaite connaissance de soi et de son environnement implique, pour un système informatique autonomique, que chacun de ses éléments indique son état et ses performances à un "organe" de supervision. Aussi, le système autonomique a besoin de connaître ses capacités ultimes ou d'être en mesure de les identifier ou de les calculer. Il doit aussi connaitre les systèmes auxquels il est relié et éventuellement pouvoir fusionner avec eux ou se séparer en plusieurs petits sous-systèmes en cas de besoin.
2. **Détecter les pannes** et les éviter - Si une panne survient dans le système, il faut qu'elle soit détectée. L'élément perturbateur doit alors être isolé et ses tâches en cours déportées automatiquement vers un autre système. L'idéal restera que le système puisse prévoir la panne et apporter des actions en prévention.
3. **Réparer les pannes** et les erreurs commises - Puisque les pannes et les erreurs humaines ne pourront jamais être totalement évitées, des mécanismes de réparation sont essentiels et doivent être intégrés au système. Un mécanisme d'annulation ou de retour dans un état de fonctionnement doit exister pour contrer n'importe quel fonctionnement anormal.
4. Garder la **maîtrise des données** - Si un ordinateur tombe en panne, ses disques durs ne seront plus accessibles. Les données doivent donc être conservées indépendamment des machines qui les traitent et être partagées entre tous les éléments le constituant.
5. **S'optimiser** en s'auto-éduquant - L'ordinateur doit apprendre à optimiser de lui-même le fonctionnement du système d'exploitation ou d'une base de données, à

4. en anglais : "self-managing"

rechercher une meilleure voie de communication et, au mieux, à tirer parti de ses propres erreurs.

6. **Comprendre les intentions** des humains - Autonomes, autonomiques ou pas, les machines travaillent quand même pour l'homme. Elles doivent comprendre nos intentions, dénicher l'information où qu'elle se trouve et la présenter différemment selon l'appareil utilisé.

7. L'informatique autonomique ne peut pas se concevoir et exister dans un environnement clos. Elle doit pouvoir évoluer dans un **environnement ouvert, hétérogène** et caractérisé par la multiplicité des normes et des protocoles informatiques. Il ne peut donc s'agir d'une solution propriétaire.

2.2 La boucle de contrôle autonomique

Dans cette section, nous développons la boucle de contrôle autonomique MAPE-K introduite par IBM au début des années 2000 et nous détaillons les quatre tâches principales que sont l'**Observation**, l'**Analyse**, la **Planification** et l'**Exécution**.

2.2.1 La boucle MAPE-K

Pour atteindre les objectifs d'une informatique autonomique, l'idée principale est de s'inspirer des boucles de contrôles utilisées en automatique : un contrôleur guidé par un modèle ajuste en continu des paramètres sur les objets contrôlés en fonction des mesures de retour. IBM a suggéré un modèle de référence pour la boule de contrôle autonomique en informatique [4], qui est souvent appelée la boucle MAPE-K (**Monitor**, **Analyse**, **Plan**, **Execute**, **Knowledge**). Ce modèle est de plus en plus utilisé pour détailler les aspects architecturaux des systèmes autonomiques. Il est illustré par la figure 2.1 et est similaire au modèle générique d'agents [8], dans lequel un agent intelligent perçoit son environnement à travers des capteurs et utilise sa perception pour déterminer des actions à exécuter sur l'environnement.

Dans la boucle MAPE-K, les **éléments gérés** représentent des ressources logicielles ou matérielles auquel un comportement autonomique est fourni en les couplant avec un **gestionnaire autonomique**. Les **capteurs**, aussi appelés senseurs ou sondes [5], collectent des informations, aussi appelées métriques, sur les éléments gérés. Les actionneurs prennent en charge les changements à effectuer sur les éléments gérés. Ces changements peuvent être vus à deux granularités différentes : le niveau gros grain par le changement de plusieurs éléments en même temps ou le niveau petit grain par le changement de paramètres internes à un élément. La distinction architecturale de ce gestionnaire autonomique contribue à une complexification du système. Au terme de la maturité de l'administration autonomique, cette distinction s'effacera pour devenir presque conceptuelle plus qu'architecturale.

5. en anglais : probes, sensors ou gauges

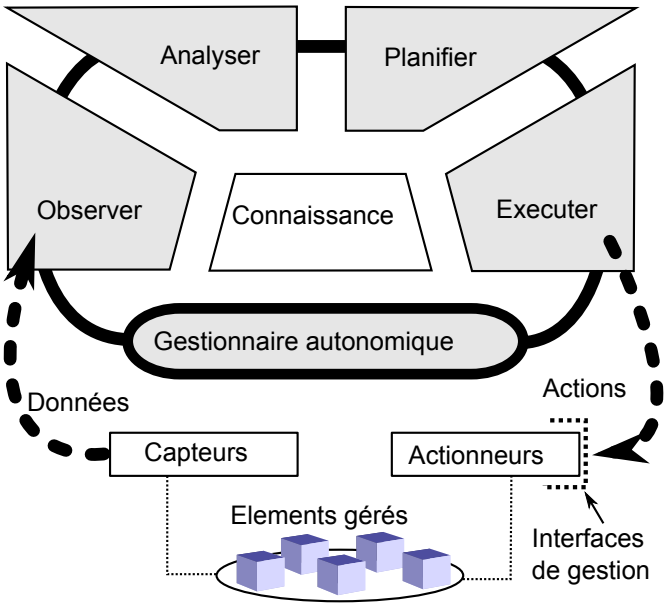


FIGURE 2.1 – Boucle de contrôle autonomique MAPE-K

2.2.1.1 Le gestionnaire autonomique

Les informations collectées par les capteurs permettent au gestionnaire autonomique d'observer les éléments gérés et d'exécuter les changements au travers des actionneurs. Ce gestionnaire autonomique est un composant logiciel qui est idéalement configuré par des administrateurs humains en utilisant une description de haut niveau de l'expression de leur stratégie (les politiques). Le gestionnaire utilise ensuite les informations des capteurs et sa connaissance interne pour planifier et exécuter, en fonction des politiques décrites, toutes les actions de bas niveau nécessaires pour atteindre l'objectif de la stratégie.

2.2.1.2 L'observation ou "monitoring"

La tâche d'observation de la boucle MAPE-K apporte des mécanismes de capture de certaines propriétés de l'environnement. Pour cette tâche on utilise les verbes sonder ou par abus de langage "monitorer" (issu de l'anglais "monitoring"). Ces propriétés sont à priori toutes importantes pour la gestion autonomique du système. Le gestionnaire autonomique a besoin d'informations appropriées pour reconnaître le fonctionnement anormal ou la sous-performance d'éléments du système et a besoin de savoir interpréter ces propriétés, d'où une éventuelle mise en forme des informations remontées par les sondes (ou capteurs). En général on parle d'observation active et passive ou "monitoring" passif et actif. Le **"monitoring passif"** n'a pas besoin de modifier le fonctionnement interne du système observé (par exemple, le rajout d'un code spécifique supplémentaire dans un logiciel). Il observe les interactions du système et ses états en étant le moins intrusif possible, c'est à dire en évitant au maximum de perturber le fonctionnement normal du système. L'observation ou **"monitoring actif"** fait appel à un certain niveau à des techniques d'ingénierie qui modifient ou rajoutent du code ou des instruments à l'implémentation des éléments du système. Par exemple, la capture des appels de fonctions pour une appli-

cation ou la capture des appels de bas niveau pour le système d'exploitation. Le surplus de code pour le "monitoring" actif peut souvent être rajouté automatiquement par des outils spécifiques. Par exemple, le logiciel ProbeMeister [9] peut insérer des sondes dans le bytecode compilé de Java.

2.2.1.3 L'analyse

La tâche d'analyse de la boucle de contrôle autonomique MAPE-K prend en compte les données d'observation fournies par les sondes pour les comparer suivant les politiques et les stratégies globales de gestion. Cette comparaison permet de déterminer une différence entre l'état d'exécution et d'opération effectif du système et celui attendu. Cette tâche fournit donc des indicateurs quantitatifs et qualitatifs de l'opération du système.

2.2.1.4 La planification

La tâche de planification de la boucle de contrôle autonomique produit une série de changements à effectuer sur les éléments gérés. Ces changements dépendent de la comparaison effectuée par la tâche précédente d'analyse. Ils sont fonction des différences de fonctionnement réel et attendu. Cette tâche planifie aussi de manière temporelle les changements à apporter au système. En effet, il peut y avoir des contraintes temporelles retardant les changements ou des priorités à prendre en compte. Parmi les contraintes temporelles notons celle induite par le temps d'exécution total de la boucle de contrôle autonomique.

2.2.1.5 L'exécution

La tâche d'exécution se charge de respecter la planification des changements et d'exécuter en temps voulu les actions nécessaires sur les éléments gérés. Cette tâche fait le lien entre les calculs de la boucle de contrôle et le monde réel. Elle peut être en charge de distribuer des actions sur plusieurs machines si un changement fait intervenir plusieurs éléments gérés.

2.2.1.6 La base de connaissances

Reliant les différentes tâches de la boucle MAPE-K, la base de connaissances K[6] peut provenir de différentes sources collectant l'information de manière automatique ou même manuelle. Il peut s'agir d'historiques de fonctionnement, d'observations du comportement du système, de données accumulées et stockées (par exemple les logs des sondes) ou même d'informations rajoutées par des experts humains pouvant aider le gestionnaire autonomique. Les quatre tâches peuvent venir chercher des informations ou rajouter des informations dans cette base. Il y a nécessité de définir une méthode de représentation et de manipulation de ces données présentes dans la base.

6. le K vient du mot Knowledge en anglais

2.3 Contexte de l'étude et définitions

Dans cette section, nous définissons les applications et plateformes matérielles nous ayant servi d'études de cas pour notre recherche.

2.3.1 Grappes de machines ou "Clusters"

Un "cluster" [7] est un ensemble d'ordinateurs appelés noeuds [8] ou machines reliés par un réseau local classique de type ethernet (1 Gbit/s) ou à très haut débit comme Myrinet, Infiniband ou Quadrics [10]. Dans la très grande majorité des cas, un cluster est composé d'un ensemble homogène de machines (même type de matériel et logiciel : processeur, mémoire, système d'exploitation). Les programmes s'exécutant sur des clusters utilisent généralement des API [9] telles que MPI [10] [11] pour la communication entre les processus distribués sur les machines. L'avantage de regrouper plusieurs ordinateurs indépendants est le coût de l'installation finale comparé aux superordinateurs multiprocesseurs. Un autre avantage est l'utilisation de petits systèmes que l'on peut connecter à la suite les uns des autres selon l'évolution des besoins. Les inconvénients sont la vitesse limitée du réseau inter-machine comparé aux bus matériels des cartes-mères des superordinateurs et surtout la gestion de la complexité grandissante à cause de la parallélisation de tous les systèmes (installation, mises à jour, pannes).

2.3.2 Grilles informatiques

Une grille informatique est un ensemble de ressources informatiques (en général des clusters), distribuées, hétérogènes et partagées entre des entités géographiquement dispersées et autonomes (c'est à dire ayant leur propre politique de gestion). Le mot grille provient de l'anglais "grid", lui-même issu de la métaphore d'aggrégation des réseaux autonomes comme à l'époque du développement de l'électricité appelé alors "power grid". En effet, il est fait une distinction entre un réseau où les entités collaborent et une grille où les entités partagent leurs infrastructures. De ce point de vue, une grille informatique intègre obligatoirement un intergiciel [11], qui permet le dialogue entre les différents éléments d'une application répartie sur une grille. Cet intergiciel masque la complexité des échanges inter-éléments. La grille informatique est donc une aspiration à utiliser des protocoles standards afin de pouvoir partager les ressources par un groupement d'entités autonomes consententes. En faisant le rapprochement avec la "power grid", les entités produisant de l'électricité et consommant de l'électricité sont reliées dans un même but : partager de la puissance électrique par une interface universelle : la prise de courant. La complexité d'acheminement, de transformation du courant ou de partenariat/contrats entre les entités productrices d'électricité est alors cachée à l'utilisateur final. Dans une grille informatique, les entités qui partagent leurs ressources informatiques sont qualifiées d'Organisations Virtuelles (VOs [12]). Elles ont chacune leur propre politique de gestion

7. en français : une grappe de serveur
8. en anglais : nodes
9. Application Programming Interface
10. Message Passing Interface
11. en anglais : middleware, pourrait se traduire littéralement par "élément du milieu"
12. en anglais VO : Virtual Organizations

locale et un contrat d'appartenance à la grille. Ces contrats définissent la politique de gestion de la grille au niveau global, les partages définis entre VOs et l'utilisation éventuellement différenciée de la grille par les VOs. Cette gestion et ces contrats sont plus ou moins automatisés en fonction de l'intelligence introduite au niveau de l'intergiciel de la grille.

Exemple : la grille de calcul Grid'5000

Le projet national Grid'5000 a permis de construire une plateforme expérimentale de recherche constituée d'une grille informatique de grande taille (avec l'objectif d'atteindre le nombre symbolique de 5000 ordinateurs à l'échelle de la France) [12]. Dans cette grille, les VOs sont des laboratoires de recherche et des universités françaises qui partagent des clusters d'environ 500 ordinateurs chacuns, situés dans les villes (aussi appelés sites) de Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay (Paris), Rennes, Sophia-Antipolis et Toulouse. Les sites sont interconnectés par de la fibre optique dédiée (fibre dite "noire") à 10Gbps, ce backbone optique étant géré par Renater [13]. L'intergiciel utilisé pour réserver tout ou partie de cette grille est OAR [14], développé à l'IMAG de Grenoble. OAR est un système de commandes interactives [13] écrit en Perl qui utilise une base de données MySQL pour la gestion des ressources partagées dans la grille. Il est composé de modules qui sont des programmes indépendants communiquants par la base de données :

- **oarsub** permet de réserver des ressources localement à un site, en spécifiant éventuellement des contraintes sur des propriétés (mémoire, processeur, disque, système).
- **oargrid** permet de réserver des ressources de la même manière mais à l'échelle de la grille entière.
- **monika** permet de visualiser les réservations actuelles et futures sous forme de tableaux.
- **drawOARGantt** permet de visualiser les réservations sous forme de diagrammes de GANTT.

Chaque site possède un noeud d'accès connecté à internet appelé la "frontale" qui possède les modules d'OAR. Pour des raisons de sécurité (utilisation de toutes les ressources simultanément pour des attaques), Grid'5000 fonctionne en environnement clos : les noeuds doivent passer par un proxy (une machine par site) pour accéder à internet. La grille Grid'5000 étant dédiée à la recherche, sa spécificité est la liberté de manipulation des noeuds, avec la possibilité d'installer son propre système d'exploitation personnalisé à l'aide de l'outil Kadeploy [15]. Dans cette thèse, nous utilisons Grid'5000 pour des expériences de calculs distribués et pour le déploiement de simulations électromagnétiques distribuées.

2.3.3 Centres de calcul et de traitement des données "Data centers" et nuages informatiques "clouds"

Un centre de calcul et de traitement des données [14] regroupe un ensemble de "clusters" dans un même lieu géographique, on parle aussi de ferme de machines ou ferme de

13. en anglais : batch system
14. en anglais : data center, qui peut aussi s'écrire datacenter ou datacentre

serveurs [15]. Un tel centre contient des aménagements physiques (armoires de machines, double sols et plafonds, climatisations, alimentations, transformateurs, batteries et générateurs de secours) afin d'accueillir des services sur des serveurs spécialisés (serveurs de base de données, de calcul, de fichiers, d'applications). L'administration d'un tel centre fait intervenir la gestion d'une connectivité très dense, avec de multiples routeurs et liens réseaux très souvent redondants. Des équipements de sécurité réseau y sont aussi présents : pare-feu, VPN [16], systèmes de détection d'intrusion.

Les nuages informatiques ou "clouds" sont une couche supplémentaire rajoutée aux data centers pour faciliter l'accès à ses ressources. Un fournisseur de cloud peut déployer cette couche sur de nombreux data centers répartis à l'échelle mondiale. Ce nouveau service d'accès distant fait partie de la mouvance de délocalisation vers le nuage informatique [17] [16]. Cette couche est composée d'une part de machines virtuelles (les VMs [18], logiciels émulants un ordinateur) gérées par un hyperviseur (gestionnaire de VMs pouvant tourner sur une ou plusieurs machines) et d'autre part de la définition de l'interface d'accès fournie à l'utilisateur. On distingue trois types de services d'accès offerts par les clouds : l'accès direct aux VMs appelé PaaS [19], l'accès au machines brutes (sans système d'exploitation) IaaS [20] et l'accès aux logiciels applicatifs appelé SaaS [21] [17]. Les services PaaS/IaaS et SaaS offrent de multiples interfaces : humaines par le web ou par l'intégration dans des interfaces graphiques et automatiques par des API compatibles avec de nombreux langages. Ceci amène des problématiques de gestion dynamique et de réactivité pour suivre les fluctuations des demandes des multiples utilisateurs du cloud.

Exemple : la plateforme Grid'MIP

Grid'MIP est une plateforme ayant un coeur de réseau comparable à celui d'un datacentre. Cette plateforme de recherche est basée au CICT [22] et sa spécificité est de pouvoir réserver l'infrastructure dans son entier pour des expériences. L'infrastructure est composée de 6 clusters reliés à des routeurs Cisco Catalyst 7600. Six routeurs de ce type sont reliés par une topologie maillée avec des liens à 10Gb/s, une topologie comparable à celle présente par exemple dans un datacenter. L'ensemble des matériels (machines et équipements d'interconnection) sont accessibles à distance et configurables au plus bas niveau (par exemple, chargement de paramètres des routeurs, du système d'exploitation des routeurs ou des machines). Dans cette thèse, nous utilisons Grid'MIP pour des expériences d'optimisation de la consommation d'énergie des équipements réseaux, à l'échelle d'un datacenter (Annexe D).

15. En anglais : compute farms, server famrs
16. Virtual Private Network, avec des serveurs spécialisés pour l'encryption des communications
17. En anglais : "The compute cloud"
18. Virtual Machines
19. Platform As A Service, le service d'accès à la plateforme
20. Infrastructure As A Service, le service d'accès à l'infrastructure
21. Software As A Service, le service d'accès aux logiciels
22. Centre Interuniversitaire de Calcul de Toulouse

2.3.4 Réseaux de capteurs

Un réseau de capteurs regroupe un grand nombre de noeuds qui, au lieu d'être des machines comme dans les cas précédents, sont des micro-capteurs capables de récolter et de transmettre des mesures physiques [18]. Ces capteurs sont en général dispersés dans une zone géographique correspondant au terrain d'intérêt pour le phénomène mesuré. Les noeuds intègrent quatre unités de base :

- l'unité de **captage** : capte le phénomène physique et le transforme en signal numérique.
- l'unité de **traitement** : processeur traitant le signal numérique et petite unité de stockage.
- l'unité de **transmission** : effectue les émissions et réceptions des données filaires ou sans fil.
- l'unité de **contrôle d'énergie** : batteries, gestionnaire de répartition d'énergie et systèmes de recharge.

Selon le domaine d'application, il peut aussi contenir des modules supplémentaires tels qu'un système de localisation géographique (GPS) ou bien un système générateur d'énergie (cellule solaire). Quelques micro-capteurs, plus volumineux, sont dotés d'un système mobilisateur chargé de les déplacer en cas de nécessité. Ces noeuds sont donc de véritables systèmes embarqués (matériel avec système d'exploitation et logiciels spécifiques) et le déploiement de plusieurs d'entre eux en vue de collecter et transmettre des données vers un ou plusieurs points de collecte soulève des problématiques de complexité grandissante et donc de gestion autonomique.

2.3.4.1 Exemple : les SunSPOTs

Les SunSPOTs [23] [19] sont de petites plateformes (tenant dans la main) sans fils qui sont capables d'exécuter des programmes écrits dans le langage java. Elles embarquent une batterie, un ensemble de capteurs (accéléromètre 3 axes, lumière, température), 8 LEDs [24] tricolores, 6 ports d'entrée analogiques et 9 ports d'entrée/sorties génériques pour la connexion à des capteurs externes ou d'autres équipements (hauts-parleurs, moteurs...). Ces capteurs ont la particularité de pouvoir se configurer, se contrôler et recevoir des applications java à distance (par le réseau sans fil). Lors d'un déploiement sur le terrain d'un grand nombre de capteurs, leur gestion manuelle devient sensible. Dans cette thèse, nous utilisons les SunSPOTs dans le cadre d'un échange avec l'université de Tokyo et dans le laboratoire NII [25] pour des expériences de configuration et re-configuration sans intervention humaine.

23. Sun Small Programmable Object Technology
24. Diodes electroluminescentes
25. National Institute of Informatics

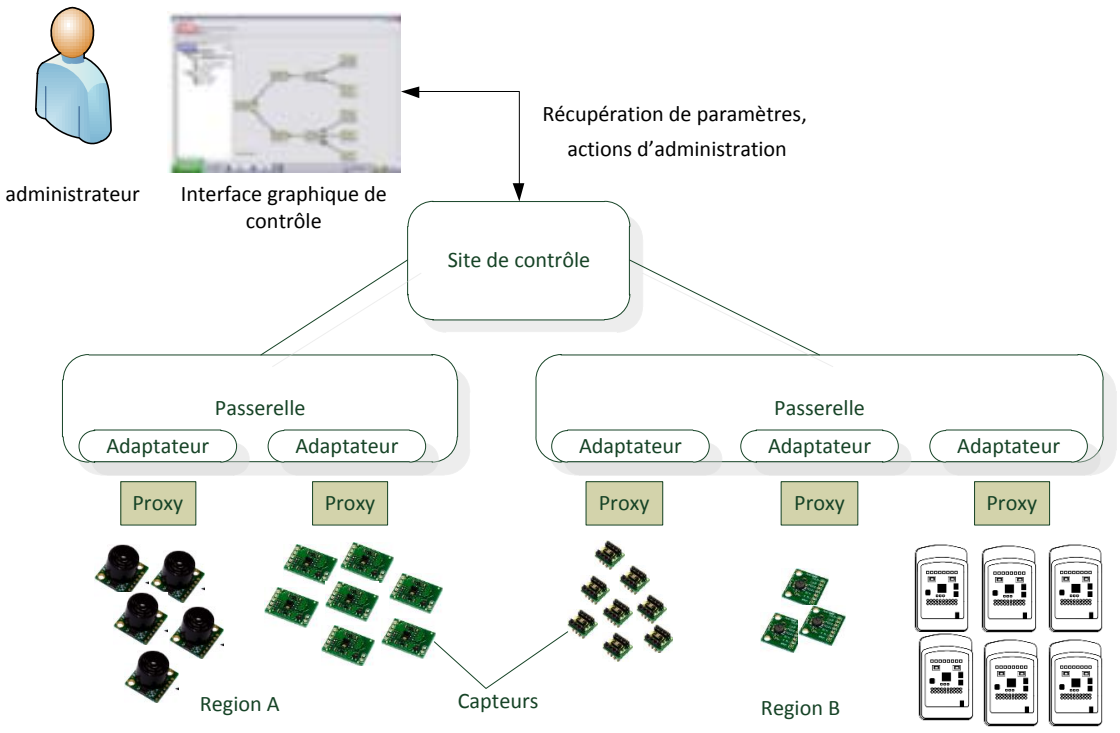


FIGURE 2.2 – Architecture hiérarchique de XSStreaMWare

2.3.5 L'intergiciel XSStreaMWare de gestion des réseaux de capteur

2.3.5.1 Présentation globale d'XSStreaMWare

XSStreaMWare est un intergiciel [26] orienté services qui est spécialisé dans le contrôle des réseaux de capteurs hétérogènes. Il permet de déclencher des opérations de gestion tels que recevoir/envoyer des paramètres vers les capteurs, installer/désinstaller des firmwares ou modules applicatifs et diagnostiquer l'état du matériel. Pour cela, il utilise un modèle de données générique qui permet de gérer des capteurs de différents types et provenant de divers constructeurs. Son approche orientée services fournit un cadre logiciel dit "plug & manage".

XSStreaMWare est basé sur une architecture distribuée hiérarchique composée de quatre couches représentée en figure 2.2. Le site de contrôle (*Control site*) est le point d'entrée du système de contrôle des réseaux de capteurs. Il constitue le plus haut niveau de l'architecture hiérarchique. Il est responsable du contrôle d'un environnement global qui est en général décomposé en plusieurs régions. Chaque région est contrôlée par une passerelle (*Gateway*) qui héberge plusieurs adaptateurs (*Adapters*). Ces adaptateurs font l'interface entre les logiciels propriétaires développés par les constructeurs des capteurs appelés *proxies* et la passerelle. Enfin, les *capteurs* sont physiquement distribués à travers

26. en anglais : middleware

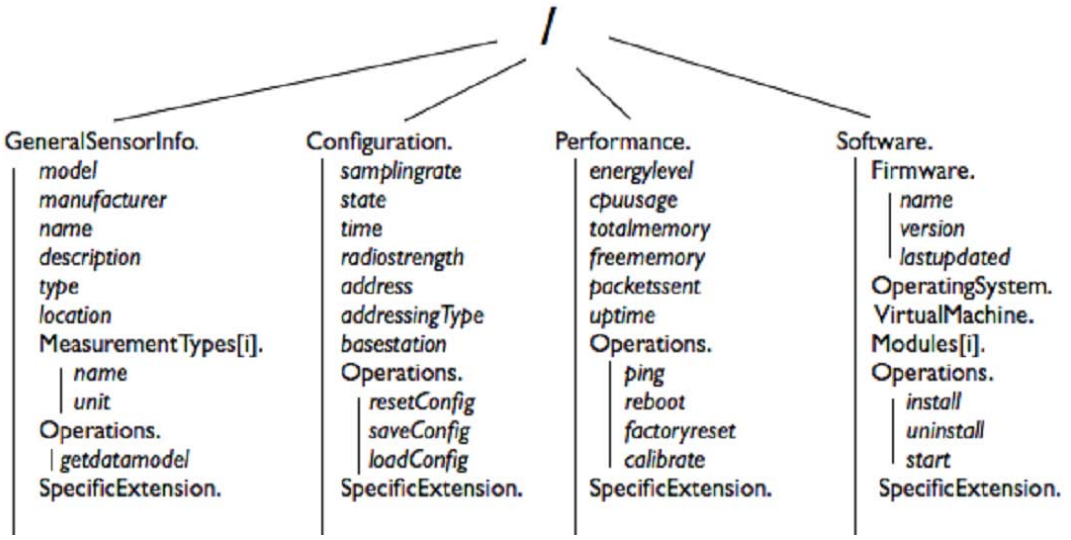


FIGURE 2.3 – Hiérarchie du modèle générique XSStreaMWare : le DATAMODEL

une région géographique prédéfinie et forment le niveau le plus bas de la hiérarchie.

2.3.5.2 Hiérarchie du modèle générique XSStreaMWare : le DATAMODEL

La figure 2.3 représente la vue hiérarchique du *DATAMODEL*, le modèle générique de données introduit par XSStreaMWare. En effet, les protocoles de gestion d'XSStreaMWare spécifient des modèles de données génériques aussi appelés des bases d'information de gestion **MIB** [27] afin d'exécuter les opérations de gestion sur une structure bien définie. Quatre familles sont définies en fonction du domaine fonctionnel correspondant :

- **General Information** : cette famille contient des informations générales sur les capteurs comme le nom de matériel, le constructeur, le modèle, le type, la localisation, la description et aussi des opérations globales comme la récupération du *DATAMODEL* supporté par le type de capteur.
- **Configuration** : cette famille contient les paramètres réseaux, système et applicatifs. Par exemple, la table des voisins du capteur, l'adresse logique ou physique, la définition des seuils pour des alarmes, les protocoles et algorithmes supportés par le matériel, le taux d'échantillonnage. Les opérations correspondantes sont aussi définies comme par exemple réinitialiser, sauvegarder ou charger la configuration.
- **Software** : cette famille contient des informations sur les capteurs et les opérations pour le déploiement de firmwares, de modules logiciels internes au capteur ou pour le contrôle du cycle de vie du capteur (changement en différents modes de fonctionnement). Par exemple, il peut s'agir d'une installation d'un nouveau firmware, d'une mise à jour d'une machine virtuelle interne ou de l'installation/désinstallation de modules logiciels.
- **Performance** : cette famille contient les paramètres qui doivent être continuellement observés, comme le délai de communication, le niveau d'énergie, l'utilisation du processeur ou de la mémoire. Les opérations correspondantes sont des tests radio, des calibrations automatiques ou des fonctions de bas niveau pour redémarrer

27. Management Information Base

le capteur ou pour le réinitialiser avec les paramètres constructeur par défaut.

Les trois opérations de gestion génériques utilisées par XSStreaMWare sont **GET**, **SET** et **ACT**. GET permet de récupérer des paramètres du DATAMODEL, SET permet de les modifier et ACT permet de faire appel aux *operations* définies dans le datamodel.

2.3.6 Les logiciels patrimoniaux existants

Les nouveaux concepts introduits par l'autonomic computing soulèvent en parallèle un nouveau paradigme de programmation. En effet, la boucle de contrôle autonomique (cf. section 2.2) et le modèle MAPE-K proposé pour l'intégration de cette boucle est surtout destiné aux futurs systèmes distribués. Leur développement s'inscrira dans ce nouveau modèle pour satisfaire les futurs besoins en gestion autonomique. Mais qu'en est-il des systèmes déjà existants ? Ce problème de prise en compte des systèmes propriétaires déjà en production a été soulevé par Kaiser en 2005 [20], rappelant que de nombreux travaux prouvent que la robustesse d'un système n'est améliorée qu'avec une analyse dynamique des mesures du fonctionnement du système. Ces analyses déterminent les modifications et adaptations appropriées au cours de la vie du système. Il mentionne qu'un des problèmes des futurs systèmes compatibles avec la boucle MAPE-K est qu'ils risquent de mélanger les codes spécifiques aux mesures et analyses (tâche de "monitoring" M et d'analyse A de la boucle MAPE-K) avec des codes spécifiques au problème traité par le système.

L'environnement logiciel présenté dans cette partie et sur lequel nous nous appuyons pour les expériences restera inchangé au niveau du fonctionnement interne. Il n'y a donc pas de risque de mélange entre les codes spécifiques au domaine de gestion autonomique et ceux concernant le logiciel administré. Même si certains des logiciels utilisés ont un code source ouvert, on peut assimiler dans ce cas ces logiciels à des boîtes noires ayant un code source fermé et non modifiable. On parle alors de **logiciel propriétaire** ou encore de **logiciel patrimonial** [28].

2.3.7 L'application DIET

DIET [29] [21] est un ordonnanceur de tâches de calculs développé au sein du projet INRIA GRAAL au LIP (ENS de Lyon) et au LIFC (Université de Franche-Comté). Il est constitué d'un ensemble de serveurs de calculs répartis sur des "clusters". DIET permet d'offrir un accès à des services de calculs à des clients répartis sur différents réseaux. La localisation des ressources se fait grâce à plusieurs ordonnanceurs appelés agents, eux-mêmes répartis sur les réseaux des différents "clusters". Les agents sont de trois types et organisés de façon arborescente : MA (Master Agent), LA (Local Agent) et SeD (Server Daemon). La figure 2.4 montre l'architecture arborescente de DIET.

L'agent **MA** est directement relié aux clients. Il reçoit les requêtes de calculs des clients et choisit un ou plusieurs SeD ayant la capacité de résoudre le problème. Il a une vue globale sur toute l'architecture déployée.

L'agent **LA** constitue un niveau intermédiaire dans l'arboresence DIET. Il peut être le lien entre un MA et un SeD, entre un autre LA et un SeD ou entre deux LA. Notons que ce niveau intermédiaire est optionnel, les SeD pouvant être liés directement au MA. Le

28. En anglais : **legacy** software
29. Distributed Interactive Engineering Toolbox

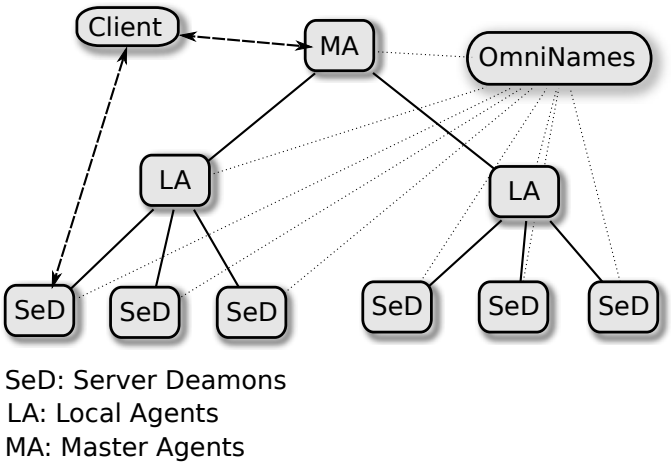


FIGURE 2.4 – Architecture de DIET

but du LA est de diffuser les requêtes et les informations entre le MA et les SeD. Il tient à jour une liste des requêtes en cours de traitement et pour chacun de ses sous-arbres, le nombre de serveurs pouvant résoudre un problème donné, ainsi que des informations liées aux données (taille des données de calcul).

Un **SeD** est le point d'entrée d'un serveur de calcul sur un "cluster". Il gère l'ensemble des ressources qui lui sont allouées. Il peut aussi bien s'agir d'un processeur que d'un "cluster". Il tient à jour une liste des données disponibles sur un serveur, une liste des problèmes qui peuvent y être résolus et des informations concernant sa charge (charge processeur, mémoire disponible, taille du disque). Les SeD sont essentiellement chargés d'effectuer une sous-partie du calcul global en vue de la résolution du problème soumis par le client.

Un **client** est une application qui utilise DIET pour résoudre des problèmes. Différents types de clients sont en mesure de se connecter à DIET à travers un MA : depuis une page web, un environnement de résolution de problème tel que Matlab ou directement depuis un programme écrit en langage C ou en langage Fortran. Des interfaces permettent d'utiliser d'autres langages tels que java ou d'exécuter les programmes nécessaires pour la résolution du problème.

Pour soumettre un calcul, un client de DIET doit se connecter au MA le plus approprié ou le plus proche. Une fois que son MA est identifié, le client peut lui soumettre un problème. Pour choisir le serveur le plus approprié pour résoudre ce problème, le MA propage une requête dans ses sous-arbres afin de trouver à la fois les données impliquées et les serveurs capables d'effectuer l'opération demandée. Les opérations sont par exemple des produits matriciels, sommes matricielles etc. Ensuite, le MA renvoie l'adresse du serveur choisi au client et effectue le transfert des données impliquées dans le calcul. Le client communique ses données locales au serveur et la résolution du calcul peut être éffectuée. Les résultats pourront être renvoyés au client en fonction du problème.

Le déploiement d'une architecture DIET nécessite un service de nommage CORBA (omniNames). Ce service doit être impérativement démarré avant les autres agents DIET (MA, LA, SeD) afin que ceux-ci puissent s'enregistrer auprès de lui sous une référence. Cet enregistrement permet par exemple à un client de retrouver un MA et à un LA de retrouver son père dans l'arborescence. Après le démarrage du service de nommage, l'agent MA

peut être démarré. L'agent localise le service de nommage et s'enregistre. Par la même procédure, les LA puis les SeD sont démarrés dans l'ordre approprié. Chaque agent s'enregistre dans l'annuaire du service de nommage et utilise ce dernier pour localiser son père dans l'arborescence.

Dans cette thèse, nous utilisons DIET comme démonstrateur couplé à notre approche d'informatique autonomique pour montrer d'une part une optimisation de performance et d'autre part une gestion des pannes pouvant se passer d'administrateurs humains.

2.3.8 Les applications de simulation électromagnétiques

2.3.8.1 Les sciences CEM : TLM et SCT

La science **CEM** [30] résout numériquement un ensemble complexe d'équations de Maxwell en utilisant des ressources informatiques. Ces solutions décrivent les interactions physiques et les phénomènes entre les particules chargées et les matériaux. La compréhension de ces phénomènes permet de concevoir, entre autres, des antennes, des microcircuits électroniques ou des systèmes fibre optique. L'évolution rapide des techniques **CEM** a permis d'atteindre un niveau de prédiction très précis du comportement électromagnétique (notamment pour les antennes) mais reste extrêmement gourmand en temps de calcul et complexe à mettre en place. En effet, les techniques **CEM** définissent plusieurs critères qui peuvent être utilisés pour ces simulations : le domaine (fréquentiel, temporel), la dimension (2D, 3D), le type d'équations requises (intégrales, différentielles), la taille de la discrétisation ou la modélisation (par ligne de transmission **TLM** [31] [22] ou par changement d'échelles **SCT** [32] [23]).

2.3.8.2 L'application YatPac

Yatpac [33] est un logiciel de simulation électromagnétique basé sur la méthode de calcul TLM [34] [22] développé à l'Institut de l'Ingénierie des Hautes Fréquences à l'Université de Munich en Allemagne. Ce logiciel permet de caractériser un comportement électromagnétique dans un domaine temporel défini. Il peut simuler diverses structures comme des guides d'ondes, des lignes de transmission, des circuits micro-ondes ou des antennes. Ce logiciel est décomposé en plusieurs modules qui communiquent en s'échangeant des fichiers. Les principaux modules sont :

- **YatGUI** : est une interface graphique pour préparer les structures des simulations électromagnétiques.
- **Yatpre** : est un préprocesseur formattant les données des fichiers décrivant les structures à simuler en entrée pour les transformer en modèle TLM.
- **Yati2of** : est un prévisualisateur des strucures sous la forme de modèle TLM.
- **Yatsim** : est le simulateur utilisant la méthode de calcul TLM.
- **Yatspar** : est un outil de post-processing permettant de sélectionner et formatter les résultats d'une simulation pour sa future utilisation ou visualisation.

30. Computational ElectroMagnetics
31. Transmission Line Matrix
32. Scale Changing Technique
33. Yet Another TLM Package
34. Transmission Line Matrix

– **Yatvis5d** : est un outil de visualisation des résultats des simulations.

En fonction de la simulation à effectuer, les différents modules peuvent être lancés dans diverses configurations. Ce logiciel est utilisé au LAAS-CNRS de Toulouse par le groupe MINC [35] pour la simulation de grandes structures ou le calcul des meilleures structures qui peuvent atteindre un comportement électromagnétique voulu. La gestion manuelle du déploiement des modules, de toutes les configurations, de la récupération des résultats est un processus long et sujet à de multiples erreurs. De surcroît, lors de pannes de machines, la détection et la relance manuelle des morceaux de la simulation échouée est fastidieuse. Nous apporterons une approche autonomique pour répondre aux besoins de la complexité grandissante de la gestion de ces simulations, tant du point de vue de la configuration globale que de la réparation en cours d'exécution.

2.4 Positionnement par rapport à l'existant

Dans cette section, nous nous positionnons par rapport aux travaux existants. Dans notre cas, nous mettons l'accent sur les approches qui relèvent de l'objectif de cette thèse : donner la possibilité à des systèmes existants d'accéder aux propriétés de l'auto-gestion ou "self-management" (voir section 1.2.3). Pour cela nous étudions pour chacune des approches concernées comment elles se positionnent par rapport aux quatre objectifs du "self-managmenent" : le **"self-healing"**, le **"self-configuring"**, le **"self-protecting"** et le **"self-optimizing"**. Nous étudions aussi si ces approches permettent de prendre en compte les systèmes déjà existants sans avoir à modifier leur fonctionnement, c'est à dire leur **degré de généricité**. En effet, certaines approches proposent des architectures de programmation qui fournissent des possibilités de "self-management" à condition d'implémenter des interfaces de programmation au sein du système concerné en y rajoutant du code, ce qui modifie le fonctionnement du système en question.

2.4.1 Degré d'autonomicité et degré de généricité

Ganek considère qu'il est important d'évaluer le **degré d'autonomicité** des systèmes [24] soit en décrivant dans quelle mesure ils intègrent les quatre propriétés du "self-management", soit par une échelle plus précise allant d'une gestion totalement manuelle à une gestion autonomique. Cette échelle se compose de cinq niveaux :

– Niveau 1 - la gestion basique [36] : chaque élément du système est géré indépendamment par un expert qui le met en route, le surveille et éventuellement le répare ou le remplace.
– Niveau 2 - la gestion aidée [37] : des technologies d'administration sont utilisées pour collecter des informations disparates dans de grands systèmes, afin de faciliter le temps qu'il faut pour les administrateurs pour faire une synthèse.
– Niveau 3 - la gestion prédictive [38] : le système lui-même peut reconnaitre des motifs pouvant l'amener dans des situations instables et peut prévoir une meilleure confi-

35. MIcro et Nanosystèmes pour les Communications sans fils
36. En anglais : basic level
37. En anglais : managed level
38. En anglais : predictive level

guration. Le système donne ensuite des conseils aux administrateurs sur l'ensemble des actions qu'ils peuvent mener.
- Niveau 4 - la gestion adaptative [39] : le système est guidé par des SLAs [40] pour choisir les meilleures opérations d'administration. Chaque action est choisie en fonction des opérations et peut soit être donnée à l'administrateur pour qu'il l'effectue, soit être exécutée automatiquement.
- Niveau 5 - la gestion autonomique [41] : toutes les opérations du système sont gouvernées par des politiques de haut niveau appelées "business policies". Les administrateurs n'interagissent avec le système qu'en observant les processus métiers appelés "business processes" ou en redéfinissant les objectifs globaux.

Pour notre positionnement, nous avons donc choisi de placer les approches existantes suivant deux facteurs : le nombre de propriétés de "self-management" considérées et la généricité. La figure 2.5 montre notre vision globale des approches suivant ces deux facteurs. L'axe horizontal dénote le **degré de généricité** et l'axe vertical le **degré d'autonomicité**.

Pour le premier facteur, nous étudions le nombre de "self-*" impliqués dans la gestion du système. Nous prenons aussi en compte certains systèmes n'ayant aucun "self-*" mais qui peuvent fournir une base pour la gestion, comme les systèmes d'observation ou les systèmes de gestions spécifiques à des domaines mais entièrement manuels.

Pour le deuxième facteur, notre échelle de généricité varie entre une gestion totalement intégrée dans le système à une gestion générique sans modification des logiciels gérés.

- Pour la **gestion intégrée**, par exemple avec une forte imbrication dans le code source, elle est spécifique à l'application et ne peut donc s'appliquer à d'autres applications que très difficilement. Certains concepts ou idées peuvent néanmoins servir d'inspiration.
- Le niveau supérieur est une **gestion appliquée** à un domaine de compétences comme par exemple les réseaux de capteurs ou le stockage de données. Les différentes applications du domaine en question sont supportées par l'approche concernée.
- Vient ensuite la **gestion générique par API**, c'est à dire qui supporte une grande variété d'applications touchant à plusieurs domaines, mais avec modification des logiciels à gérer. Il s'agit en général d'approches fournissant un support de développement [42] sous forme d'API [43] que les logiciels doivent implémenter pour accéder aux propriétés de "self-management".
- Le dernier niveau constitue **la gestion générique sans modification** des logiciels patrimoniaux. Le code source des applications reste inchangé, ce qui donne accès pour les applications propriétaires (dont on ne peut modifier le comportement) aux propriétés de "self-management".

39. En anglais : adaptative level
40. Service Level Agreements, en français : accords pour des niveaux de services
41. Autonomic level
42. En anglais : a development framework
43. En anglais : Application Programming Interface.

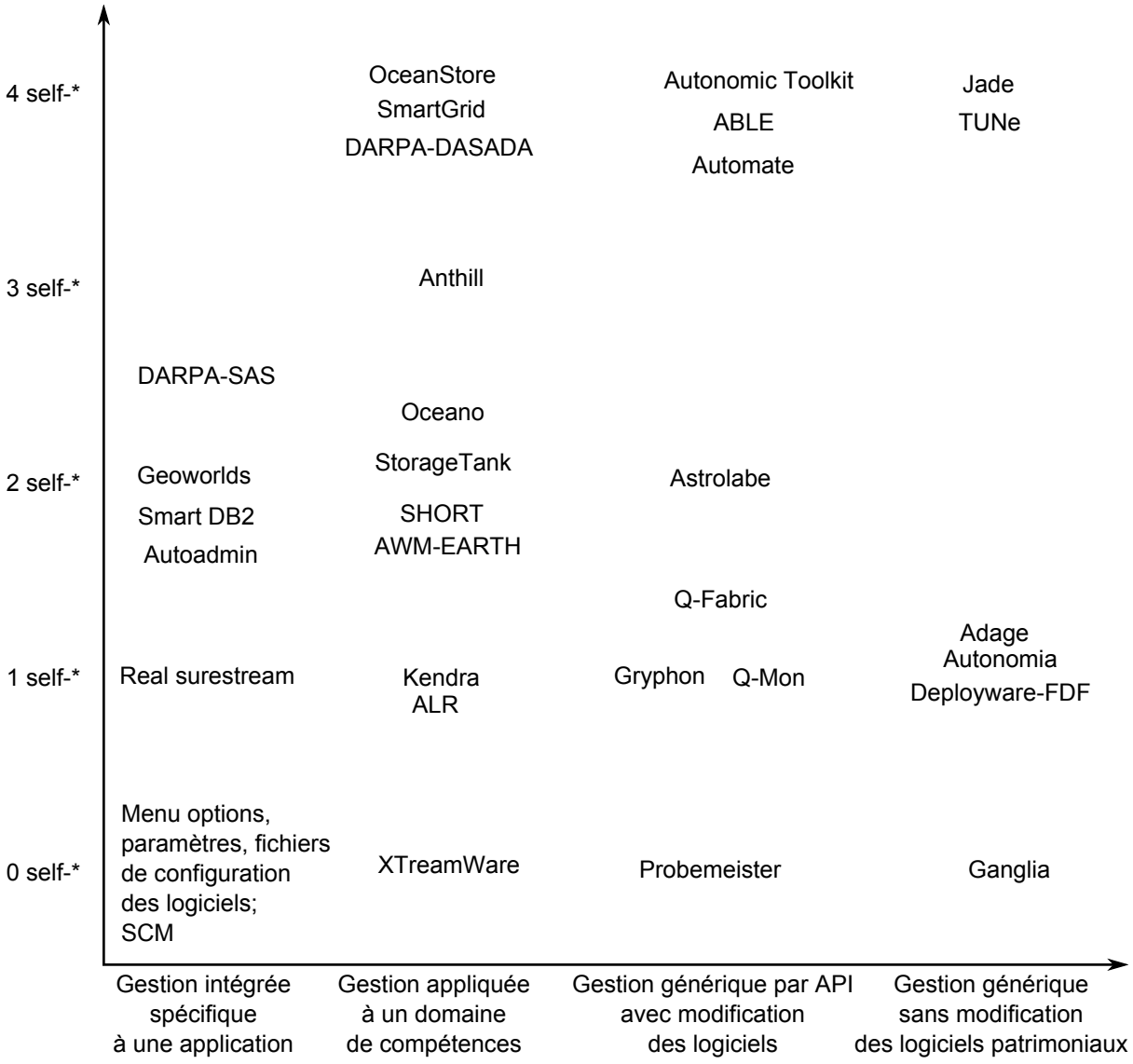


FIGURE 2.5 – Placement des approches de gestion suivant le degré d'autonomicité et leur généricité

2.4.2 Exemples d'approches existantes

2.4.2.1 Niveau 1 : Gestion intégrée spécifique à une application

Nous commençons par la gestion intégrée spécifique à une application. Si cette gestion est totalement manuelle (*0 "self-*"*), c'est qu'il s'agit en général des **menus options, paramètres** ou des **fichiers de configuration** de ces logiciels qui permettent de contrôler le fonctionnement de l'application par une intervention manuelle. Pour une gestion d'un parc d'applications similaires, il existe des outils de génération de configuration automatique dits **SCM**[44] [25]. Pour une gestion de configuration d'applications réparties, il existe aussi des outils gérant le synchronisme pour la gestion des configurations en parallèle [26]. Cette dernière approche, bien que toujours manuelle, fournit à l'administrateur

44. Software Configuration Management

humain, à l'aide d'un algorithme distribué et de méthodes de vérification formelles, les étapes menant à une configuration correcte et cohérente du système dans sa globalité. En effet, les applications distribuées ont besoin d'ordre de configuration (des éléments doivent être configurés et démarrés avant d'autres).

En montant d'un niveau dans le degré d'autonomicité (*1 "self-*"*), de nombreux exemples de "self-optimizing" sont présents dans les systèmes de streaming où le codec audio ou vidéo change en fonction des fluctuations de débits. Le but de ces systèmes est de continuer la diffusion des médias avec la plus haute qualité possible, comme avec le codec dynamique **Real Surestream** [27]. Comme le montre la figure 2.5, l'approche Real Surestream ne fait intervenir qu'un "self-*" (ici le "self-optimizing") et est intégrée à l'application de diffusion propriétaire RealPlayer.

Toujours dans la gestion intégrée spécifique à une application, mais un cran plus haut dans l'autonomicité (*2 "self-*"*), nous avons classé l'approche proposée par **Geoworlds** [28] et **smart DB2** [29].

La première est issue d'un outil GIS [45] d'analyse intelligente de données cartographiques en temps réel. Cet outil est décomposé en un ensemble de services communicants par Jini [30]. Les premières versions de Geoworlds voyaient certains de leurs services s'arrêter inopinément ou se bloquer. En effet, les services de Geoworlds utilisent des mécanismes de rapatriement d'information de plusieurs sources et notamment de plusieurs sites internet, qui font souvent l'objet de ralentissements ou d'indisponibilités (à cause de pannes réseaux, d'attaques de déni de service ou de panne de serveur). Dans ce cas de figure, la seule possibilité était de redémarrer manuellement le service Geoworlds tentant d'accéder aux ressources externes indisponibles. La version suivante utilise une approche intégrée de "self-healing" qui détecte le service bloqué ou arrêté et qui le redémarre automatiquement. De plus, les premières versions envoyaient certaines données à des serveurs de calculs. Le choix du serveur de calcul était aléatoire et dans le cas d'utilisation à plus grande échelle cela avait pour conséquence de ne pas répartir la charge et de globalement ralentir l'outil. La version suivante utilise une approche qui intègre la propriété de "self-optimizing" à l'aide d'un retour d'information sur la charge de chaque serveur pour faciliter le choix et optimiser globalement la performance du système.

La deuxième approche, Smart DB2, est similaire à celle de Geoworlds et consiste à réduire les interventions humaines et le coût de gestion des serveurs de base de données DB2 d'IBM. De la même manière, la propriété de "self-optimizing" est amenée par le calcul d'un index qui s'adapte au fur et à mesure de la construction de celui-ci et qui s'adapte aussi aux requêtes entrantes, afin d'optimiser le temps de réponse. La propriété de "self-healing" est possible grâce à des mécanismes de récupération des données en cas de désastre [46] qui s'enclenchent de manière automatique. On peut noter aussi l'approche similaire **Autoadmin** [31] pour les bases de données Microsoft SQL Server.

Comme nous l'avons vu en section 2.1.2, les premiers systèmes adaptatifs contiennent quelques propriétés de "self-management". Si nous reprenons par exemple le **SAS de la**

45. Geographical Information System
46. En anglais : disaster recovery

DARPA, il met l'accent sur le "self-protecting" et le "self-configuring". Le "self-protecting" est géré car une certaine intégrité est assurée par la configuration rapide à l'établissement de la transmission pour ne pas perdre le début du message. Le "self-configuring" reconfigure en continu les fréquences et débits des éléments du système en cours de fonctionnement. Le "self-optimizing" est considéré en partie par la négociation des communications avec le voisinage. En effet, le système n'est pas optimisé dans sa globalité, mais plutôt localement par zone géographique en ajustant les configurations lien par lien, en fonction de l'environnement à traverser. Au niveau du degré de généricité, ce projet est une gestion intégrée aux équipements que portent les soldats sur le terrain et reste donc spécifique à une application militaire.

2.4.2.2 Niveau 2 : Gestion appliquée à un domaine de compétences

XSStreamWare [32] n'intègre pas de propriétés de "self-management" (*0 "self-*"*) mais fourni un cadre intéressant en ce qui concerne la gestion appliquée au domaine des réseaux de capteurs. En effet, XSStreamWare adopte une approche orientée services OSGi[47] [33] pour la gestion d'un grand parc de capteurs hétérogènes. L'idée de base est de cacher l'hétérogénéité des capteurs à l'aide de services génériques d'interrogation. Aussi, la dynamicité du système (l'arrivée, le départ, la modification des capteurs et des services) est prise en compte et est capable de générer des événements. La généricité est assurée par un modèle simple de trois actions GET, SET et ACT et de propriétés de localisation des capteurs. Typiquement, l'implémentation de la boucle MAPE-K viserait à connecter les événements issus de ce système de gestion aux trois actions d'administration possibles afin de lui rajouter des propriétés de "self-management". C'est une des contributions de cette thèse présentée dans le chapitre "self-configuring".

En continuant dans l'autonomicité (niveau *1 "self-*"*), de même que pour Real surestream (voir section 2.4.2.1), l'approche du système de négociation de codecs **Kendra** [34] ne considère que le "self-optimizing". Elle permet de changer dynamiquement la configuration de codecs audio ou vidéo, mais contrairement à Real surestram, elle fait intervenir une gestion qui peut s'appliquer au domaine plus large de la diffusion des médias. En effet, elle est applicable pour de nombreux codecs et de nombreux diffuseurs, à l'aide d'un service d'association et de choix automatisés en fonction de l'environnement réseau.

Les propriétés de "self-optimizing" appliquées à la gestion de la qualité de service sur les réseaux au niveau logiciel sont aussi présentes au niveau matériel. Ces approches émergentes comme **ALR**[48] (niveau *1 "self-*"*) [35] ou **AWM-EARTH**[49] [36] (niveau *2 "self-*"*) utilisent une optimisation qui prend en compte la qualité de service fournie par le réseau (la performance) et l'énergie consommée par le réseau. Elles s'inscrivent dans le cadre du "self-optimizing" appliqué pour la gestion de l'énergie (la mouvance "green").

ALR utilise une adaptation dynamique du débit des liens physiques en fonction du trafic réseau. Pour cela, une paire d'équipements est utilisée : un équipement fournissant

47. OSGi est maintenant une Alliance. Elle était précédemment connue sous le nom de Open Services Gateway initiative.

48. Adaptive Link Rate

49. Active Windows Management - Energy Aware service Rate Tuner Handling

une bonne performance mais consommant beaucoup d'énergie et un équipement fournissant une performance faible et à consommation faible. En fonction des performances demandées, l'un ou l'autre des équipements ou les deux en même temps sont allumés. Néanmoins cette approche utilise un protocole qui n'est pas générique et qui nécessite des changements architecturaux dans le réseau pour doubler les équipements.

AWM-EARTH propose des algorithmes pour trouver le nombre minimum d'équipements réseau allumés nécessaires pour satisfaire la demande de trafic. Un modèle est introduit pour la variation de trafic dans un coeur de réseau d'opérateur en fonction des heures d'une journée. L'énergie consommée par les équipements réseaux est fonction de ce modèle de variation et du taux $\gamma \in [0, 1]$ de "self-optimizing" désiré. Si $\gamma = 1$ alors il n'y a pas de "self-optimizing", si $\gamma < 1$ alors la puissance totale du réseau est basée sur la formule suivante :

$$P_i^{TOT} = \gamma.P_i^D.\left(\sum_j \frac{f^{ij}}{\eta C^{ij}}\right) + (1-\gamma)P_i^C$$

Avec pour chaque noeud i, une part de puissance constante P_i^C et une part qui varie en fonction du trafic dans le réseau P_i^D. Les flots entre deux noeuds i et j s'écrivent f^{ij}, un coefficient de sur-provisionnement η est donné, ce coefficient est donné par l'opérateur du réseau pour garder une marge entre le débit maximal injecté par tous les utilisateurs sur les liens du réseau et la capacité des liens C^{ij}. Dans [37], les auteurs proposent une approche de coopération entre les fournisseurs de contenu (**CP** [50]) sur internet et les opérateurs (fournisseurs d'accès à internet FAI ou **ISP** [51]). Chaque entité a une consommation d'énergie liée à son réseau (équipements serveurs et routeurs) : P_{CP} pour les fournisseurs de contenu et P_{ISP} pour le FAI. Le problème de "self-optimizing" consiste à minimiser la puissance totale de l'infrastructure :

$$min(P_{TOT} = \alpha.P_{CP} + (1-\alpha)P_{ISP})$$

avec $\alpha \in [0, 1]$ un paramètre de poids qui permet d'établir un compromis entre les deux entités. Et les variables du problème étant des booléens représentant l'allumage et l'extinction des équipements et des serveurs. Les fournisseurs de contenu étant géographiquement disparates, si $\alpha = 1$ l'optimisation risque de choisir des serveurs de contenu qui consomment moins (avec probablement une qualité de service dégradée) mais éventuellement très éloignés des clients. Dans ce cas l'énergie consommée par le FAI peut augmenter à cause de la distance à parcourir. Si $\alpha = 0$, l'énergie consommée par le FAI est minimisée donc les contenus sont à distance minimale mais les serveurs de l'infrastructure des fournisseurs de contenu risquent alors de consommer beaucoup plus.

La limite de ces approches est qu'il manque une évaluation qualitative et quantitative de la qualité de service réellement perdue et donc de la perception de la qualité du service rendue par l'infrastructure. Ces approches se focalisent uniquement sur la consommation d'énergie. Nous proposons une approche qui rajoute ces données quantitatives et quali-

50. Content Provider
51. Internet Service Provider

tatives.

Au niveau *2 "self-*"*, nous avons catégorisé les deux approches nommées **StorageTank** [38] et **S.H.O.R.T.** [52] [39]. Ces approches font appel aux propriétés de "self-optimizing" et "self-healing".

StorageTank est utilisé pour la gestion dans le domaine du stockage de données et offre une accessibilité universelle (compatible avec de nombreux systèmes de fichiers) et multi-platefomes des données. La propriété de "self-optimizing" est assurée par la redirection de requêtes vers des serveurs disponibles et par le choix dynamique de la connectivité (Fibre Channel ethernet, InfiniBand, SCSI [53]). La propriété de "self-healing" est assurée par un service de restauration automatique en cas de perte des données qui est basé sur les logs et par un processus de "fail-over" en cas de pannes. Le tout est contrôlé par des politiques de stockage de haut niveau accessibles par les administrateurs.

S.H.O.R.T. s'applique dans le domaine des réseaux mobiles ad-hoc. Des contraintes comme l'optimisation d'énergie sont prises en compte par le calcul, pour chaque noeud mobile du réseau, de différentes métriques pour les routes du réseau (longueur des routes, consommation d'énergie le long de la route, charge des liens et influence sur la consommation d'énergie). Une méthode est proposée pour paramétrer en continu les noeuds du réseau afin soit de maximiser la performance du réseau, soit de minimiser la consommation d'énergie. Une des contributions de cette thèse est de fournir un compromis entre performance et consommation d'énergie (dans le chapitre "self-optimizing").

Oceano [40] est une approche appliquée au domaine de la gestion de clusters pour les applications gourmandes en calcul ou pour les applications ayant une charge processeur très variable en fonction du temps (serveurs web). La particularité de cette approche est sa façon de gérer la propriété de "self-optimizing" : des politiques dictées par des contrats SLAs permettent de spécifier un niveau de service par type de cluster ou par client (utilisant un ou plusieurs clusters). Des agents d'observation sont ensuite disséminés pour contrôler les différentes métriques définies. Ces agents peuvent détecter quand certains clusters ne satisfont pas les contrats et générer des événements correspondants. Les métriques prédéfinies dans Océano sont le nombre de connections actives par serveurs dans un cluster, le temps de réponse global du cluster, le temps de réponse spécifique à un service du cluster (par exemple le temps de réponse de la partie base de données) ou la bande passante sortante totale. Quand les mesures des agents d'observation dépassent les seuils définis pour une ou plusieurs métriques pendant une certaine durée (violation du contrat SAL) ou restent en dessous des seuils inférieurs pendant un certain temps, un moteur de corrélation décide du scénario à appliquer pour remettre le système géré dans un état optimal. Pour cela, un ensemble de scénarii est prédéfini dans une base de données et comprend différentes suites d'actions comme l'allocation de nouveaux clusters depuis une ferme de clusters, l'installation de systèmes d'exploitation ou de machines virtuelles ou bien encore la reconfiguration du réseau (création ou suppression de VLANs [54] pour l'isolation du problème). Certaines de ces actions font partie de la propriété de "self-

52. Self-Healing and Optimizing Routing Technique
53. Small Computer System Interface
54. Virtual Local Area Networks

configuring" car l'installation ou la reconfiguration dépend du sénario choisi. La limite du système réside dans le fait que la définition des métriques et des scénarii se fait en dur dans le code d'Océano ou dans sa base de données. De plus, le système a besoin d'une ferme de clusters relativement homogène avec des agents d'allocation préinstallés dessus (appelés Dolphins). En effet, Océano n'est pas compatible avec des gestionnaires de ressource existants. Enfin, le système lui-même ne se contrôle pas et peut se trouver dans un état de rajout à l'infini de nouveaux clusters (pas de "self-protecting"). Dans notre approche, nous fournissons un cadre de contrôle interne qui permet d'éviter l'explosion du système.

Anthill [41] est une approche réservée exclusivement au domaine de la gestion d'applications P2P [55]. Un système Anthill est composé d'une collection d'agents autonomes qui observent l'environnement local et peuvent exécuter des actions basées sur ces observations. Ce modèle est basé sur le concept d'intelligence répartie (suivant la métaphore de l'intelligence des colonies de fourmis) appelé aussi "swarm intelligence". Le réseau formé est caractérisé par une absence de structure fixée et par une organisation autonome des liens inter-agents. La propriété de "self-optimizing" est obtenue par cette organisation autonome qui prend en compte l'état des liens et leur capacité, cette approche utilise le terme "self-organizing". La propriété de "self-healing" est assurée par un mécanisme dit "de resilience" qui permet de répartir les informations entre les agents pour éventuellement pouvoir les récupérer en cas de panne d'un des agents. Aussi, la propriété de "self-configuring" est obtenue grâce à une adaptation en continu (l'approche emploie le terme "self-adapting") qui prend en compte les pannes, l'arrivée de nouveaux agents, les configurations des agents voisins. La limite du système est que les applications doivent être intégrées dans les agents et doivent supporter la communication P2P à sauts multiples. De plus, malgré la simplicité des actions que peuvent mener les agents, Anthill est totalement dépourvu d'une coordination centralisée. La conséquence est que la recherche de la simplicité des actions distribuées peut amener le système à des comportements qualifiés "d'émergents" qui sont complexes et imprévisibles [42].

OceanStore [43] est utilisé pour le domaine de stockage distribué et persistant de données. Son but principal est l'implémentation des quatre propriétés du "self-management" appliquées à la haute disponibilité des données. Ses particularités sont pour le "self-healing" la tolérance aux pannes par la redondance des données et la réparation automatique par recopie des données récupérées. Le "self-protecting" utilise une observation géographique et peut détecter des pannes régionales ou des attaques de déni de service distribuées et réagir en conséquence. Le "self-optimizing" utilise le déplacement des données de manière pro-active en étudiant les accélérations de demandes (augmentation du taux de demandes de requêtes) par régions géographiques. Le "self-configuring" se base sur l'hétérogénéité de la plateforme matérielle en configurant de manière dynamique et par région géographique les paramètres de stockage. Cette configuration a pour contrainte de générer un identifiant unique appelé GUID [56] pour chaque objet stocké et de générer de manière autonome toutes les configurations locales pour tous les fragments de l'objet. Cette configuration autonomique qui change en fonction des pannes permet un accès

55. Peer-to-Peer, en français : Pair à Pair
56. globally unique identifier

transparent à un objet à l'aide du GUID et de cacher leur duplication ou leur localisation à l'utilisateur final.

Notons aussi le projet **SmartGrid** [44] au niveau *4 "self-*"*, dont l'approche ne s'applique qu'à la gestion autonomique des réseaux électriques [57]. En effet, les propriétés de "self-management" permettent de garantir que le réseau électrique fonctionne du mieux possible même en cas de panne (lien électrique coupé, transformateur défaillant ou centrale électrique à l'arrêt). La particularité de cette approche est que les reconfigurations s'effectuent sans intervention humaine et avec une contrainte temporelle très forte, permettant ainsi d'éviter qu'une coupure électrique ne se fasse ressentir. Pour chaque partie du réseau électrique, il y a en général deux modes d'opération : le mode normal et le mode dégradé. Pour les parties en mode normal, des agents logiciels locaux coordonnent les capacités de production électrique et les charges électriques fluctuantes demandées (fortes le jour, faibles la nuit) en optimisant le coût d'opération (allumage de cellule photovoltaïques, de turbines à vent, de micro-turbines locales, de piles à combustible). Dans le cas du mode dégradé, la région impactée par la panne est délimitée et isolée, avec éventuellement une reconfiguration des zones avoisinantes pour diminuer la charge électrique et une reconfiguration de la zone isolée avec une augmentation de la capacité de production électrique locale.

Comme nous l'avons vu dans le petit historique de l'informatique autonomique (section 2.1.2), l'approche architecturale nommée **DASADA** reste appliquée au domaine de compétence militaire, mais les concepts utilisés sont transposables pour des architectures plus génériques. Par exemple, la propriété de "self-healing" est obtenue par une intégration d'un style architectural qui spécifie des stratégies de réparation [45]. Les applications militaires qui suivent l'approche DASADA ont donc un style d'écriture défini, qui doit implémenter plusieurs niveaux de contraintes. L'idée principale est qu'au cours de l'exécution de l'application, si une violation de contrainte est détectée, la stratégie de réparation appropriée est déclenchée. Chaque contrainte est donc couplée à une stratégie de réparation. Une stratégie de réparation a deux fonctions principales : la première est de rechercher pourquoi la contrainte a été violée pour trouver la cause du problème et la seconde doit déterminer comment la réparer. La forme générale d'une stratégie de réparation s'apparente à une séquence de plusieurs "tactiques" de réparations. Chaque tactique est associée à une pré-condition qui détermine si elle est applicable pour la réparation en question. L'évaluation de la pré-condition examine plusieurs paramètres, notamment architecturaux, pour valider l'applicabilité de la tactique. Si elle est applicable, la tactique exécute un script de réparation écrit dans un langage de programmation impératif. Dans le cas où plusieurs tactiques sont applicables, la stratégie de réparation décide quelles tactiques elle doit exécuter en fonction d'une politique globale. Par exemple, cette politique peut exécuter la première tactique applicable de la séquence ou peut exécuter l'ensemble des tactiques ou utiliser un autre parcours spécifique.

Une autre particularité très intéressante est l'utilisation des transactions pour la propriété de "self-protecting". En effet, le corps d'une stratégie de réparation est typiquement entouré d'un domaine transactionnel qui permet d'annuler la réparation en cas d'erreurs

57. en anglais : Power grids

pendant la réparation elle-même. Les erreurs arrivant pendant l'exécution de la réparation sont en général dues aux scripts des tactiques qui ont échoué ou à des blocages au niveau de l'environnement (blocage de droits systèmes ou pannes successives dues aux tactiques choisie). L'annulation transactionnelle permet de remettre le système géré dans l'état initial, c'est à dire celui au début de l'exécution de la stratégie de réparation.

2.4.2.3 Niveau 3 : Gestion générique par API avec modification des logiciels

Dans cette partie, les approches sont plus génériques car elles peuvent s'appliquer à plusieurs domaines de compétence. Néanmoins, elles restent limitées par le fait qu'elles nécessitent une modification du fonctionnement du système à gérer. Il s'agit notamment de supports logiciels à implémenter pour accéder aux propriétés de "self-management".

Un exemple de modification du fonctionnement d'une application existante est l'approche proposée par **Probemeister** [46]. Il s'agit d'une instrumentation automatisée de programmes pour récupérer des métriques utilisées ensuite pour la tâche d'observation ou "monitoring" de la boucle MAPE-K. Certes le logiciel résultant est capable de fournir des métriques, mais il est considérablement ralenti (par exemple dans le cas d'application de calcul haute performance) par "l'overhead" [58] nécessaire pour la récupération de ces métriques.

L'approche **QMON** [47] quant à elle rajoute un degré d'autonomicité supplémentaire (le "self-optimizing") par rapport à Probemeister. Ce système permet de faciliter l'observation des applications autonomiques. Il adapte la fréquence d'observation et le volume des données transférées pour minimiser le surplus d'information ou "l'overhead" tout en maximisant la pertinence des données recueillies. C'est en quelque sorte un système d'observation semi-autonomique pour faciliter la construction d'autres systèmes autonomiques.

Gryphon [48] apporte à la tâche d'observation une notion de prioritisation des événements, ainsi que de transformation ou d'agglomération des événements. Il s'agit d'un système de gestion événementielle auto-adaptable, en effet, cette approche utilise aussi des événements qualifiés de meta-événements qui déclenchent une reconfiguration de son fonctionnement interne. Les quatre fonctions de traitement des événements sont :

- la souscription et le filtrage d'événements selon des prédicats (en lieu et place de catégorisation prédéfinie des événements).
- la transformation d'événements qui convertit les événements entrants en d'autres événements suivant des fonctions de conversion.
- l'interprétation et l'agrégation d'un flot d'événements qui permet à une séquence d'événements d'être convertie en un unique événement et inversement.
- la réflectivité, qui permet de générer des événements par l'observation interne (par exemple si le taux de réception d'événements augmente trop).

Ces étapes de transformation et de prioritisation peuvent être importantes dans la mesure ou un système autonomique à grande échelle peut produire une grande quantité d'événements que la tâche d'observation doit traiter.

58. En français : le surplus de calcul

Q-Fabric [49] et **Astrolabe** [50] sont deux approches fournissant une API pour développer des applications ayant un besoin pour une ou deux propriétés de "self-management". Q-Fabric fournit un modèle de représentation de la qualité de service et des fonctions de contrôle de cette qualité, ainsi que de conseil automatisé pour le choix d'une reconfiguration (pour implémenter les propriétés de "self-configuring" et de "self-optimizing"). Astrolabe sert à collecter les états d'un système à très grande échelle (plusieurs milliers de noeuds à plusieurs millions) en fonction de zones et peut être utilisé pour implémenter les propriétés de "self-optimizing" ou "self-configuring".

D'autres approches telles que **Autonomic Toolkit** [51], **ABLE**[59] [52] ou **Automate** [53] ne font que fournir un canevas de programmation se conformant au modèle MAPE-K. Les applications doivent être programmées en se conformant strictement à ce canevas pour pouvoir profiter des propriétés de "self-management". Soulignons que Automate a aussi une petite spécialisation pour les applications tournant sur les grilles de calcul.

2.4.2.4 Niveau 4 : Gestion générique sans modification des logiciels patrimoniaux

GANGLIA [54] est un ensemble d'outils d'observation ou "monitoring" génériques, qui n'implémente pas les propriétés de "self-management". La généricité est assurée par une multitude de "plugins" écrits dans différents langages de programmation. Ces plugins servent en général à observer des ressources matérielles distribuées. Ils fournissent des métriques sur l'état des machines (pannes), des disques, de la mémoire, des processus, de la charge processeur. Ces métriques sont ensuite accessibles soit par une API pour d'autres programmes, soit par des interfaces web pour des administrateurs humains. Dans le cadre de la gestion d'applications distribuées, c'est en général des administrateurs humains qui surveillent les pannes, les surcharges à travers l'interface web et enclenchent les processus de réparation ou d'optimisation nécessaires.

L'approche **Autonomia** [55] gère le "self-healing". La généricité est assurée par un **AME**[60] qui permet de définir le comportement des processus de l'application gérée. Des agents spécialisés dans l'observation sont automatiquement couplés aux machines sur lesquelles l'application gérée va s'exécuter. Ces agents vérifient ensuite que l'application est présente en tant que processus (suivant la description donnée par l'**AME**). Si le processus n'est pas présent, un mécanisme de redémarrage du processus est enclenché. La limite de cette approche se situe à deux niveaux. D'abord de nouvelles machines doivent être accessibles à tout moment et l'approche n'est pas compatible avec les ordonnanceurs de ressource existants.

2.4.2.4.1 Le concept d'encapsulation, modèle à composants Fractal

Pour arriver à un certain niveau de généricité, il y a nécessité d'encapsuler les éléments à gérer dans des capsules ou wrappers qui définissent des méthodes d'administration génériques. Cette couche supplémentaire permet de faire abstraction des différentes fonctions d'administration propriétaires, spécifiques au logiciel administré. L'approche Autonomia rajoute

59. A toolkit for building multiagent autonomic systems
60. Application Management Editor

cette couche avec l'AME, mais reste limitée car elle ne permet pas de décrire des paramètres de configuration spécifiques à l'application (par exemple dans le cadre de "self-configuring"). Les approches suivantes telles que **DeployWare** [56] et **Jade** [57] utilisent le concept d'encapsulation à l'aide d'un modèle à composants. Chaque entité logicielle à administrer est encapsulée de manière plus ou moins automatisée dans un composant et la gestion s'effectue de manière générique en utilisant les interfaces du modèle à composants. L'idée pour encapsuler les logiciels propriétaires dont le code est inaccessible est de pouvoir trouver une manière de décrire les interfaces qui permettent de communiquer ou de contrôler au maximum ce logiciel sans avoir à en modifier le comportement. Une présentation plus détaillée du modèle à composants Fractal utilisé par l'outils dans lequel nous intégrons nos contributions est donnée en annexe A.1.

2.4.2.4.2 DeployWare L'objectif principal de l'approche **DeployWare** [56], aussi connu sous le nom de **FDF**, est le déploiement autonomique des applications distribuées (utilisation de la propriété de "self-configuring" pour le déploiement autonomique). Cette approche définit trois rôles dans la gestion du logiciel : "l'expert logiciel" est chargé de définir le processus de déploiement, c'est le spécialiste de la technologie du logiciel à déployer. "L'administrateur système" donne les configurations réseaux (description de l'infrastructure matérielle de déploiement). "L'utilisateur final" utilise la console graphique d'administration DeployWare eXplorer pour administrer son application. La particularité de Deployware est qu'il propose un langage spécifique pour le déploiement (un DSL [61]) et une machine virtuelle pour ce langage. Le langage DeployWare est défini par un méta-modèle et propose une notation graphique sous la forme d'un profil UML [58]. Ce langage permet de décrire les entités logicielles d'une application qui seront exécutées par la machine virtuelle FDF. FDF est implanté sous la forme de composants Fractal représentant les logiciels à déployer et l'infrastructure matérielle. Pour déployer un logiciel, il doit être au préalable encapsulé (wrappé) dans un composant Fractal. Un formalisme est utilisé pour décrire l'encapsulation (les wrappers). Les auteurs proposent un formalisme basé sur une extension de Fractal ADL. La syntaxe XML de Fractal ADSL est abandonnée au profit d'une syntaxe simplifiée. En effet, certaines notions de Fractal sont cachées, ce qui permet aux administrateurs de gérer leurs logiciels sans avoir la connaissance du modèle à composants Fractal.

Un exemple de langage DeployWare comparé à Fractal est donné en annexe A.2. Il est à noter qu'il est possible de rajouter une description des opérations à effectuer pour déployer une application. De plus, une particularité intéressante est la vérification du processus de déploiement par validation avec le méta-modèle. En effet, le méta-modèle spécifie que chaque action du processus doit avoir son action contraire ("start"-"stop" ou "install"-"uninstall") afin de pouvoir annuler le processus de déploiement en cas d'erreur. Cela permet de rajouter au processus de déploiement un propriété de "self-configuring". Cette approche comporte plusieurs limites. En effet, l'administrateur doit fournir de manière manuelle dans le fichier de description de chaque entité logicielle, son paquetage et le noeud de déploiement. L'administrateur doit donc avoir une connaissance du type de paquetage compatible avec chaque noeud. La gestion de l'hétérogénéité n'est en effet pas

61. Domain Specific Language

automatiquement prise en compte. L'administrateur doit aussi s'assurer de la compatibilité entre le paquetage d'une entité logicielle et son noeud de déploiement pour le bon fonctionnement du paquetage et du processus d'administration. Enfin, une fois le logiciel déployé, il n'y a pas la possibilité de modifier le déploiement dynamiquement en fonction de l'environnement ("self-optimizing", "self-healing" ou "self-protecting").

2.4.2.4.3 Adage Au même niveau que DeployWare, **Adage** [59] [60] est un outil automatisant le déploiement développé par l'INRIA PARIS et à l'IRISA de Rennes. Les besoins en ressources de l'application sont données par un modèle de description appelé GADe qui permet de décrire aussi bien des applications réparties que des parallèles, voire des hybrides. Pour la description des ressources, Adage utilise un format XML inspiré du format produit par le "Monitoring and Directory Service" Globus [61], enrichi de l'information sur leur topologie. Adage établit l'association entre les éléments de l'application et des ressources, en fonction des capacités des ressources à répondre aux besoins de l'application. Il planifie et réalise l'installation et l'exécution de l'application sur les ressources de manière autonome. Cependant, cette approche n'apporte pas de politique de gestion au cours de l'exécution de l'application, une fois que celle-ci est déployée.

2.4.2.4.4 JADE Dans cette approche très intéressante, les logiciels patrimoniaux déjà existants sont pris en compte. **JADE** [57] est une plateforme spécifiquement conçue pour une approche qui offre une gestion d'infrastructures logicielles propriétaires. Le point fort est donc la généricité de l'approche par la prise en compte des applications déjà existantes. Les auteurs utilisent aussi le terme d'applications patrimoniales pour désigner un logiciel vu comme une boîte noire, uniquement accessible via son interface applicative propriétaire. JADE est essentiellement composé de deux parties : un canevas pour l'encapsulation des ressources administrées, qui leur donne une interface d'administration uniforme et un canevas de construction de gestionnaires autonomes, qui gèrent pendant l'exécution un ensemble de ressources suivant une politique particulière. On distingue trois grandes composantes qui orchestrent la gestion :

La représentation du système permet essentiellement de conserver une copie de l'architecture du système. Cette copie contient l'ensemble du système : les ressources gérées ainsi que l'application à administrer. La représentation du système et le système géré sont maintenus en cohérence, toute action ou changement d'état de l'un étant répercuté sur l'autre et réciproquement.

Le système administré est composé d'un ensemble d'entités logicielles qui constituent l'application patrimoniale. Pour avoir une vision homogène de l'environnement, les entités logicielles sont encapsulées dans des composants Fractal.

Un gestionnaire fournit l'automatisation d'un aspect de l'administration. Il existe un gestionnaire prenant en charge le déploiement avec la propriété de "self-configuring", un autre qui prend en charge les aspects d'allocation des machines. Il existe aussi des gestionnaires dits autonomiques. Un gestionnaire autonomique fournit les propriétés de "self-management" au cours de l'exécution, pour la gestion de la ressource dont il a la charge. Le principe général de ces gestionnaires est de rétroagir sur le système à partir des informations issues de la tâche d'observation. Leur usage dépend du besoin particulier des applications et de leurs contextes d'utilisation.

L'approche du modèle à composants permet d'obtenir une grande souplesse dans les possibilités d'adaptation et d'extension de JADE. Chaque gestionnaire peut être ajouté ou enlevé de façon indépendante ou bien reconfiguré afin d'implémenter telle ou telle autre politique de gestion de ressource. L'architecture de JADE est distribuée. Cependant le processus de gestion est orchestré par un gestionnaire centralisé contenant la représentation à composants de l'application. Dans cette approche, les propriétés de "self-management" sont prises en compte par des gestionnaires autonomiques qui peuvent être adaptés et réutilisés pour différentes applications. Néanmoins, l'utilisation de JADE n'est pas une tâche facile. L'utilisateur doit assimiler les concepts d'architecture à composants, en plus des concepts spécifiques de Fractal, pour gérer son application. Le manque de description plus générale pour les grands systèmes engendre le problème de passage à l'échelle lors de la description de l'application. En effet le déploiement d'une architecture incluant un nombre important d'instances logicielles peut nécessiter la création d'un long fichier de description XML. Des exemples sont donnés en annexe A.3. Dans notre approche, nous introduisons des langages qui permettent de décrire l'abstraction logicielle et les politiques de reconfiguration avec un formalisme de plus haut niveau.

2.5 Présentation du système pour l'intégration des contributions

Cette section présente TUNe, le gestionnaire autonomique dans lequel nous avons implanté nos contributions.

2.5.1 Le système de gestion autonomique TUNe : Toulouse University Network

TUNe [62] [63] [64] est un gestionnaire autonomique basé sur un modèle à composants. Sa particularité est de permettre d'ajouter des comportements autonomiques à différents types de logiciels patrimoniaux déjà existants. Il offre une vision uniforme par l'encapsulation [62] des logiciels administrés dans des composants. L'administration utilise ensuite l'interface uniformisée offerte par ce modèle à composants et un ensemble de sondes génériques ou de squelettes de sondes réutilisables pour les spécificités du logiciel. Ainsi, le modèle à composants est utilisé pour implémenter une couche de gestion qu'on appelle SR [63] au dessus de la couche patrimoniale composée des logiciels patrimoniaux gérés et des sondes (Figure 2.6).

La couche SR contient la représentation des logiciels applicatifs encapsulés et des liaisons inter-logiciels, ainsi que la représentation de l'infrastructure matérielle et des liaisons matériel-logiciels. Cette représentation est basée sur le modèle à composants Fractal et fournit une interface de gestion uniforme pour les logiciels encapsulés. Cette interface permet ainsi de contrôler l'état du composant de manière homogène en évitant de manier des interfaces de configuration complexes et propriétaires.

La couche patrimoniale est la couche où s'exécute l'application patrimoniale et les

62. En anglais : wrapping
63. System Representation

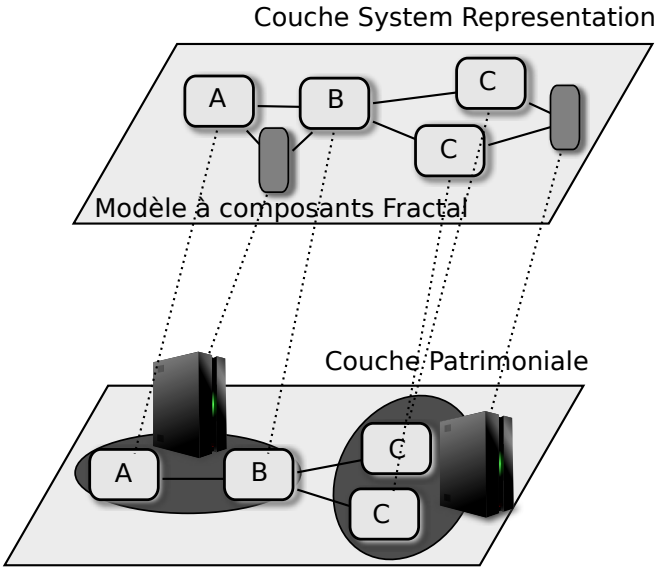


FIGURE 2.6 – Couches SR et patrimoniale dans TUNe

sondes. Elle est répartie sur plusieurs machines ou clusters et est le reflet de la couche SR dans le monde réel.

L'administrateur dispose de plusieurs interfaces lui permettant de décrire le système à gérer, le placement des sondes et le processus de gestion. Certaines se basent sur des profils UML [64] [65] car l'un des avantages d'UML est qu'il est plus "intuitif" [66] que le langage de description d'architecture basé sur des formalismes tels que XML. Le premier profil UML premet de décrire graphiquement sous forme de diagrammes le déploiement, c'est à dire l'architecture logicielle désirée (application à déployer et sondes génériques ou propriétaires) et l'architecture matérielle disponible. Le deuxième profil spécifie des règles de démarrage, de configuration et de reconfiguration. Enfin, un langage d'encapsulation des logiciels est introduit. Il permet de décrire les fonctions d'administration telles que la configuration, le démarrage et l'arrêt d'un logiciel.

2.5.2 Les phases d'administration autonomique dans TUNe

Les différentes fonctions d'administration (démarrage, configuration, reconfiguration) sont appelées et exécutées par TUNe lors de la gestion autonomique du système. Pour plus de clarté, nous décomposons ici cette gestion autonomique en quatre phases dont nous ferons référence dans la suite de cette thèse :

1. **Phase de déploiement** : lors de cette phase, le profil UML de déploiement est utilisé pour accéder aux ressources matérielles, après les avoir éventuellement réservées. Une fois les ressources matérielles accessibles, elles sont préparées à la réception des éléments logiciels (création de répertoires, installation d'un système d'exploitation). La copie des logiciels, des bibliothèques et extensions nécessaires au fonctionnement du logiciel et des sondes s'effectue ensuite en suivant un protocole de transfert spécifié dans le diagramme de déploiement.

64. Unified Modelling Language

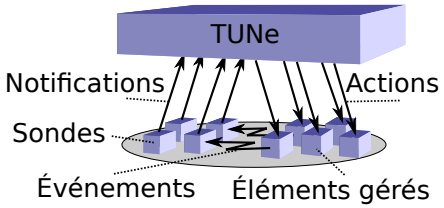


FIGURE 2.7 – Vocabulaire du triangle TUNe

2. **Phase de configuration initiale et de démarrage** : lors de cette phase, le profil de configuration et de démarrage est utilisé pour amorçer les différentes étapes de mise en route de l'application. Il peut s'agir de la création des fichiers de configuration dynamiques en fonction de l'emplacement des éléments logiciels suivi du démarrage dans un ordre précis des divers éléments.
3. **Phase de gestion par politiques** : lors de cette phase, les profils de reconfiguration sont exécutés en fonction des mesures effectuées sur le système par les sondes.
4. **Phase d'arrêt** : le système est arrêté dans un ordre précis, les différentes données sont rapatriées et les ressources matérielles nettoyées et éventuellement libérées (si elles ont préalablement été réservées).

2.5.3 Vocabulaire du triangle TUNe et fonctionnement global

La figure 2.7 montre le fonctionnement global de TUNe et représente les interactions possibles de TUNe avec son environnement lors de la phase de gestion par politiques. L'environnement comprend ici les éléments gérés et les sondes déployés par TUNe. TUNe fournit des sondes génériques que l'utilisateur peut utiliser mais il peut aussi développer ses propres sondes propriétaires. Voici le vocabulaire utilisé pour ces interactions lors de la phase de gestion par politiques :

- On parle de détection d'**événements** par les sondes ou par les éléments eux-mêmes si le logiciel patrimonial possède ses propres fonctions d'observation. En fonction du profil de déploiement, les sondes peuvent observer un ou plusieurs éléments et détecter un ou plusieurs événements en parallèle.
- La détection d'un ou plusieurs événements peut générer une ou plusieurs **notifications** suivant l'intelligence intégrée dans la sonde. Grâce à cet envoi de notifications, TUNe obtient une remontée d'informations d'observation. Ces notifications peuvent être générées soit par les sondes (les éléments spécifiques d'observation), soit par les éléments eux-mêmes. Dans la version initiale de TUNe, elles suivent une syntaxe particulière et comportent : l'identification de la réaction que TUNe doit appliquer, c'est à dire quelle règle de reconfiguration TUNe doit exécuter et deux arguments : qui a généré l'événement et qui envoie la notification.
- TUNe peut ensuite agir et réagir sur le système grâce aux **actions** descendantes sur les éléments du système (logiciels patrimoniaux ou sondes). Cette réaction est déclenchée sur réception des notifications et TUNe met en place une reconfiguration conformément à la règle spécifiée dans la notification.

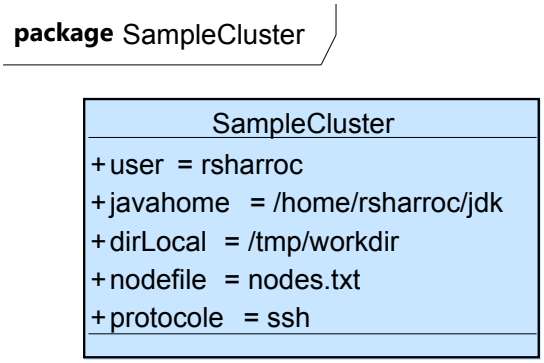


FIGURE 2.8 – Diagramme simple de description d'une infrastructure matérielle dans TUNe

Par exemple, dans le cas des sondes génériques mises à disposition dans TUNe, **l'événement** détecté par la sonde générique peut être la panne de processus applicatif c'est à dire que le PID [65] n'est plus présent et la **notification** envoyée à TUNe fait appel à des règles de reconfiguration qui exécutent des **actions** pour redémarrer le processus mort.

2.5.4 Etapes de construction d'une gestion autonomique avec TUNe

2.5.4.1 1ère étape : description de l'infrastructure matérielle

Dans le cas de l'administration avec TUNe, la phase de déploiement (voir section 2.5.2) est décrite en utilisant un profil UML basé sur les diagrammes de classes. A cet effet, un diagramme est introduit afin de spécifier l'infrastructure matérielle disponible. Dans le diagramme, une classe UML représente une famille de machines qui comporte les mêmes caractéristiques. Ces caractéristiques sont définies sous forme d'attributs de la classe et sont communes à l'ensemble des machines représentées par la classe. Un exemple de caractéristique peut être : le chemin d'installation de java, le protocole de connexion ou de copie des fichiers à distance utilisable (ssh, ftp...). La figure 2.8 montre la description d'une infrastructure matérielle très simple composée d'une famille de machines. Le nom de la famille est situé en haut de la classe : *SampleCluster*. Dans un premier temps et dans un souci de simplification, certaines caractéristiques ne sont pas mentionnées sur la figure. Parmi les caractéristiques d'une famille de machines, on peut citer :

- *user* [OPTIONNEL] : cet attribut contient le login de l'utilisateur pour se connecter au noeud distant. S'il n'est pas défini, l'utilisateur courant est utilisé.
- *javahome* [OBLIGATOIRE] : cet attribut contient l'emplacement sur le noeud distant de la machine virtuelle java nécessaire à l'exécution de TUNe.
- *dirlocal* [OBLIGATOIRE] : cet attribut contient le nom du répertoire où le déploiement s'effectue sur le noeud distant. Il correspond au répertoire d'installation des paquetages et les bibliothèques du logiciel.
- *nodefile* [OBLIGATOIRE] : cet attribut représente le nom d'un fichier qui contient les noms des noeuds ou les adresses IP des machines de la famille.

65. Process IDentification

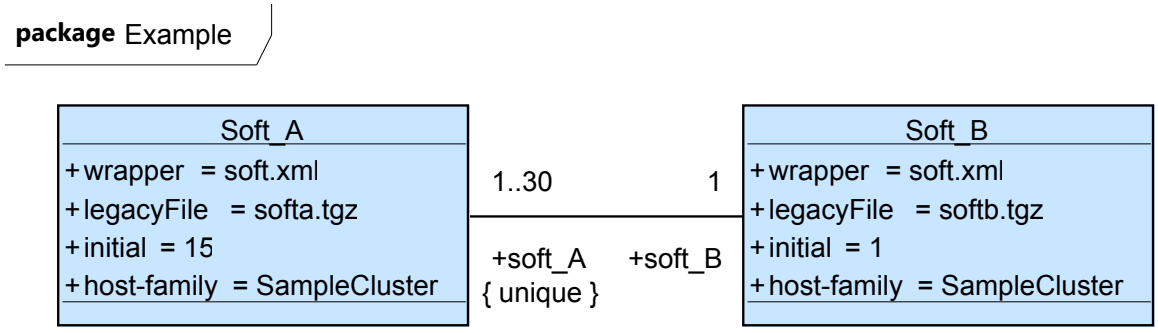


FIGURE 2.9 – Diagramme simple de description d'une infrastructure logicielle dans TUNe

- *protocole* [OPTIONNEL] : cet attribut indique le protocole de connexion sur les noeuds de la famille. Il représente aussi le protocole de copie du logiciel patrimonial et d'exécution à distance des démons sur le noeud distant. Par exemple, dans le cas de ssh, le protocole de copie est la commande scp. S'il n'est pas défini, le protocole ssh est choisi par défaut. Des plugins peuvent être implémentés dans TUNe pour gérer différents protocoles.
- *keypath* [OPTIONNEL] : cet attribut permet de définir une clé ssh personnalisée à utiliser pour la connexion ssh.

2.5.4.2 2ème étape : description de l'infrastructure logicielle

La phase de déploiement (voir section 2.5.2) nécessite que l'architecture de l'application à déployer soit décrite en utilisant le formalisme UML des diagrammes de classes. Lors du déploiement, ce diagramme est interprété pour créer l'architecture à composant Fractal et déployer la couche patrimoniale. La figure 2.9 montre la description d'une infrastructure logicielle très simple composée de deux logiciels applicatifs *soft_A* et *soft_B*. Chaque élément est représenté par une classe (une boîte) et correspond à une entité logicielle dont le nom général est écrit en haut de la boîte. Les instances porteront le même nom suivi d'un incrément i automatiquement ajouté sous la forme *NomDeClasse_i*. Les associations entre les éléments correspondent aux dépendances entre les entités logicielles. Une cardinalité est associée à chaque extrémité de l'association. Dans cet exemple, la cardinalité "1..30" exprime qu'un *soft_B* doit être relié au minimum à 1 *soft_A* et au maximum à 30 *soft_A*. De même, la cardinalité "1" exprime qu'un *soft_A* doit être relié à un et un seul *soft_B*.

Des attributs sont définis pour chaque élément. Une partie de ces attributs est prédéfinie et l'utilisateur doit en donner les valeurs, une autre partie est aussi prédéfinie mais TUNe en affecte les valeurs et enfin une partie est à la discrétion de l'utilisateur. Ces derniers lui permettent de décrire les paramètres de configuration interne d'une entité logicielle. Les attributs prédéfinis à remplir par l'utilisateur sont les suivants :

- *wrapper*[66] [OBLIGATOIRE] : cet attribut doit avoir comme valeur par défaut le nom du fichier qui contient, dans le langage d'encapsulation des logiciels, les fonctions d'administration qui lui sont associées (démarrage, configuration). Un même wrapper peut être utilisé pour plusieurs éléments (ici, *Soft_A* et *Soft_B*

66. Wrapper pourrait se traduire en fraçais par enveloppe, il s'agit ici de l'encapsulation des logiciels

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<wrapper name='genericWrapper'>

    <method name="start" key="extension.GenericStart" method="genericstart" >
        <param value="$node.dirLocal/bin/executable $srname"/>
        <param value="LD_LIBRARY_PATH=$dirLocal"/>
    </method>

    <method name="stop" key="extension.GenericStop" method="stop" >
    </method>

</wrapper>
```

FIGURE 2.10 – Exemple de wrapper générique décrit avec le WDL

utilisent le même fichier *soft.xml*).
- *legacyFile*[67] [OBLIGATOIRE] : cet attribut doit avoir comme valeur le nom du fichier tgz[68] qui contient les fichiers binaires patrimoniaux à déployer pour cet élément applicatif.
- *host-family* [OBLIGATOIRE] : cet attribut permet de déclarer sur quelle famille de noeuds les éléments logiciels doivent être déployés. Ces noms correspondent aux noms des classes du diagramme représentant l'architecture matérielle.

Les attributs prédéfinis et remplis automatiquement par TUNe (non visibles sur la figure 2.9) sont les suivants :

- *nodeName* : cet attribut représente le nom du noeud sur lequel le logiciel patrimonial a été déployé. Le noeud est alloué par un allocateur de noeuds qui est un tourniquet[69] par défaut.
- *tubeAddr* : cet attribut indique le nom d'un tube de communication inter-processus qui permet à un élément logiciel, notamment les sondes, d'envoyer des notifications à TUNe.
- *srname* : cet attribut désigne le nom unique du composant au sein du SR. Cet attribut peut être utilisé lors d'une configuration pour identifier de façon unique une entité logicielle.
- *initial* [OPTIONNEL] : cet attribut spécifie combien d'instances de l'entité logicielle doivent être déployées initialement. S'il n'est pas défini, la valeur est fixée automatiquement à 1.

Tous les noms d'attributs ne peuvent pas contenir les caractères "$", "/" ou le caractère espace car ils sont utilisés dans la définition de l'encapsulation des logiciels décrite dans le point suivant.

```java
package extension;

import java.io.BufferedReader;

public class GenericStart
{

    public void genericstart(String [] args)
    {
    ....
    }
    ....
}
```

FIGURE 2.11 – Exemple de squelette de méthode java utilisé pour l'encapsulation

2.5.4.3 3ème étape : définition de l'encapsulation des logiciels

Comme évoqué pour la description de l'infrastructure logicielle (section 2.5.4.2), chaque entité logicielle est liée à un fichier qui contient ses fonctions d'administration. Ce fichier décrit l'encapsulation du logiciel, on l'appelle un **wrapper**. Lors de la description de l'architecture logicielle, chaque classe a un attribut wrapper qui définit le nom du fichier. Ce fichier-wrapper présente l'encapsulation de l'entité permettant ainsi d'interagir avec elle par exemple pour les opérations de configuration, de reconfiguration ou de démarrage. L'encapsulation consiste à définir l'interface de contrôle de l'application. Cette interface de contrôle définit les méthodes pouvant être appelées depuis les diagrammes définissant les règles de reconfiguration lors de la phase de gestion par politiques. Un fichier-wrapper va donc contenir les méthodes avec les paramètres d'appel et les classes java qui implémentent ces méthodes. Il faut distinguer deux parties : une partie spécification décrite dans un langage de description de l'encapsulation et une partie code java qui implémente les méthodes correspondantes aux fonctions d'administration d'une entité logicielle.

La partie spécification est décrite dans un langage de description de l'encapsulation qui est appelé WDL [70]. Le WDL permet de décrire les fonctions d'administration sous forme de méthode avec des paramètres passés à ces méthodes. C'est un langage très simple qui possède une syntaxe XML qui, comme le montre la figure 2.10, utilise seulement trois balises :

- la balise *wrapper* avec l'attribut *name* qui définit le nom du wrapper.
- la balise *method* avec les attributs :
  - *name* qui est le nom de la fonction d'administration (ce nom est utilisé dans les règles de configuration et reconfiguration).
  - *key* qui est le nom de la classe java qui inclut le code de la méthode. Comme le montre la figure 2.11, cette classe est dans un paquetage d'extensions de TUNe,

67. Un legacy représente un logiciel propriétaire, patrimonial
68. TAR Gunzip, un format d'archivage compressé
69. En anglais : "round robin", la récupération des éléments se fait dans une boucle (a la fin de la liste des éléments, on revient automatiquement au début)
70. Wrapping Description Language

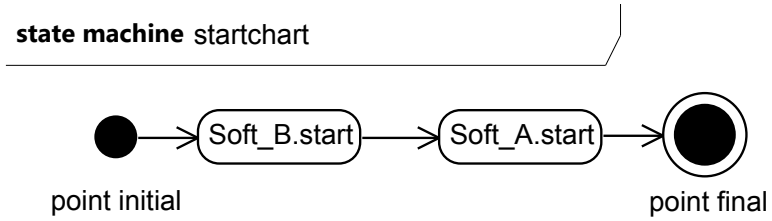


FIGURE 2.12 – Exemple de diagramme de démarrage

qui peut être étendu avec des classes personnalisées.
  - *method* qui est le nom de la méthode java dans la classe *key*.
- la balise *param* : on utilise une balise *param* pour chaque paramètre à passer à la méthode.

L'intérêt du WDL est d'utiliser la notation pointée dans la balise *param* en vue de naviguer dans le SR et passer des arguments aux méthodes java. Ainsi, dans la balise *param*, une chaine

- qui contient des symboles $ signifie qu'un ou plusieurs attributs doivent être récupérés dans le SR. Tous les attributs prédéfinis ou définis par l'utilisateur peuvent être utilisés. Cette chaîne aura comme composant source le composant sur lequel la méthode doit s'appliquer.
- qui ne commence pas par $ sera passée telle quelle à la méthode java.

Pour naviguer dans le SR, on part du composant courant comme source de la navigation. Ainsi, *$Soft_A.nodeName* va chercher l'attribut *nodeName* dans le composant *Soft_A* relié au composant courant. Si plusieurs composants *Soft_A* sont reliés, les attributs seront récupérés et séparés par des ";". Par exemple, pour l'architecture simple décrite à la figure 2.9, si on se situe au niveau du wrapper du *Soft_B* (le wrapper du composant courant), le paramètre *$Soft_A.nodeName* va récupérer le nom de la machine (attribut *nodeName*) sur laquelle s'exécute l'entité logicielle *Soft_A* qui est reliée au *Soft_B* courant.

Il est possible de suivre un chemin plus long en utilisant plusieurs "." dans la chaîne. Cette navigation est effectuée en suivant les liaisons Fractal entre les composants du SR. Le mot-clé *node* est ajouté à ce langage WDL pour pouvoir accéder aux attributs de classe correspondant à la famille de noeuds (les attributs de la classe définie par l'attribut *host-family*). Ainsi, *$node.dirLocal* représente la valeur de l'attribut *dirLocal* de la classe définie par l'attribut *host-family* du composant courant. En considérant que la figure 2.10 représente le wrapper *soft.xml* de la figure 2.9, *"$node.dirLocal/bin/executable $srname"* sera remplacé par *"/tmp/workdir/bin/executable soft_A_0"* si le composant courant est la première instance (numérotée 0) de Soft_A.

2.5.4.4 4ème étape : définition des règles de configuration

Un profil UML est utilisé pendant les trois phases qui suivent la phase de déploiement (voir section 2.5.2) : la phase de configuration initiale et de démarrage, la phase de gestion par politiques et la phase d'arrêt. Ce profil consiste en une simplification du diagramme d'états-transitions UML en le transformant en diagramme d'actions-transitions. Les trois phases utilisent des diagrammes différents : la phase de configuration initiale et de démarrage utilise un diagramme de démarrage appelé obligatoirement **startchart**.

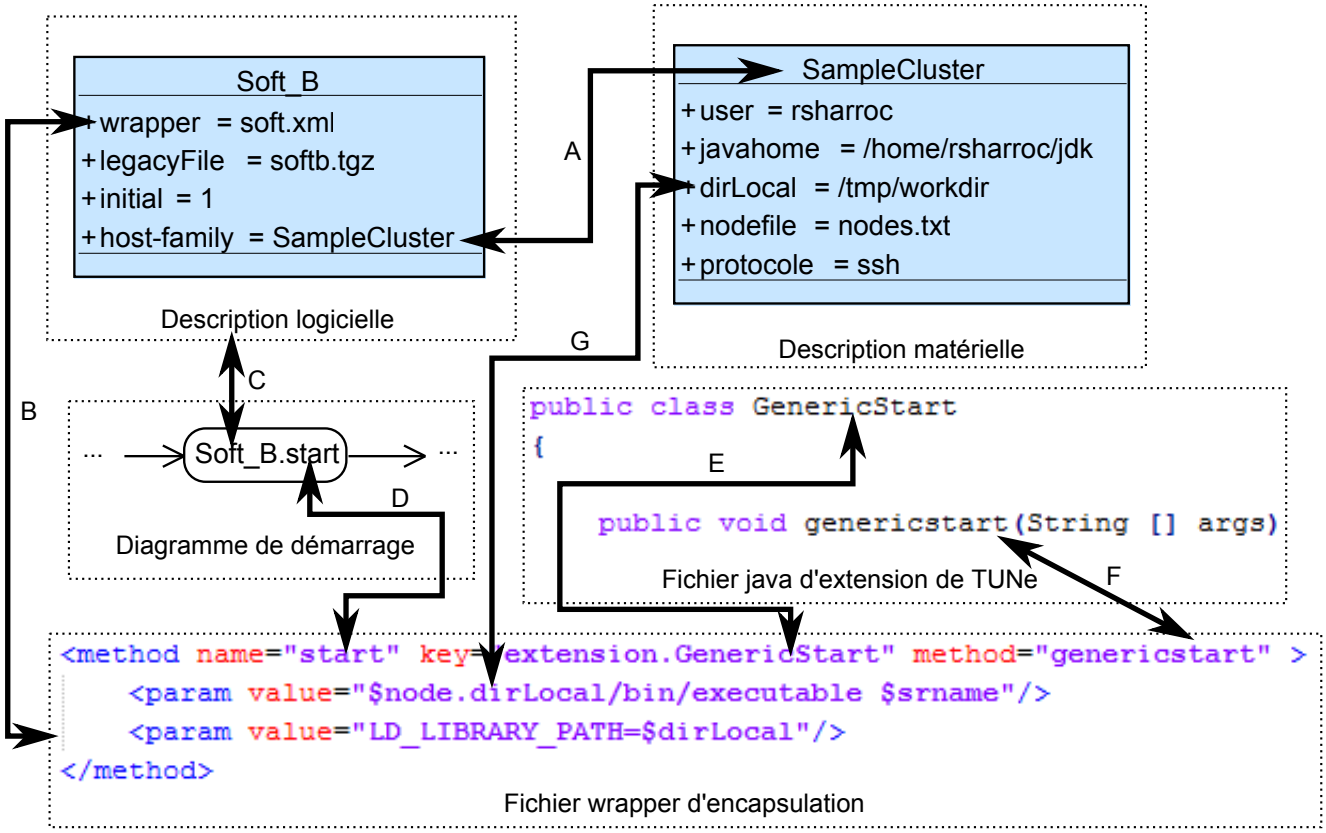


FIGURE 2.13 – Vue globale de la construction d'une gestion autonomique avec TUNe

La phase de gestion par politiques utilise des diagrammes de reconfiguration qui ont des noms définis par l'utilisateur en fonction des besoins (par exemple plusieurs diagrammes peuvent représenter des politiques de gestion différentes). Enfin, la phase d'arrêt utilise un diagramme appelé obligatoirement **stopchart**.

Ces diagrammes permettent de spécifier une suite d'actions à effectuer, l'ordre de ces actions, ainsi que le niveau de parallélisation souhaité entre elles. Chaque action correspond à une méthode définie dans le wrapper de l'entité concernée. L'ordre est respecté en suivant le chemin partant du point initial pour arriver au point final. Un niveau de parallélisation peut être atteint à l'aide d'un élément *fork* d'où partent plusieurs transitions. De même, la synchronisation des chemins parallélisés s'effectue à l'aide d'un élément *join* d'où arrivent plusieurs transitions.

Un exemple de diagramme de démarrage est donné par la figure 2.12. Il donne l'ordre de démarrage des entités logicielles de l'application décrite par la figure 2.9. L'action *Soft_B.start* applique en premier la méthode *start* sur l'entité logicielle instanciée de *Soft_B*. L'action *Soft_A.start* applique ensuite en second la méthode *start* sur toutes les entités logicielles instanciées de *Soft_A*, dans cet exemple au nombre de 15.

2.5.4.5 Vue globale de la construction d'une gestion autonomique avec TUNe

La figure 2.13 présente la vue globale de l'imbrication des quatre étapes nécessaires à la construction d'une gestion autonomique avec TUNe. Pendant la première et la deuxième étape, les descriptions logicielles et matérielles sont données. La liaison *A* représente le

lien entre les entités logicielles et le matériel sur lequel elles seront exécutées. Ce lien est décrit par l'attribut *host-family* de l'entité logicielle, il est aussi appelé *mapping*.

Pendant la troisième étape, les méthodes d'administration des entités logicielles sont définies par des fichiers *wrapper* d'encapsulation. La liaison *B* représente le lien entre une entité logicielle et son fichier *wrapper* décrivant l'encapsulation du logiciel. Ici, le *wrapper* de *Soft_B* est le fichier *soft.xml*.

Pendant la quatrième étape, les différents diagrammes nécessaires pour le démarrage, les configurations, les reconfigurations et l'arrêt sont spécifiés. Les actions des diagrammes font référence aux entités logicielles par la liaison *C* et aux méthodes qu'il faut appliquer dessus par la liaison *D*. Ces méthodes sont codées dans des fichiers java d'extension de TUNe (inclus dans le paquet extension de TUNe) et sont référencées par la liaison *E* pour le nom de la classe concernée et par la liaison *F* pour le nom de la méthode java concernée. Notons que la méthode java est terminée quand la transition vers la prochaine action correspondante est enclenchée. Enfin, les paramètres des méthodes peuvent utiliser une syntaxe pointée qui fait référence aux descriptions logicielles ou matérielles comme le montre la liaison *G*.

2.5.4.6 Introduction d'une dénomination

Pour clarifier l'utilisation des différents diagrammes et des différentes descriptions dans TUNe, nous introduisons une dénomination qui est utilisée par la suite tout le long de cette thèse. Cette dénomination est spécifique à chaque facette de gestion et chaque phase d'administration. Pour cela, nous différencions quatre types de spécifications utilisées dans TUNe :

- La spécification matérielle **HD** [71].
- La spécification logicielle **SD** [72].
- La spécification de l'encapsulation **SWD** [73].
- La spécification des politiques de gestion **PD** [74] : il s'agit de l'extension proposée ici.

Pour chacune de ces spécifications nous associons un langage de description qui peut être sous forme textuelle (utilisation d'une syntaxe XML) ou graphique (utilisation de diagrammes) :

- La spécification matérielle est décrite par un diagramme **HDD** [75].
- La spécification logicielle est décrite par un diagramme **SDD** [76].
- La spécification de l'encapsulation est décrite par un langage **SWDL** [77].
- La spécification des politiques de gestion est décrite par un diagramme **PDD** [78].

Toutes ces descriptions définissent la vue globale du système et sa gestion avec un haut niveau d'abstraction. Elles sont utilisées pour décrire les quatre phases d'administration autonomique dans TUNe. Les HDD et SDD sont utilisés pour **la phase de déploiement**.

71. Hardware Description
72. Software Description
73. Software Wrapper Description
74. Policy Description
75. Hardware Description Diagram
76. Software Description Diagram
77. Software Wrapper Description Language
78. Policy Description Diagram

Les SWDL et PDD sont utilisés pour les trois autres phases : la **phase de configuration initiale et de démarrage**, la **phase de gestion par politiques** et la **phase d'arrêt**.

2.6 Etat de l'art : synthèse

L'informatique autonomique est un domaine vaste qui soulève de nombreuses problématiques à long terme et dans de nombreux domaines. Ce concept transposé du domaine biologique vers le domaine informatique vise à procurer les quatre propriétés de **"self-management"** aux systèmes informatiques : **"self-configuring"**, **"self-healing"**, **"self-protecting"** et **"self-optimizing"**. En effet, la complexité grandissante des systèmes informatiques actuels risque d'être un frein majeur à leur développement dans les dix prochaines années. Une approche communément admise pour le développement des futures applications autonomiques est celle de l'implémentation de la boule de contrôle autonomique MAPE-K. Cette boucle décompose les tâches principales de gestion : l'**Observation ("Monitoring")**, l'**Analyse**, la **Planification**, l'**Exécution** et la **Connaissance**. Une autre problématique est celle de la rétro-compatibilité avec les systèmes propriétaires déjà existants. En effet, le framework MAPE-K est surtout destiné au développement de futurs systèmes autonomiques. Pour les approches dans la littérature, nous avons donc introduit un positionnement grâce à un degré de généricité allant de la gestion totalement intégrée aux applications jusqu'à une gestion générique sans modification des systèmes existants. Les principales contributions que nous retiendrons sont :

- Pour la propriété de **"self-healing"** : la détection des pannes avec des sondes génériques et la réparation des éléments avec par exemple le redémarrage des processus morts.
- Pour la propriété de **"self-configuring"** : la génération automatique de fichiers de configuration et la reconfiguration dynamique en cours d'exécution en fonction de l'environnement d'exécution.
- Pour la propriété de **"self-optimizing"** : une approche double qui touche aussi bien au niveau logiciel qu'au niveau matériel pour l'adaptation à des charges fluctuantes, ainsi que le concept de compromis entre optimisation de l'énergie et performance des équipements.
- Pour la propriété de **"self-protecting"** : la possibilité d'annulation des actions pour que le système reste dans un état conforme et cohérent.

Chapitre 3

Auto-guérison - "Self-healing"

Sommaire

3.1 Introduction 49

3.2 Méta-modélisation des diagrammes de spécification de politiques de gestion - "Policy Description Diagrams" (PDD) 50

3.2.1 Forme MOF "Meta-Object Facility" et forme EBNF "étendue de Backus-Naur" 50

3.2.2 Méta-modèle des PDD 51

3.3 Validation 60

3.3.1 Description des expériences 60

3.3.2 PDD utilisés pour les expériences 63

3.4 Conclusion 68

3.1 Introduction

La propriété d'auto-guérison ou "self-healing" d'un système autonomique permet de décrire la façon de gérer les dysfonctionnements d'un ou plusieurs éléments le composant. Grâce à cette propriété, le système peut maintenir son activité en surmontant les événements routiniers ou extraordinaires. Il s'agit de la **gestion des pannes** qui sont en effet très fréquentes dans un environnement composé de centaines voire de milliers de machines et d'entités logicielles. Nous rappelons qu'il existe deux types de panne : la panne matérielle et la panne logicielle. Pour gérer ces pannes, il faut détecter la défaillance d'une machine ou d'un logiciel et ramener le système dans un état opérationnel. L'application de la propriété de "self-healing" nécessite donc d'observer, de détecter les erreurs ou les situations qui peuvent se manifester plus tardivement comme des erreurs et d'initier une réparation.

Une des problématiques sur lesquelles nous nous focalisons dans cette section est de trouver une manière de décrire des politiques de réparation de systèmes distribués propriétaires. Cela soulève un double problème : celui de la généricité (gestion non-intrusive) et de la facilité de création de politiques (utilisation de concepts facilement maîtrisables) pour les administrateurs du système. Pour cela, nous avons choisi d'étendre le langage

graphique précédemment utilisé par TUNe. Ce langage est en effet de haut-niveau et permet une généricité par l'encapsulation des logiciels propriétaires.

Dans cette section, nous nous penchons particulièrement sur la **phase de gestion par politique**, appelée aussi **phase de reconfiguration** et nous introduisons particulièrement le méta-modèle des PDD qui permettent de décrire des politiques de gestion autonomiques. Nous validons ensuite notre approche à l'aide d'une expérience de réparation autonomique lors de crash de certains éléments du système DIET. Ce crash est notamment dû à une panne matérielle (arrêt d'un cluster d'une grille de calcul) qui impacte une partie des logiciels de DIET. Les politiques de réparation sont décrites avec des PDD issus de la méta-modélisation.

3.2 Méta-modélisation des diagrammes de spécification de politiques de gestion - "Policy Description Diagrams" (PDD)

3.2.1 Forme MOF "Meta-Object Facility" et forme EBNF "étendue de Backus-Naur"

Pour la description des politiques de gestion autonomique dans TUNe, nous introduisons les diagrammes de spécification de politiques de gestion "Policy Description Diagrams" **PDD**. Ces diagrammes modélisent les politiques sous forme de flot de travail ou "workflow" [1] graphique et sont inspirés des diagrammes d'activité UML [67]. Afin de formaliser de manière précise le modèle que nous introduisons, nous utilisons le concept de méta-modélisation. Cette méta-modélisation concerne aussi bien le niveau graphique (enchaînement des éléments graphiques du workflow) que le niveau textuel (syntaxe utilisée dans les éléments graphiques). Pour la méta-modélisation du niveau graphique, nous utilisons le formalisme le plus répandu de **MOF** "Meta-Object Facility" [68] et pour la méta-modélisation textuelle nous utilisons la forme étendue de Backus-Naur **EBNF** [69].

Le **MOF** est un standard issu de l'OMG [2] qui s'auto-définit et qui s'intéresse à la représentation des méta-modèles et à leur manipulation. Le principal avantage de l'utilisation de ce standard est qu'un modèle conforme à un méta-modèle **MOF** peut être sérialisé en **XMI** [3] [70], c'est à dire transformé textuellement dans un format **XM**L **I**nter-échangeable. Nous introduisons un moteur qui permet de déchiffrer ces fichiers **XMI** pour exécuter les flots de travail graphiques ainsi créés. Le standard **MOF** est situé au sommet d'une architecture de modélisation en quatre couches et nous situons notre approche à partir de la couche **M2** :

- **couche M3**, le méta-méta-modèle MOF (couche auto-descriptive).
- **couche M2**, les méta-modèles. Nous décrivons pour cette couche le méta-modèle des PDD qui formalise un cadre de création graphique des diagrammes.

1. Le mot workflow est un anglicisme. On pourrait le traduire par un flot de travaux ou flot opérationnel.
2. Object Management Group
3. XML Metadata Interchange

– **couche M1**, les modèles. Il s'agit de l'étape de création par les utilisateurs de TUNe des diagrammes PDD qui respectent le méta-modèle de la couche M2.
– **couche M0**, le monde réel. Il s'agit de l'exécution des PDD, ce qui permet de mettre en oeuvre les politiques de gestion autonomiques sur le système réel. Dans cette couche nous trouvons notre moteur de déchiffrement XMI des modèles créés en couche M1.

La forme étendue de Backus-Naur **EBNF** est une notation permettant de décrire des règles syntaxiques. C'est donc un méta-langage. Elle est utilisée dans certains livres pour décrire le langage étudié, mais également par de nombreux logiciels d'analyse syntaxique pour travailler sur des fichiers sources de plusieurs langages différents. Dans notre méta-modélisation, nous avons besoin de définir une syntaxe précise pour chaque tâche du flot de travail graphique. En effet, les différents types de tâche sont décrits par des expressions textuelles qui manipulent des listes, des propriétés, des actions ou des récupérations de paramètres.

3.2.2 Méta-modèle des PDD

3.2.2.1 Méta-modèle de la partie graphique sous forme MOF

La figure 3.1 est le méta-modèle sous forme MOF du flot de travail graphique de l'extension que nous proposons pour la spécification des politiques de gestion dans TUNe. Ce méta-modèle spécifie qu'un PDD est identifié par son nom (attribut *name*) et est composé de plusieurs noeuds et éventuellement de paramètres par les liens de composition entre le stéréotype *PDD* et les stéréotypes *Node* et *Parameters*. Les paramètres sont de type entrants (Paramètre Entrant avec la méta-classe *Input parameter*) ou sortants (Paramètre Sortant avec la méta-classe *Output parameter*) et sont tous ordonnés (à l'aide de l'attribut *order*). Les noeuds ont tous un identifiant unique (attribut *id*) et sont reliés entre eux à l'aide d'un certain nombre de transitions (stéréotype *Edge*, comprenant également un attribut *id* unique) qui peuvent être entrantes ou sortantes. Une transition entrante est donc source (association de type *Source*) d'un noeud et une transition sortante a comme destination (association de type *Destination*) le noeud suivant du flot de travail. Le nombre de transitions est représenté par les cardinalités (N,M) et dépend du type de noeud. Le tableau 3.1 fournit les valeurs de ces cardinalités. Par exemple, un noeud de type *Initial* est source d'une seule transition (sortante), un noeud de type *Fork* est source d'au minimum deux transitions (sortantes du noeud) et destination d'une seule transition (entrante dans le noeud).

Nous pouvons voir qu'un noeud peut être défini par deux stéréotypes possibles soit d'action (stéréotype *Action*), soit de contrôle (stéréotype *Control*). Le stéréotype d'action décrit les tâches à effectuer pour la politique de gestion modélisée par le PDD. Chaque tâche est décrite suivant une expression (attribut *expression* pour le stéréotype *Action*). Le stéréotype de contrôle est quant à lui utilisé pour guider le chemin d'exécution global du PDD, il contrôle la suite logique des tâches pour décrire un flot de travaux de gestion.

3.2.2.1.1 Le stéréotype Action :

Les modifications structurelles (stéréotype *Structural modification*) sont des types

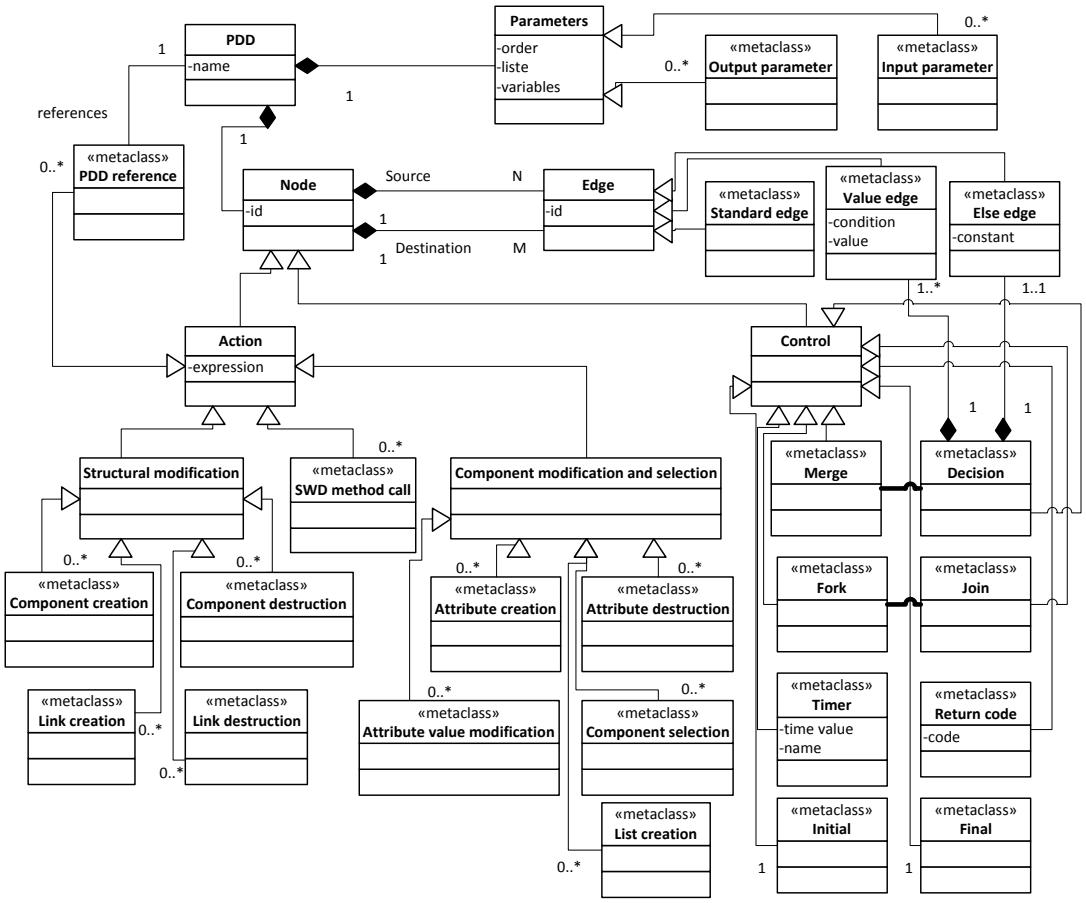


FIGURE 3.1 – Méta-modèle au format MOF des PDD

d'actions car elles représentent des tâches qui ont un impact direct sur le système géré. En effet, les quatre méta-classes issues de ce stéréotype sont la création de composants (méta-classe *Component creation*) la destruction de composants (méta-classe *Component destruction*), la création de liens (méta-classe *Link creation*) et la destruction de liens (méta-classe *Link desctruction*). Ces quatre actions de modifications structurelles modifient l'architecture du système géré en cours d'exécution et sont utilisées dans le PDD pour décrire des créations/destructions d'éléments du système et des liens inter-éléments.

Les modifications et sélection de composants (stéréotype *Component modification and selection*) permettent de décrire des créations d'attributs (méta-classe *Attribute creation*), des destructions d'attributs (méta-classe *Attribute destruction*) ou des modifications d'attributs (méta-classe *Attribute value modification*) des éléments du système de manière dynamique pendant l'exécution. Elles permettent aussi de créer une liste d'éléments (méta-classe *List creation*) ou de filtrer une liste d'éléments, c'est à dire de sélectionner une partie des éléments d'une liste suivant la valeur des attributs (méta-classe *Component*

Type de noeud	N	M
Initial	1..1	0..0
Final	0..0	1..1
Action	1..1	1..1
Fork	2..*	1..1
Join	1..1	2..*
Decision	2..*	1..1
Merge	1..1	2..*
Return code	1..1	1..1
Timer	1..1	1..1

TABLE 3.1 – Cardinalités en fonction du type de noeud

selection).

La méta-classe *SWD method call* permet de décrire des actions d'appel aux méthodes définies dans l'encapsulation des logiciels en SWDL. Ces tâches font appel aux fonctions d'administration sous forme de méthode avec des paramètres passés à ces méthodes. Ces fonctions sont par exemple le démarrage, l'arrêt ou la reconfiguration des composants en cours d'exécution.

La dernière méta-classe issue du stéréotype Action permet de référencer un autre PDD (méta-classe *PDD reference*). Une telle action permet de référencer un unique PDD (association de type *references*) par contre un PDD peut être référencé plusieurs fois par d'autres PDD.

3.2.2.1.2 Le stéréotype Control :

Tous les noeuds de contrôle étendent le stéréotype *Control*. Ils peuvent être classifiés en tant que noeuds fork/join (méta-classes *Fork* et *Join*) utilisés dans le PDD pour paralléliser le flot d'exécution des tâches. Un noeud de type *Fork* permet de créer un thread, il a donc plusieurs transitions sortantes qui créent plusieurs chemins d'exécution parallèles. Un noeud de type *Join* permet de synchroniser plusieurs threads, il a donc plusieurs transitions entrantes qui synchronisent les chemins d'exécution entrants. L'association entre le *Fork* et le *Join* identifie le fait qu'une création de thread doit obligatoirement s'associer d'une synchronisation ultérieure de ces threads. Ainsi le nombre de thread doit être égal à un avant la fin de l'exécution du PDD.

Les noeuds de décision (méta-classe *Decision*) et les noeuds de fusion (méta-classe *Merge*) créent des chemins d'exécution mutuellement exclusifs. Ces chemins ne sont parcourus que dans deux cas. Le premier cas est si la valeur définie dans les transitions sur valeur (attribut *valeur* de la méta-classe *Value edge*) est égale au retour du noeud précédent. Le deuxième cas est si la condition spécifiée est vérifiée (attribut *condition* de la méta-classe *Value edge*). Les conditions et les valeurs sont obligatoirement mutuellement exclusives. L'association entre *Decision* et *Merge* spécifie que tous les chemins créés par une décision doivent obligatoirement être fusionnés avant la fin du PDD. Notons qu'un noeud de décision est source d'au minimum une transition sur valeur (cardinalité *1..** de l'association avec *Value edge*) et obligatoirement d'une et une seule transition sinon (cardinalité *1..1* de l'association avec *Else edge*). En effet, pour éviter un blocage du flot

d'exécution, une des transitions sortantes d'un noeud de décision doit forcément s'activer si toutes les autres transitions ont des valeurs ou des conditions de passage non vérifiées. C'est à dire que si toutes les transitions sur valeur ne peuvent pas être franchies, l'exécution continuera quand même sur l'unique transition sinon de cette décision, ce qui évite un "dead-lock" [4].

Dans un PDD on retrouve aussi comme noeuds de contrôle le noeud initial qui est le point d'entrée unique (méta-classe *Initial*) et le noeud final qui est le point de sortie unique (méta-classe *Final*). D'après le tableau 3.1, il n'y a qu'une transition sortante de l'unique point d'entrée et qu'une transition entrante de l'unique point de sortie. Cela confirme que toutes les créations de thread doivent être synchronisées avant la fin du PDD et que tous les chemins issus des noeuds de décision doivent être fusionnés avant la fin du PDD. Les deux dernières méta-classes issus du stéréotype *Control* sont *Return code* et *Timer*. Un timer (méta-classe *Timer*) permet de contrôler le flot d'exécution du PDD soit en faisant une pause dont la durée est définie par la valeur du timer (attribut *time value*), soit de vérifier que le chemin d'exécution du PDD n'est pas passé à cet endroit depuis *time value*. Dans ce dernier cas, si le flot est déjà passé par ce timer il y a moins de *time value* secondes, le chemin continuera par la transition sinon sortant du noeud de décision placé juste après le timer. A la fin de l'exécution d'un PDD, celui-ci peut renvoyer un code de retour (méta-classe *Return code*, valeur donnée par l'attribut *code*), par exemple pour retourner un code d'erreur en cas d'erreur, à l'aide d'un noeud de type *Return code*. Notons que ce noeud peut être placé n'importe où dans le PDD. Lors de l'exécution du PDD c'est le dernier noeud de type *Return code* traversé qui désignera le code de retour à renvoyer. Ce code de retour est éventuellement capturé par le PDD qui a référencé le PDD renvoyant le code de retour. S'il est capturé, alors il doit y avoir obligatoirement un noeud de décision (*Decision*) placé juste après le noeud de référencement (*PDD reference*). Les transitions de valeur sortantes de ce noeud de décision peuvent alors utiliser le code de retour comme valeur, afin de créer différents chemins d'exécution possibles en fonction de la valeur retournée par le PDD référencé. Si aucune des valeurs ne correspond au code de retour, c'est la transition sinon qui est utilisée pour continuer le flot d'exécution du PDD.

3.2.2.2 Méta-modéle de la partie syntaxique sous forme EBNF

Nous introduisons ici la méta-modélisation syntaxique des expressions utilisées dans les PDD. Nous présentons pour chaque type d'action ou de contrôle les expressions qui leur sont associées sous la forme étendue de Backus-Naur EBNF. Le tableau 3.2 récapitule les caractères que nous utilisons pour cette méta-modélisation et leur signification.

En premier lieu, nous associons pour chaque PDD un environnement interne propre qui contient des paramètres, des listes et des variables. Les tâches d'un PDD manipulent cet environnement lors de son exécution. Les paramètres permettent d'envoyer et de recevoir des listes et des variables, soit en utilisant des références aux autres PDD, soit en utilisant une communication externe au PDD. Les listes permettent de regrouper un ensemble d'éléments du système géré, c'est à dire des références d'instances des classes SDD. Enfin, les variables contiennent chacune un ensemble de valeurs réelles.

Comme le montre le tableau 3.3 et en se servant du tableau 3.2, nous méta-modélisons

4. arrêt bloquant

Caractère	Signification
,	concaténation
|	choix
=	définition
;	terminaison
( )	groupe
{ ... }	répétition de ...
[...]	optionnellement ...

TABLE 3.2 – Caractères utilisés pour la méta-modélisation EBNF

l'expression des paramètres, des références aux PDD et des appels de méthodes des SWD. Pour un premier exemple, nous détaillons la concaténation pour l'expression des paramètres : un numéro d'ordre, un espace, un nom de liste, un espace et optionnellement une répétition de noms de variables séparés par des espaces. Pour la suite et pour simplifier nous ne mentionnons plus les espaces entre les concaténations et nous mentionnons directement `liste` pour les noms de liste et `variable` pour les noms de variable. L'initialisation du PDD crée son environnement interne propre en y rajoutant toutes les listes et les variables de ces paramètres d'entrée et de sortie. Elles sont créées avec les noms de liste et les noms de variables des expressions des paramètres et contiennent respectivement les instances des classes SDD et les valeurs réelles fournies lors de l'appel du PDD (appel par référence ou par communication extérieure).

Les deux méta-modélisations suivantes du tableau 3.3 spécifient la syntaxe des références aux PDD. Cette référence peut s'effectuer de deux manières : soit depuis un autre PDD (expression de référence), soit par une communication extérieure grâce à l'envoi d'une notification (expression de notification envoyée par exemple par une sonde externe).

Pour illustrer notre propos, la figure 3.2 représente les deux catégories de référence aux PDD. Cet exemple conforme au méta-modèle syntaxique comprend un PDD *exemple* avec deux paramètres d'entrée :

```
expression du paramètre 1 = 1 servers loads memories

expression du paramètre 2 = 2 clients
```

Cet exemple montre aussi les deux références correspondantes :

```
expression de référence (appel d'un PDD depuis un autre PDD) =
 exemple(servers loads memories,clients).
```

Cette expression est utilisée dans une action qui référence le PDD exemple en lui passant des paramètres.

```
expression de notification (appel d'un PDD depuis l'extérieur) =
 serverOverload;server_2 server_4;98.4 86.3;896 752;client_1.
```

Elle est envoyée par une sonde externe qui génère la notification

Paramètres d'entrée et de sortie du PDD : associée au noeud graphique *Parameters*, attributs *order*, *liste* et *variables*.

```
expression d'un paramètre du PDD = numéro d'ordre, " ", nom de liste, "
", [ { nom de variable, " ", } ];
```

Référence à un autre PDD : associée au noeud graphique *PDD reference*, attribut *expression*.

```
expression de référence d'un PDD = [ sorties , "=" , ] nom du PDD
référencé , "(" [ , entrées ] , ")";
sorties = sortie , [ { "," , sortie } ];
sortie = liste , [ { " " , variable } ];
entrées = entrée , [ { "," , entrée } ];
entrée = liste , [ { " " , argument d'entrée } ];
argument d'entrée = variable | nom d'attribut des classes SDD | valeur;
```

Référence à un PDD par communication extérieure (notification) : non associée à un noeud graphique.

```
expression de notification = nom du PDD, [ { ";", noms des références de
classe SDD, ";", [ { liste de valeurs, ";", } ] } ];
nom de référence de classe SDD = { nom, " ", };
liste de valeurs = { valeur, " ", };
```

Appel de méthode du SWD : associé au noeud graphique *SWD method call*, attribut *expression*.

```
expression d'appel = liste , ".", nom de méthode du SWD;
```

TABLE 3.3 – Paramètres, références et appels de méthodes en EBNF

Lors de l'initialisation du PDD *exemple*, son environnement interne propre est peuplé de deux listes qui sont créées, respectivement servers et clients et de deux variables loads et memories. D'après l'expression de notification, la première liste contient les références 2 et 4 d'une classe SDD server. La deuxième liste contient la référence 1 d'une classe SDD client. La variable *loads* contient les valeurs réelles 98.4 et 86.3, alors que la variable *memories* contient les valeurs réelles 896 et 752.

L'appel de méthode du SWD correspond au déclenchement de la fonction de gestion nom de méthode du SWD définie dans le SWD des éléments du système référencés dans la liste *liste*. Il est à noter que ce déclenchement se fait de manière parallèle sur tous les éléments référencés.

Le tableau 3.4 méta-modélise les syntaxes des actions de modifications structurelles. Elles permettent de créer ou de détruire de nouveaux éléments dans le système et des liaisons inter-éléments lors de l'exécution de la politique de gestion. Pour la création de composants et la destruction de composants, le nombre d'instances à créer ou à détruire est soit donné directement en valeur entière, soit calculé par une formule dont la syntaxe est similaire à celle de la condition du noeud de contrôle transition sur valeur. Les créations

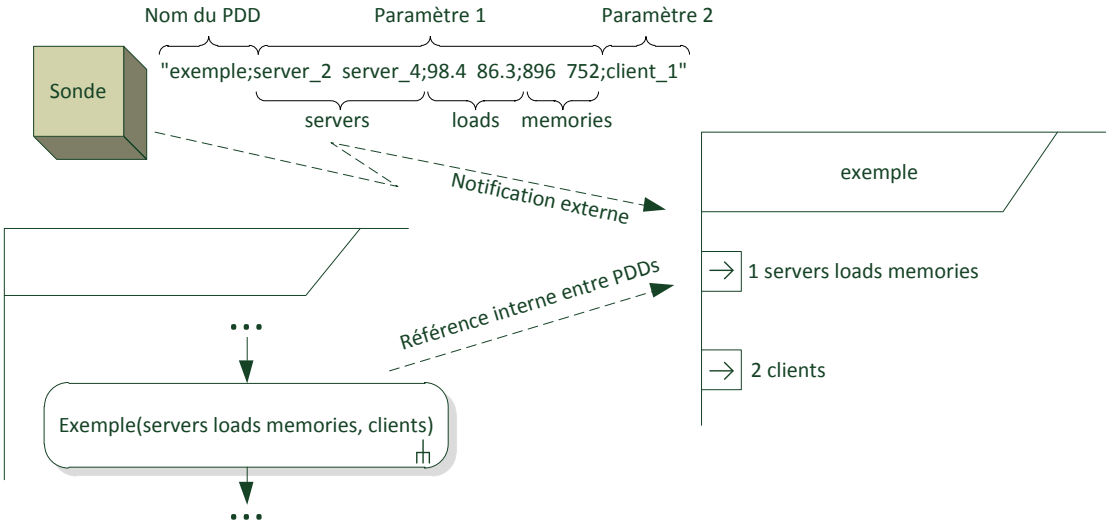


FIGURE 3.2 – Représentation des deux catégories de référence aux PDDs

s'effectuent de manière parallèle (réservation de tous les matériels nécessaires suivant le HDD et déploiement du nombre d'éléments de la classe SDD concernée). De même, la destruction de composants s'effectue de manière parallèle sur les références de la liste et au maximum sur un nombre limité de références. Ce nombre est donné par *nombre d'instances à supprimer*. La création de liens inter-composants s'effectue quand à elle un à un suivant les composants des deux listes liste 1 et liste 2. Si les listes ne sont pas de la même taille, elle s'effectue avec un équilibrage du nombre de liens entre les composants des deux listes. En effet, les composants ayant le moins de liens sont choisis en priorité pour la création des liens. Il en va de même pour la destruction de liens inter-composants qui choisit d'abord les composants qui ont le plus de liens si les listes ne sont pas de la même taille.

Le tableau 3.5 méta-modélise les actions qui permettent de créer et de détruire des attributs pour les composants ou de modifier leur valeurs. Ces attributs sont ensuite utilisables pour le filtrage de listes qui effectue une sélection des composants satisfaisant une opération de filtrage sur les attributs. Les attributs créés lors d'une exécution de PDD restent ensuite utilisables comme tous les attributs des classes SDD pour la configuration des éléments du système y compris une fois que le PDD a terminé son exécution. La création et la modification d'un attribut ont une syntaxe identique. En effet, si l'attribut désigné par l'expression d'accès à l'attribut n'existe pas alors il est créé. Notons qu'un attribut spécial nommé *length* permet d'accéder à la taille d'une liste. Cet attribut n'est pas modifiable mais nous le faisons apparaître ici car l'expression d'accès à l'attribut est utilisée plus tard notamment pour les opération de filtrage de listes. La modification d'un attribut affecte une nouvelle valeur qui peut être donnée directement sous forme d'une répétition de type double ou depuis une variable qui contient les valeurs réelles. Le nombre de valeurs contenues dans la variable doit alors être égal au nombre de références dans la

Création de composants : associée au noeud graphique *Component creation*, attribut *expression*.

```
expression de création de composants = liste, "=", nom de classe SDD,
"++", [ "[" , nombre d'instances à créer , "]" ] ;
```

Destruction de composants : associée au *Component destruction*, attribut *expression*.

```
expression de destruction de composants = liste, "-", [ "[" , nombre
d'instances à supprimer , "]" ] ;
```

Création de liens inter-composants : associée au noeud graphique *Link creation*, attribut *expression*.

```
expression de création de liens = "bind", liste 1, " ", liste 2 ;
```

Destruction de liens inter-composants : associée au noeud graphique *Link destruction*, attribut *expression*.

```
expression de destruction de liens = "unbind", liste 1, " ", liste 2 ;
```

TABLE 3.4 – Modifications structurelles en EBNF

liste. La sélection de composants par filtrage crée une nouvelle liste dans l'environnement interne du PDD qui contient des références ayant passé l'opération de filtrage. Cette opération peut être un minimum *min(...)* ou un maximum *max(...)* sur les valeurs des attributs, dans ce cas si plusieurs références ont le même minimum ou maximum elles seront contenues dans la liste créée. L'opération peut aussi faire intervenir les opérateurs de comparaison standard ($\leq$, $\geq$, $=$,...) pour filtrer certains composants suivant des bornes sur les valeurs des attributs. De plus, ces opérations peuvent être des opérations de comparaison standard sur des valeurs directes ou le contenu de variables. Enfin, une liste filtrée peut être créée à l'aide d'une liste existante et d'un nombre d'instances à récupérer entre crochets. Ce nombre d'instances peut être donné de trois manières : directement en valeur entière, à l'aide d'une variable ou à l'aide d'un calcul dont l'expression est similaire à celle de la condition du noeud de contrôle transition sur valeur.

Enfin, le tableau 3.6 méta-modélise les syntaxes utilisées dans certains noeuds de contrôle. En général, les codes de retour sont des chaines de caractère "[OK]" si le PDD n'a rencontré aucune erreur, "[NOK]" dans le cas contraire. L'utilisateur peut spécifier un code de retour personnalisé, par exemple pour déterminer quel chemin a été parcouru dans le PDD. Les transitions sur valeur sont un élément clé des PDD car ils permettent de spécifier plusieurs comportements possibles dans une même politique de gestion. Une telle transition n'est franchie que si la valeur spécifiée correspond à la valeur retournée par la tâche précédente ou si la condition spécifiée est vérifiée. Il ne peut y avoir qu'une valeur ou qu'une condition spécifiée par transition. Une condition est définie comme une opération standard fournissant un valeur réelle suivie de la même opération de filtrage utilisée précédemment dans le tableau 3.5. Nous rajoutons aux opérations courantes : les fonctions calculant la somme *sum* et la moyenne *avg* des attributs d'une liste. La transition sinon quand a elle a une expression associée "[else]" qui est constante et non modifiable. Enfin, l'expression d'un timer est définie comme un nom, un espace et une

Création et modification d'un attribut : associée aux noeuds graphiques *Attribute value modification* ou *Attribute creation*, attribut *expression*.

```
expression de modification ou de création d'un attribut = expression
d'accès à l'attribut, "<-", valeur;
expression d'accès à l'attribut = chemin, ".", nom d'attribut | liste,
".length";
chemin = liste , [ {".", lien } ];
lien = nom de classe SDD | nom de lien inter-classe SDD;
valeur =  type Double, " "  | variable;
```

Destruction d'un attribut : associée aux noeuds graphiques *Attribute destruction*, attribut *expression*.

```
expression de destruction d'un attribut = expression d'accès à
l'attribut, "<-";
```

Création de listes de composants : associée aux noeuds graphiques *List creation*, attribut *expression*.

```
expression de création de liste = nom de liste, "=", nom de classe SDD |
chemin;
```

Sélection de composants par filtrage : associée aux noeuds graphiques *Component selection*, attribut *expression*.

```
expression de sélection par filtrage = nom de liste filtrée, "=", (
expression d'accès à l'attribut, opération de filtrage) | ( "min(" |
"max(", expression d'accès à l'attribut, ")" ) | ( liste, "[", nombre
d'instances, "]" ) ;
opération de filtrage = "=" | "<" | "<=" | ">" | ">=", valeur | variable;
```

TABLE 3.5 – Modifications et sélections de composants en EBNF

valeur qui détermine le temps de pause en secondes.

Noeuds de contrôle

Code de retour d'un PDD : associée au noeud graphique *Return code*, attribut *code*.
```
code de retour = "[OK]" | "[NOK]" | ( "[", chaîne personnalisée, "]" ) ;
```

Transitions sur valeur : associée au noeud graphique *Value edge*, attribut *value* (valeur) ou attribut *condition* (condition).
```
valeur = code de retour | variable | valeur;

condition = opération standard | variable | valeur , opération de
filtrage;
opération standard = opérande, "+" | "-" | "*" | "/", opérande;
opérande = opération standard | ( "sum(" | "avg(" , expression d'accès à
l'attribut, ")" ) | variable | valeur | expression d'accès à l'attribut;
```

Transitions sinon : associée au noeud graphique *Else edge*, attribut *constant*.
```
constant = "[else]";
```

Expression d'un timer : associée au noeud graphique *Timer*, attributs *name* et *time value*.
```
expression d'un timer = nom du timer, " ", valeur;
```

TABLE 3.6 – Noeuds de contrôle en EBNF

3.3 Validation

3.3.1 Description des expériences

Nous proposons pour la validation de notre méta-modélisation une expérience de "self-healing", consistant à réagir face à une panne d'un cluster, pouvant impacter une partie d'une architecture logicielle déployée. L'expérience se base sur l'application distribuée DIET, déployée sur la plateforme expérimentale Grid'5000. L'expérience met en évidence un mécanisme de réparation automatique d'une architecture logicielle mis en oeuvre lors d'un crash de cluster à l'aide des politiques définies par des PDD. Nous comparons deux cas : dans le premier cas, chaque élément déployé sur le cluster est observé par une sonde et dans le deuxième cas on utilise une sonde pour observer plusieurs éléments. Ces sondes génèrent des expressions de notification pour faire un appel aux PDD correspondants aux réparations (voir tableau 3.3). Dans le premier cas plusieurs notifications seront reçues par TUNe et donc plusieurs PDD seront exécutés. Dans le deuxième cas la sonde agrège plusieurs événements pour en faire une seule notification.

3.3.1.1 Cadre matériel de l'expérience

Pour cette expérience, nous utilisons la plateforme Grid'5000 qui est modélisée avec le HDD de la figure 3.3. Nous retrouvons les attributs usuels de TUNe : *user*, *javahome*, *dirlocal*, *protocole* et *keypath* (voir section 2.5.4.1). Pour cette expérience, nous avons

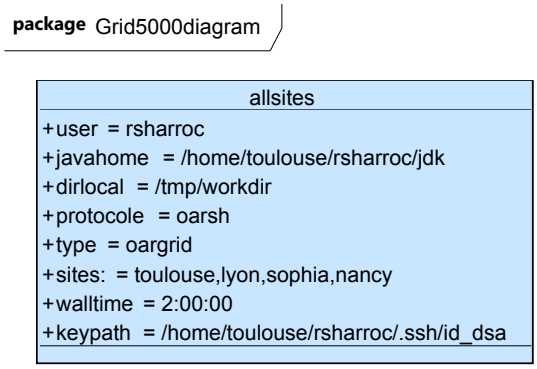


FIGURE 3.3 – HDD utilisé pour les expériences

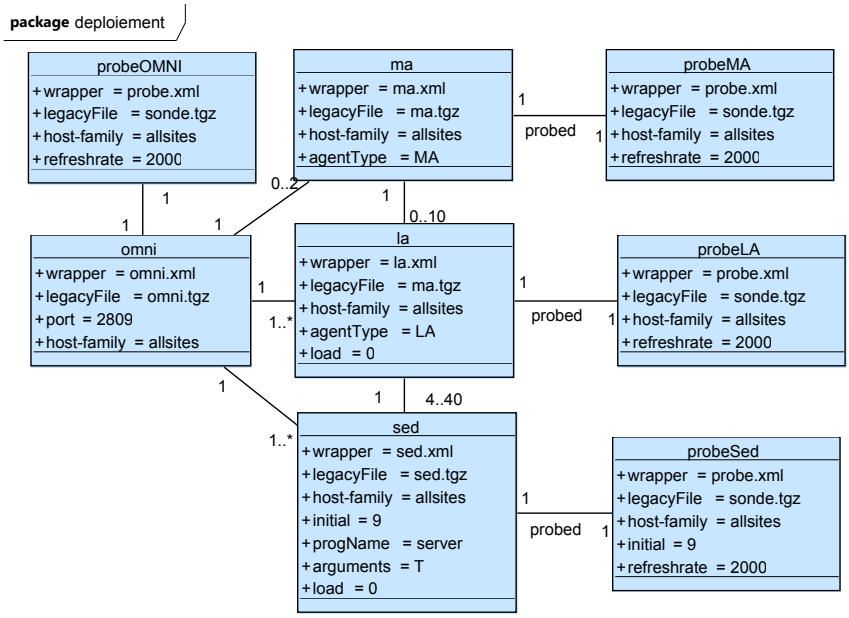


FIGURE 3.4 – SDD utilisé pour le déploiement de DIET, 1 probe par SED

implémenté un **plugin OARTUNe** personnalisé pour le *protocole oarsh* qui permet d'accéder aux fonctionnalités de réservation de ressources matérielles sur Grid'5000.

Ici nous introduisons trois attributs personnalisés utilisés par le plugin OARTUNe. L'attribut *type* spécifie à quel niveau la réservation doit se faire, ici au niveau de la grille entière (*oargrid*). L'attribut *sites* définit quels sites sont utilisés pour la réservation au niveau grille (argument d'entrée de la commande *oargrid*) et l'attribut *walltime* la durée de réservation (ici 2 heures). En réalité la commande oargrid prend aussi comme argument le nombre de noeuds à réserver pour chaque site. Ici, le nombre total de noeuds nécessaires est calculé lors du déploiement initial du SDD et est divisé équitablement sur tous les sites.

3.3.1.2 Cadre logiciel de l'expérience

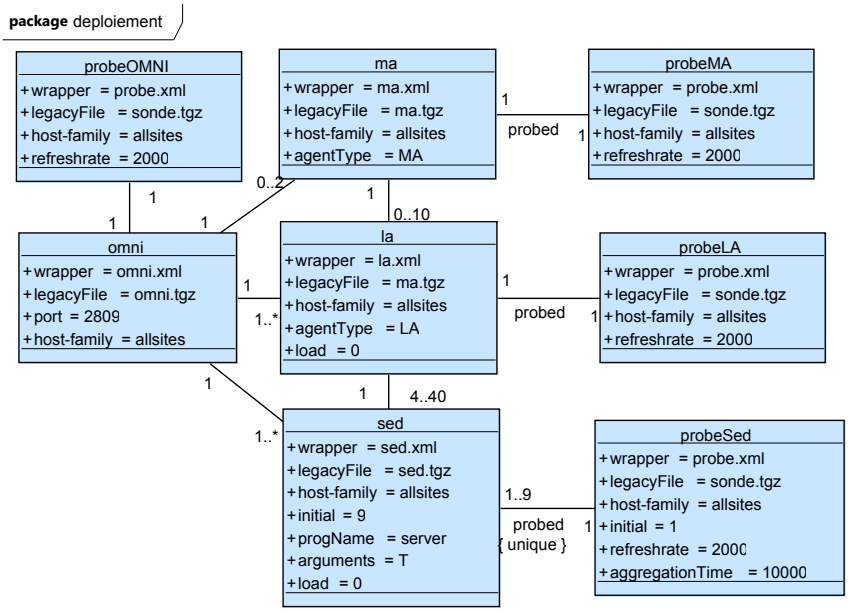


FIGURE 3.5 – SDD utilisé pour le déploiement de DIET, 1 probe d'agrégation pour les SEDs

3.3.1.2.1 Expérience sans agrégation des événements Pour les expériences de validation, nous utilisons des sondes génériques mises à disposition par TUNe par défaut qui peuvent observer l'existence des PID des logiciels déployés. La première expérience utilise le SDD de la figure 3.4 : chaque SED déployé (initialement 9 SEDs sont déployés) est observé par une sonde. Il y a donc autant de sondes que de SEDs et chaque instance de classe SDD utilisant une machine différente, cela a pour conséquence la réservation d'une machine dédiée par sonde sur la plateforme Grid'5000 pour cette expérience. L'attribut *refreshrate* est spécifique à la sonde et permet de spécifier la période de vérification de l'existence du PID (ici 2 secondes). Le fonctionnement basique de la sonde générique est le suivant : elle essaye de se connecter au noeud observé et récupère le PID de l'élément observé dans la liste des processus actifs du noeud. Si le noeud ne répond pas ou si le PID n'existe plus, la sonde génère une notification qu'elle envoie à TUNe pour démarrer l'exécution du PDD correspondant à la réparation de l'élément concerné.

3.3.1.2.2 Expérience avec agrégation des événements Dans cette deuxième expérience, la sonde utilisée pour observer les SEDs est capable d'en gérer plusieurs en même temps. Un paramètre supplémentaire *aggregationTime* est introduit pour permettre l'agrégation d'événements (voir figure 3.5). En effet, lorsque qu'un cluster tombe en panne, tous les éléments du cluster sont potentiellement impactés, ce qui génère plusieurs événements. Dès que la sonde détecte une défaillance, au lieu d'envoyer directement une notification à TUNe, elle attend pendant un certain temps de détecter d'autres événements. Ce laps de temps est défini par le paramètre *aggregationTime* en millisecondes. Cela permet d'agréger les événements et d'envoyer une liste des éléments défaillants à TUNe. En contrepartie, la sonde est moins réactive que dans l'expérience précédente car elle n'envoie la notification qu'après avoir attendu un certain temps.

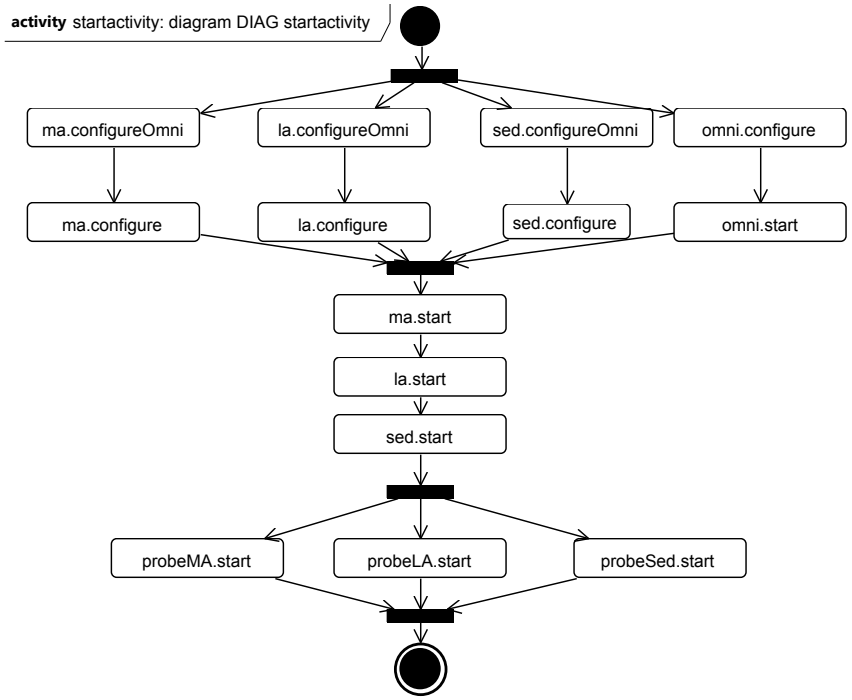


FIGURE 3.6 – PDD utilisé pour le démarrage de l'architecture DIET

3.3.2 PDD utilisés pour les expériences

3.3.2.1 PDD utilisés pour la phase de démarrage

Pour la **phase de démarrage** nous introduisons un PDD nommé *startactivity* qui définit une politique de démarrage automatiquement exécutée après la **phase de déploiement**. La figure 3.6 montre le PDD créé pour le démarrage de l'application distribuée DIET. Il est constitué de deux noeuds de type *fork* créant des threads pour la parallélisation du démarrage et deux noeuds de type *join* pour la synchronisation des threads créés. Tous les autres noeuds sont le noeud initial d'où démarre l'exécution, le noeud final et des noeuds de type *SWD method call*. Nous voyons que le démarrage de l'architecture DIET consiste à configurer (par l'appel de méthode *omni.configure*) et démarrer (par l'appel de méthode *omni.start*) un serveur de nommage omniNames et à configurer les autres éléments du système pour qu'ils puissent utiliser ce serveur. Les appels de méthode *ma.configureOmni*, *la.configuraOmni* et *sed.configureOmni* créent des variables d'environnement et éventuellement des fichiers de configuration spécifiques pour se connecter au serveur omniNames. Les méthodes suivantes *ma.configure*, *la.configure* et *sed.configure* créent des fichiers de configuration utilisés par les agents DIET et qui comprennent des paramètres spécifiés par certains attributs dans le SDD (par exemple l'attribut *agentType*). Le démarrage des agents **MA** (*ma.start*), **LA** (*la.start*) et des **SED** (*sed.start*) se fait une fois toutes les configurations terminées et le serveur de nommage démarré, à l'aide d'une synchronisation par un noeud de type *join*. Nous rappelons que l'appel de méthode d'un SWD se fait aussi de manière parallèle sur tous les éléments de la liste concernée. Par exemple *sed.start* démarre les neuf SEDs initialement déployés en parallèle. Notons aussi que *sed* représente un nom de classe SDD, ce qui équivaut à une

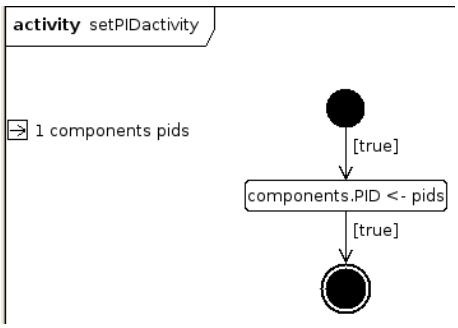


FIGURE 3.7 – PDD utilisé pour la récupération des PIDs

liste de toutes les instances de cette classe. Une fois les agents DIET démarrés, les sondes associées démarrent en parallèle après le noeud fork (*probeMA.start*, *probeLA.start* et *probeSed.start*).

Lors du démarrage de chaque élément, les appels de méthode *start* récupèrent le PID des processus lancés sur les noeuds distants. Cette méthode est fournie par défaut dans un squelette SWD réutilisable pour créer des SWD personnalisés. Les PIDs sont stockés dans un attribut *PID* à l'aide du PDD *setPIDactivity* (figure 3.7). Par défaut, la méthode *start* envoie une notification pour exécuter ce PDD. L'expression de notification (voir tableau 3.3) contient donc le nom du PDD *setPIDactivity*, les références des éléments démarrés et les valeurs numériques des PIDs. Au démarrage du PDD, le paramètre d'entrée spécifie que les références sont stockées dans la liste *components* et les PIDs dans la variable *pids*. Ce PDD consiste donc en une seul tâche de création ou modification d'attribut $components.PID <- pids$. Cette action affecte ou crée (si l'attribut n'existe pas) pour chaque élément concerné l'attribut PID avec la valeur du PID récupérée par la méthode de démarrage. Cet attribut est ensuite utilisé par la méthode du SWD des sondes pour passer en argument le ou les PIDs qu'elle doit observer.

3.3.2.2 PDD utilisés pour les reconfigurations durant la phase de gestion

Pour notre expérience de "self-healing", nous voulons montrer qu'il est possible de définir des politiques de réparation qui permettent de remettre le système dans un état opérationnel lors d'un crash de cluster. Nous introduisons un ensemble de PDD utilisés pendant la phase de gestion de TUNe pour satisfaire les propriétés de "self-healing". En général, pour une architecture de calcul DIET, un ensemble de SEDs est réparti sur un cluster et un LA supervise ces SEDs. Nous décidons de déclencher la panne du cluster qui contient tous les SEDs ainsi que le LA les supervisant. Nous considérons que le MA étant de plus haut niveau, il est situé sur un autre cluster. De même pour le serveur de nom omniNames et les sondes qui sont réparties sur d'autres clusters. Prendre en compte le "self-healing" pour ces derniers revient à créer des PDD relativement proches de ceux fournis pour les SED et le LA.

3.3.2.2.1 PDD pour le "self-healing" des LA La figure 3.8 montre le PDD utilisé pour la réparation d'un LA. Ce PDD prend deux paramètres d'entrée : *brokenla* qui contient la liste des LAs en panne et *probe* qui contient la sonde ayant envoyé la notifica-

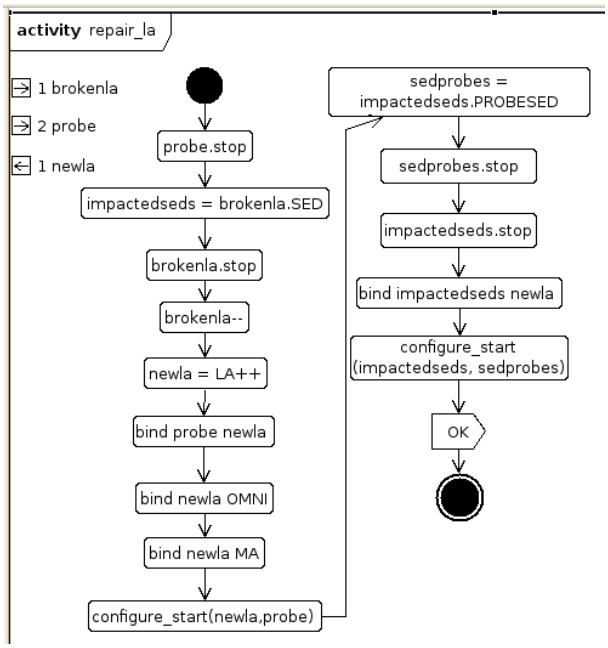


FIGURE 3.8 – PDD utilisé pour la réparation d'un LA

tion. Le paramètre de sortie *newla* retourne le LA nouvellement créé lors de l'exécution du PDD. Cette politique de réparation décrit un flot de travail séquentiel d'actions. La première action *probe.stop* (de type appel de méthode SWD) arrête la sonde ayant envoyé la notification afin qu'elle stoppe l'envoi de nouvelles notifications. Une liste *impactedseds* est ensuite créée à l'aide d'une action de type *List creation*. Elle contient tous les SEDs connectés au LA en panne. La politique essaye ensuite d'arrêter le LA en panne (*brokenla.stop*) et de détruire son composant (*brokenla–*), c'est à dire d'essayer de supprimer tous les fichiers générés et binaires déployés le concernant. Dans le cas d'une panne de cluster, ces deux actions échouent et le flot de travail continue de s'exécuter. Elles peuvent néanmoins être utilisées dans le cas de détection d'un comportement anormal du LA (par des sondes spécialisées). Un nouveau LA est déployé sur une nouvelle machine réservée à l'aide d'une action de création de composants (*newla = LA++*). Ce nouveau LA est ensuite relié à la sonde précédemment utilisée pour le LA en panne (*bind probe newla*) ansi qu'au serveur de nom omniNames (*bind newla OMNI*) et au MA (*bind newla MA*). Une fois tous les liens créés, le nouveau LA et la sonde sont configurés et démarrés à l'aide d'un PDD générique *configure_start* qui fait appel aux méthodes *configure* et *start* de ses éléments passés en paramètre d'entrée. Ce PDD est réutilisé à la fin du diagramme. La partie droite du diagramme s'occupe d'arrêter les SEDs connectés au LA en panne pour les reconfigurer avec le LA nouvellement créé et les redémarrer. Pour cela, les sondes de ces SEDs sont d'abord arrêtés (*sedprobes.stop*) pour ne pas qu'elles envoient de notification lors de l'arrêt de ceux-ci (*impactedseds.stop*). Un lien est ensuite créé avec le nouveau LA (*bind impactedseds newla*). Notons ici que l'ancien lien avec le LA en panne était déjà détruit par la destruction du LA en panne au début de l'exécution du PDD. Enfin, le PDD *configure_start* est utilisé pour configurer et démarrer les SEDs et leurs sondes. Nous voyons ici que pour raison de simplification, les cas d'erreurs ne sont pas

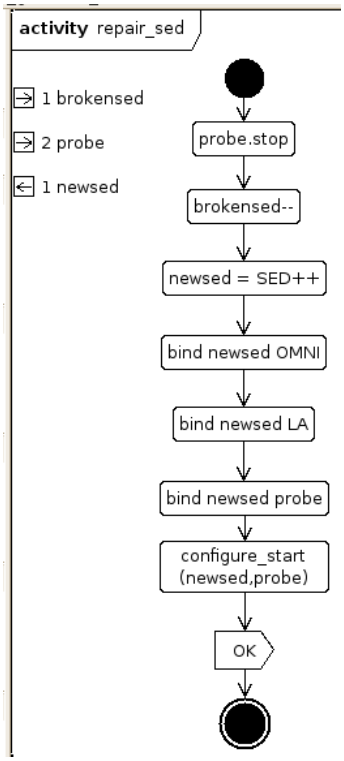


FIGURE 3.9 – PDD utilisé pour la réparation des SEDs

traités dans cette politique et le PDD utilise un noeud de type *Return code* pour renvoyer le code de retour *OK* dans tous les cas.

3.3.2.2.2 PDD pour le "self-healing" des SEDs La figure 3.9 montre le PDD utilisé pour la réparation d'un SED. De même que pour la réparation du LA, ce PDD prend deux paramètres d'entrée : *brokensed* contient la liste des SEDs en panne et *probe* contient la sonde ayant envoyé la notification. Le paramètre de sortie *newsed* retourne le SED nouvellement créé lors de l'exécution du PDD. Il est possible de créer plusieurs SEDs en remplaçant l'action `newsed = SED++` par `newseds = SED++[brokenseds.length]` Notons ici que pour la première expérience, chaque SED est relié à sa propre sonde. Chaque sonde envoie donc une notification `"repair_sed;sed_x;probeSed_x"`. Il y a donc autant d'exécution de PDD (`repair_sed`) que de SEDs en panne et la liste *brokensed* ne contient qu'une référence vers le SED en panne (`sed_x`). En revanche pour la deuxième expérience, cette liste contient plusieurs références de SEDs grâce à l'agrégation des événements par la sonde observant plusieurs SEDs. Le flot de travail de ce PDD suit la même idée que pour la réparation d'un LA. La sonde est arrêtée et le(s) SED(s) détruit(s). Autant de nouveaux SEDs sont créés, reliés au serveur de nom omniNames, au LA et à la sonde. De même que pour la réparation du LA, ils sont ensuite configurés et démarrés.

3.3.2.3 Temps de réparation du cluster

La figure 3.10 représente le temps passé pour chaque action lors de l'exécution du PDD *repair_la*. Le temps de réparation total d'exécution du PDD pour la réparation

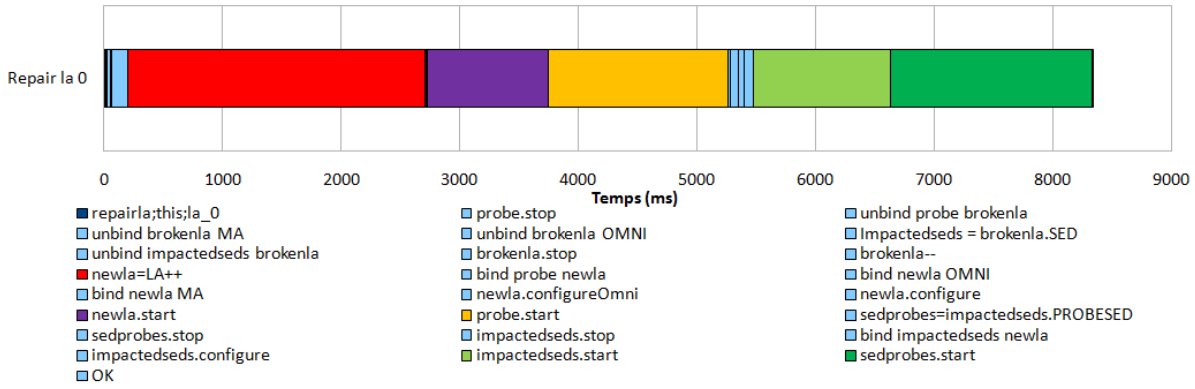


FIGURE 3.10 – Détail du temps de réparation du LA

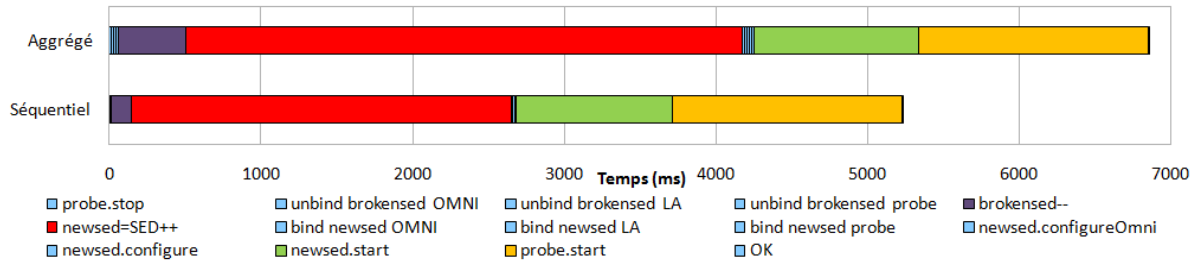


FIGURE 3.11 – Comparaison du temps d'exécution du PDD de réparation des SEDs

des LAs est d'environ 8,2 secondes. Nous précisons que nous ne prenons pas en compte le temps de réservation des nouvelles machines sur Grid'5000 avec le plugin OARTUNe. En effet, ce temps étant trop variable (pouvant varier de quelques secondes à plusieurs dizaines de secondes) nous ne le prenons pas en compte. Nous remarquons que les temps les plus importants sont le temps de création du nouveau LA (2507 ms) qui prend en compte le déploiement des binaires sur une nouvelle machine et les temps de démarrage : du nouveau LA (1016 ms), de la sonde du LA (1517 ms), des SEDs impactés (1152 ms) et des sondes des SEDs impactés (1704 ms). Notons que le démarrage de la sonde du LA prend plus de temps que le démarrage du LA lui-même car elle doit se connecter à distance à la machine du LA en utilisant le protocole ssh, relativement lent par rapport à la mise en place des communications du LA par le protocole CORBA. Les autres actions ne dépassent pas 137 ms (action *brokenla- -*).

La figure 3.11 représente le temps passé pour les actions du PDD de réparation des SEDs pour les deux expériences. Dans le cas *séquentiel*, le PDD est exécuté pour une instance de SED. Au contraire, dans le cas agrégé, le PDD est exécuté sur la liste des SEDs renvoyée par la sonde. Nous remarquons donc que le temps de suppression des SEDs est supérieur dans le cas agrégé (448 ms) que dans le cas séquentiel (133 ms). Aussi, le temps de création des nouveaux SEDs est de 3666 ms dans le cas agrégé (9 SEDs créés en parallèle) et de 2506 ms dans le cas séquentiel (1 SED créé). Les temps de démarrage restent sensiblement les mêmes (1086 ms en agrégé et 1036 ms en séquentiel pour le démarrage des nouveaux SEDs). Le temps de démarrage de la sonde reste exactement le

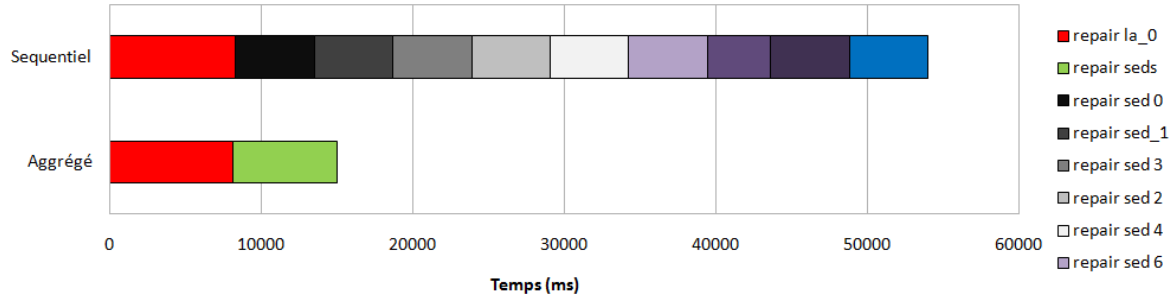


FIGURE 3.12 – Comparaison du temps de réparation du cluster

même (1514 ms) dans les deux cas car il s'agit toujours d'une seule sonde à démarrer. Le temps total de réparation dans le cas agrégé pour 9 SEDs est de 6854 ms et dans le cas séquentiel pour 1 SED de 5233 ms.

La figure 3.12 montre finalement le temps qu'il faut pour la réparation du cluster dans le cas agrégé et dans le cas séquentiel. Dans le cas séquentiel, toutes les notifications reçues par TUNe sont stockées dans une file FIFO. Pour des raisons de maintien de la cohérence de la représentation interne de la couche SR dans TUNe, les notifications sont traitées séquentiellement. Nous le voyons sur la figure avec l'exécution séquentielle des PDD de réparation du LA puis de tous SEDs. Le temps moyen de réparation pour les SEDs est de 5068 ms dans ce cas et le temps total de réparation du cluster est de 53940 ms. Nous notons un gain de performance très important dans le cas agrégé car le temps de réparation total du cluster est alors de 15012 ms. En effet, le PDD de réparation des SEDs traitant une liste de tous les SEDs en panne permet de les réparer en parallèle au sein du même PDD.

3.4 Conclusion

Nous avons introduit particulièrement le méta-modèle des PDD qui permettent de décrire des politiques de gestion autonomiques, en utilisant le formalisme du MOF ou "Meta-Object Facility" pour la partie graphique et la forme étendue de Backus-Naur pour la partie syntaxique. Nous avons ensuite donné des exemples de PDD conformes au méta-modèle et nous avons validé notre approche à l'aide d'une expérience de réparation autonomique lors de crash de clusters entraînant l'arrêt de certains éléments du système DIET.

Nous venons de montrer la faisabilité de description de politiques de gestion de haut niveau permettant de satisfaire la propriété de "self-healing" appliquée à un système propriétaire, tout en garantissant un niveau de généricité. En effet, le système de démonstration DIET n'a subi aucune modification pendant nos expériences. Les pannes étant très fréquentes dans un environnement distribué composé de machines et d'entités logicielles, nous avons introduit un mécanisme qui permet d'observer le système déployé, de détecter la panne d'un morceau du système, de décrire des politiques de "self-healing" pour les différentes réparations et de finalement ramener le système dans un état opérationnel sans aucune intervention humaine. Nous avons aussi mis en évidence que le

temps de réparation pour plusieurs éléments tombant en panne simultanément peut être réduit en utilisant des mécanismes d'agrégation des événements au niveau des sondes. La contrepartie vient d'une très légère perte de réactivité car la sonde doit attendre d'agréger suffisamment d'événements avant d'envoyer une notification de réparation à TUNe.

Chapitre 4

Auto-configuration - "Self-configuring"

Sommaire

4.1 Introduction et problématique 71
4.1.1 Rappels pour l'auto-configuration 71
4.1.2 Problématique de "self-configuring" des réseaux de capteurs . . 72
4.2 Contributions pour la propriété de "self-configuring" 74
4.2.1 Rajout de la propriété de "self-configuring" pour l'intergiciel XSStreaMWare 74
4.2.2 Validation et exemples de politiques de gestion de réseaux de capteurs 79
4.3 Conclusion pour l'étude de cas de "self-configuring" appliqué aux réseaux de capteurs 81
4.4 Problématique de "Self-configuring" pour une simulation électromagnétique sur grille 83
4.4.1 Simulation électromagnétique et grille de calcul 84
4.4.2 Validation de l'approche de "self-configuring" pour YatPac sur Grid'5000 89
4.4.3 Conclusion pour l'étude de cas de "self-configuring" appliqué aux simulations électromagnétiques sur grille de calcul 93
4.5 Conclusion et perspectives 94

4.1 Introduction et problématique

4.1.1 Rappels pour l'auto-configuration

La propriété d'auto-configuration ou "self-configuring" permet à un système de s'installer, de se paramétrer et de se mettre à jour tout seul. On peut parler de la **gestion du déploiement** et de la **gestion des mises à jour et des paramétrages** : ces fonctions sont subdivisées en plusieurs tâches à savoir : l'allocation des ressources matérielles, l'installation, la configuration, le démarrage, la désinstallation et l'arrêt des éléments d'un

système. Du point de vue logiciel, le déploiement doit pouvoir s'adapter à différents modèles d'applications (parallèle, paramétrique, répartie, flot de travail). Du point de vue matériel, le déploiement s'effectue plutôt de manière physique (déploiement sur le terrain des matériels utilisés) et le "self-configuring" consiste à paramétrer les ressources matérielles (paramétrage de bas niveau) ou à mettre à jour les micrologiciels ("firmwares"), c'est-à-dire le logiciel spécifique à la gestion du matériel.

Dans ce chapitre, nous voulons étudier la faisabilité de rajout de la propriété de "self-configuring" aux deux niveaux : logiciel et matériel. Pour cela nous menons deux cas d'étude : l'un portant sur le paramétrage de réseaux de capteurs (niveau matériel) et l'autre portant sur le déploiement d'un logiciel de simulation électromagnétique sur une grille de calcul (niveau logiciel). Le premier cas d'étude a été mené en collaboration avec le laboratoire de recherche National Institute of Informatics au Japon, lié aux universités de Tokyo et de Waseda. Le deuxième cas d'étude porte sur une collaboration avec les ingénieurs électroniciens du groupe MINC du LAAS-CNRS à Toulouse.

4.1.2 Problématique de "self-configuring" des réseaux de capteurs

4.1.2.1 Problématique générale

Suivant les avances technologiques, les petits instruments de capture (capteurs de température, chimiques, GPS, lecteurs RFID [1]) sont de plus en plus présents dans des applications très diverses. Les domaines touchés sont par exemple l'industrie, la médecine ou la domotique (contrôles automatiques domestiques). Ces applications couplent donc une partie micrologicielle (logiciel de contrôle des capteurs) et une partie matérielle (les capteurs). Elles doivent aussi inter-opérer entre-elles, avec la nécessité de configurer des systèmes hétérogènes et propriétaires (tant du point de vue matériel que logiciel) avec un très grand nombre de capteurs : on parle alors de **réseaux de capteurs**. Face à cette complexité grandissante, un besoin de mécanismes de contrôle à distance se fait de plus en plus ressentir. Par exemple, les possibilités matérielles offertes par les capteurs de dernière génération permettent de les configurer dynamiquement en déployant à distance ces micrologiciels. Les dernières générations de capteurs peuvent aussi embarquer un système d'exploitation minimaliste qui peut supporter l'exécution de modules applicatifs. Ces modules applicatifs peuvent également être déployés à distance et mis à jour durant tout le cycle de vie des capteurs. Le nombre de capteurs étant en constante augmentation, les administrateurs humains utilisant les logiciels de contrôle conventionnels manuels se retrouvent vite dépassés par les tâches d'administration. De plus, la nature temps-réel de certains réseaux de capteurs impose d'intervenir sur les capteurs avec des actions d'administration en temps contraint.

L'approche que nous proposons est d'utiliser les concepts de l'autonomic computing pour la gestion des réseaux de capteurs. Le but est de fournir un outil avec des capacités de gestion autonomique qui soit indépendant du domaine fonctionnel (gestion logicielle, gestion de la configuration ou de la performance), du domaine architectural (niveau applicatif, système ou réseau) ou du type de matériel utilisé par ces réseaux de capteurs. Dans cet optique, nous adaptons les principes de gestion par des politiques de

1. Radio Frequency IDentification

haut niveau (intégration des Policy Description Diagrams) dans le système de contrôle de capteurs manuel existant nommé XSStreaMWare [32]. La section 2.3.5 présente l'intergiciel XSStraMWare existant. Le point commun entre notre approche et celle proposée par XSStreaMWare pour la gestion des réseaux de capteurs est la proposition d'une couche d'abstraction de haut niveau pour accéder aux fonctionnalités de bas niveau d'une manière générique. En effet, nous utilisons le mécanisme d'encapsulation avec les **SWD**. XSStreaMWare utilise quant à lui le mécanisme d'**adaptateurs** (*adapters*) qui transforme toutes les données brutes des capteurs et l'appel des fonctions d'administration en un format générique.

4.1.2.2 Problème de gestion manuelle des réseaux de capteurs avec XSStreaMWare

Les administrateurs font appel aux fonctions de gestion des capteurs à l'aide de ce modèle générique pour récupérer, modifier ou déclencher une opération de gestion de manière manuelle. Les trois opérations de gestion génériques utilisées par XSStreaMWare sont **GET**, **SET** et **ACT**. Pour chacune de ces opérations, dans la version originale de XSStreaMWare c'est un administrateur humain qui utilise l'interface graphique pour définir leur paramètres d'entrée.

Le problème vient du fait que toutes ces actions de gestion sont exécutées de manière manuelle dans le logiciel de contrôle. Pour cela, un administrateur humain doit récupérer les valeurs de paramètres des capteurs (**GET**), vérifier s'il y a une nécessité d'intervenir sur un certain nombre de capteurs en fonction de ces valeurs et éventuellement demander l'exécution d'actions de maintenance (**SET** et **ACT**). Ces actions ont parfois besoin de paramètres d'entrée que l'administrateur doit fournir de manière manuelle lors de l'exécution de celles-ci. Toutes ces tâches sont effectuées à partir d'une interface graphique centralisée et rattachée comme un service au plus haut niveau de la hiérarchie d'XSStreaMWare. En effet, l'administrateur humain a la possibilité de contrôler la hiérarchie complète du système, c'est-à-dire l'ensemble des capteurs regroupés par régions géographiques ou par types, à partir du site de contrôle.

Notre but est de simplifier cette administration manuelle en introduisant des règles qui permettent d'exécuter de manière automatisée les trois actions d'administration **GET**, **SET** et **ACT**. Nous définissons des règles ECA (Evénement Condition Action) qui prennent en compte le format générique des actions d'administration introduit par XSStreaMWare. Nous introduisons différents types d'événements qui peuvent intervenir dans le système. Quand un de ces événements survient, s'il est conforme à une condition associée, les actions d'administration correspondantes peuvent être exécutées. Pour représenter ces règles nous utilisons le concept de description par workflow introduit pour les PDDs. Nous spécifions et introduisons ensuite des **moteurs ECA** basés sur le moteur de déchiffrement XMI utilisé pour les PDDs. Ces **moteurs ECA** évaluent de manière continue et autonome les règles définies aux différents niveaux de l'architecture distribuée d'XSStreaMWare.

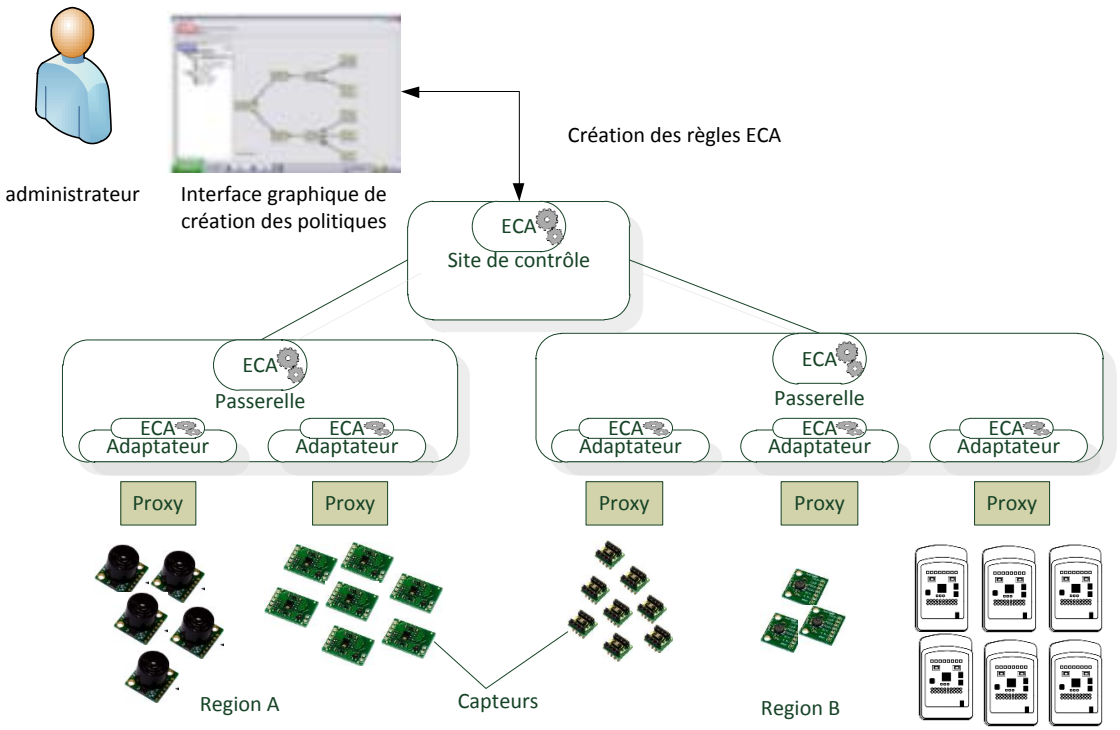


FIGURE 4.1 – Intégration de services ECA dans l'architecture hiérarchique de XSStreaMWare

4.2 Contributions pour la propriété de "self-configuring"

4.2.1 Rajout de la propriété de "self-configuring" pour l'intergiciel XSStreaMWare

4.2.1.1 Intégration du moteur de déchiffrage XMI en tant que service ECA

Notre approche consiste à introduire des **moteurs ECA** basés sur le moteur de déchiffrement XMI utilisé pour exécuter les PDD dans TUNe. Chaque moteur est un nouveau service qui s'intègre dans l'architecture existante de XSStreaMWare. Pour cela nous créons un service "plug & manage" ECA compatible et intégrable à tous les niveaux hiérarchiques d'XSStreaMWare. Comme montré sur la figure 4.1, ces services sont distribués sur trois couches de l'intergiciel : le site de contrôle, les passerelles et les adaptateurs. Chaque couche gère alors des événements à un niveau local : pour un lot de capteurs au niveau adaptateurs, pour une région au niveau passerelle et pour la hiérarchie globale au niveau du site de contrôle. En fonction des capacités techniques des capteurs il est parfois possible d'intégrer directement les services ECA au niveau des capteurs. Ces niveaux locaux de gestion sont appelés **sites autonomiques**, chacun de ces sites étant sous la responsabilité d'un **service ECA**. Ce partage de gestion permet de continuer la gestion d'un site autonomique même si une autre partie du système tombe en panne. De plus, les règles ECA sont évaluées au plus proche des capteurs dans la mesure du possible, ce

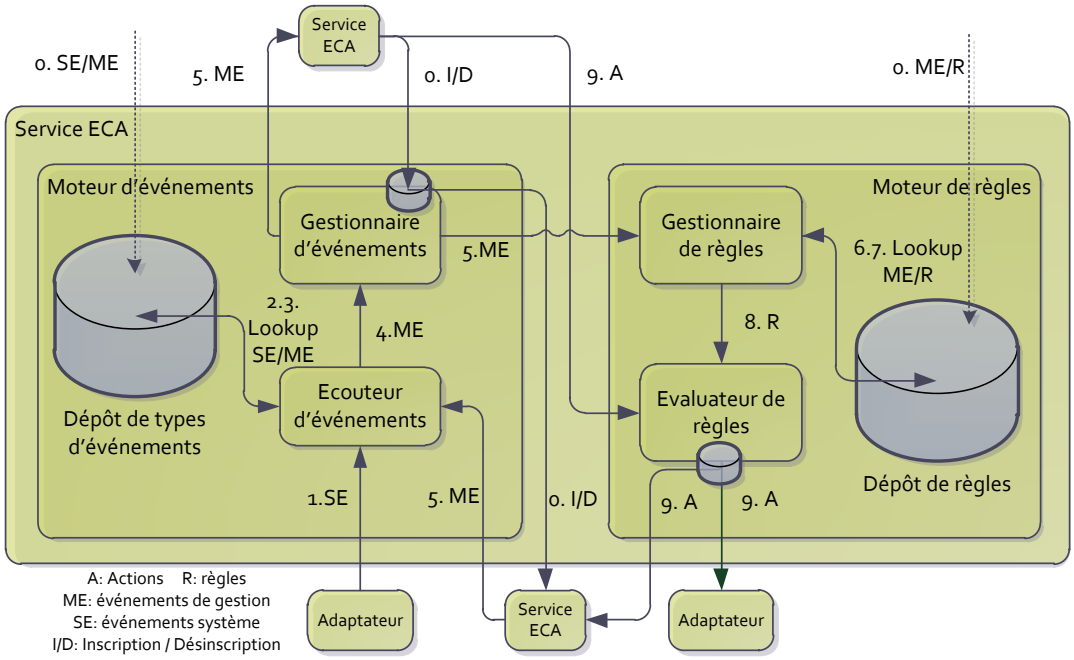


FIGURE 4.2 – Architecture d'un service ECA

qui permet de minimiser le trafic réseau dédié à la gestion et de réagir plus rapidement (plus le moteur ECA est proche du capteur plus la réaction sera rapide).

Au plus haut niveau de la hiérarchie, nous offrons au niveau du site de contrôle un service pour les administrateurs humains pour qu'ils puissent diffuser les règles ECA dans la hiérarchie. Une interface graphique leur permet de démarrer les différents services ECA et de créer les règles ECA. Il est également possible pour des applications externes de communiquer avec ce service (on parle alors de souscription au service) pour qu'ils reçoivent certains événements.

4.2.1.2 Architecture globale d'un service ECA

La figure 4.2 spécifie les mécanismes internes des services ECA que nous introduisons, ainsi que les différentes étapes d'un **cycle ECA**. Chaque service ECA est divisé en deux parties : le **moteur d'événements** *Event Engine* et le **moteur de règles** *Rule Engine*. Le **moteur d'événements** est responsable de la tâche d'observation ou "monitoring" et capture tous les événements qualifiés d'événements système (**SE**[2]). Le **moteur de règle** est quant à lui responsable d'analyser les événements et d'exécuter des actions sur les différents éléments du système.

4.2.1.3 Cycle ECA

Nous introduisons ici le processus global d'un cycle ECA qui commence par l'enregistrement des types d'événements (étape 0. SE/ME) et des règles (étape 0. ME/R) dans

2. System Events

Attributs de événements	Description
TYPE	*Hello, Bye, Periodic, Unavailable, Modified*
SOURCE	Donne la référence XSStreaMWare de l'élément qui a créé l'événement.
TARGET	Donne la référence XSStreaMWare de l'élément cible de l'événement.
TIMESTAMP	Temps estimé de détection de l'événement.
DATAMODEL	Contient des paramètres des capteurs organisés dans le format générique XSStreaMWare.

TABLE 4.1 – Attributs des événements de gestion

des dépôts, ainsi que les éventuelles inscriptions à ces types d'événements (étape 0. I/D). Le dépôt de types d'événements associe des événements systèmes avec des événements de gestion suivant un type. Les différents types sont stockés dans un dépôt appelé *Event Repository* et les règles dans un dépôt appelé *Rule Repository*. L'écouteur d'événements *Event Listener* est en charge du processus de détection (étapes 1 à 4). L'étape 1 capture les événements systèmes **SE** provenant des adaptateurs. L'étape 2 vérifie (action *lookup SE/ME*) que ces événements systèmes correspondent à un type prédéfini dans le dépôt *Event Repository*. Si tel est le cas, le dépôt renvoie (étape 3) un ou plusieurs événements que nous qualifions d'événements de gestion *Management Event* **ME**. Ces événements sont ensuite retransmis (étape 4) au gestionnaire d'événements *Event Manager* et sont diffusés (étape 5) au gestionnaire de règles et éventuellement aux différents services ECA de niveau supérieur qui ont souscrit (à l'étape 0. I/D) à ce type d'événements. Le gestionnaire d'événement vérifie (étape 6 *lookup ME/R*) dans le dépôt de règles *Rule Repository* si des règles sont associées à l'événement concerné. Si tel est le cas il renvoie les règles correspondantes (étape 7) qui sont retransmises à l'évaluateur de règles *Rule Evaluator*. Ce dernier évalue les conditions requises et exécute les actions définies dans les règles (étape 9) en les transférant soit à l'adaptateur soit au service ECA du niveau hiérarchique inférieur.

4.2.1.4 Types d'événements

Pour chaque événement, nous définissons plusieurs attributs représentés dans le tableau 4.1. Quand l'écouteur d'événement capture un événement système SE, la vérification des types s'effectue par interrogation du dépôt *Event Repository*. C'est dans ce dépôt que sont stockés ces différents attributs et qu'ils sont associés à des événements de gestion ME.

Le premier attribut est le type de l'événement (attribut *TYPE*). En effet, nous avons identifié et prédéfini plusieurs types d'événements pouvant se produire au niveau de l'intergiciel ainsi qu'au niveau des capteurs. Le premier type prédéfini est *HELLO*, c'est un événement qui se produit quand un nouveau capteur entre dans une région géographique

(ou qu'il est démarré) ou quand un nouveau service est inséré dans l'architecture XSStreaMWare. Le deuxième type est *BYE* et survient quand un capteur sort d'une région géographique (ou qu'il est éteint sans erreurs) ou qu'un service de l'architecture s'arrête correctement. Le type *UNAVAILABLE* est utilisé dans deux cas : si un capteur ou un service du système n'est pas joignable au bout d'un certain nombre de tentatives ou si un timeout survient lors de la communication (c'est à dire qu'il ne répond pas au bout d'un temps borné). Le type *PERIODIC* se réfère à un événement qui est généré de manière périodique (par exemple par un timer fixe). Finalement, le type *MODIFIED* est utilisé pour spécifier la modification d'une valeur d'un paramètre des capteurs ou des services XSStreaMWare. Ces différents types sont prédéfinis et peuvent être utilisés directement. L'utilisateur peut aussi définir des types spécifiques qui lui sont propres en utilisant une description générique d'événement. Cette description spécifie une partie des caractéristiques des événements, comme par exemple les propriétés ou les paramètres concernés par l'événement, des bornes sur leurs valeurs ou sur les dates de génération de l'événement. L'attribut *SOURCE* donne la référence XSStreaMWare de l'élément qui a généré l'événement et l'attribut *TARGET* de l'élément cible de l'événement. Par exemple si un capteur n'arrive pas à communiquer avec un autre capteur, il sera *SOURCE* d'un événement de type *UNAVAILABLE* et le capteur défaillant sera référencé dans l'attribut *TARGET*. L'attribut *TIMESTAMP* est le temps estimé de détection de l'événement. Ce temps est en effet estimé car dans l'exemple précédent, la détection du capteur défaillant peut prendre un certain temps. Enfin, pour l'attribut *DATAMODEL*, nous réutilisons le format générique introduit par XSStreaMWare pour stocker toutes les valeurs des paramètres issues du capteur, afin qu'ils puissent être accessibles en utilisant les mêmes méthodes génériques.

4.2.1.5 Modélisation de politiques pour la gestion des résaux de capteurs

Après avoir introduit les différents types d'événements prédéfinis, nous spécifions la définition des règles ECA qui permet la modélisation de politiques pour la gestion des réseaux de capteurs. A l'aide des PDD, un flot de travail ou "workflow" composé d'actions et de conditions peut être décrit. Certaines actions permettent des appels spécifiques aux fonctionnalités de gestion de l'intergiciel XSStreaMWare. Pour simplifier la manipulation des événements, des petites modifications ont été apportées aux PDD :

- Il n'y a pas de paramètres d'entrée ou de sortie. Un paramètre d'entrée est toujours accessible sans avoir à le définir : il s'agit de l'événement **e** déclenchant la politique.
- L'accès aux attributs de l'événement **e** s'effectue avec la notation pointée similaire à celle utilisée par TUNe dans les SWDL : *e.nom_attribut*.
- Une nouvelle variable typée est introduite : il s'agit d'une référence XSStreaMWare vers un capteur. Le type de la variable est **SENSOR**. Pour créer la variable, il suffit d'utiliser la même action que pour la création de liste pour les PDD, par exemple : `SENSOR s = e.source` récupère la référence XSStreaMWare du capteur qui a généré l'événement.
- Les appels par défaut aux fonctionnalités de contrôle d'XSStreaMWare s'effectuent avec la notation pointée sur une variable de type **SENSOR**, par exemple *s.get(...)*, *s.set(...)*, *s.act(...)*. Les paramètres de ces fonctions sont définis par des expressions dont la grammaire est définie en EBNF comme pour les expressions des PDDs.

GET : permet de récupérer des paramètres des capteurs.

```
expression d'appel GET = variable SENSOR, ".get("", accès au DATAMODEL, [
, operator, valeur ] , "")";
```

SET : permet de changer des paramètres des capteurs.

```
expression d'appel SET = variable SENSOR, ".set("", accès au DATAMODEL,
",", valeur, "")";
```

ACT : permet d'invoquer une commande spécifique au capteur.

```
expression d'appel ACT = variable SENSOR, ".act("", opération du
DATAMODEL, [  ",", paramètre  ] , "")";
```

TABLE 4.2 – Expressions des opérations XSStreaMWare en EBNF

Accès au DATAMODEL : permet d'exprimer un chemin pour parcourir la hiérarchie du modèle générique XSStreaMWare.

```
racine = "/";
séparateur = ".";
famille = "GeneralInfo" | "Configuration" | "Software | "Performance",
séparateur;
paramètre = nom du paramètre, séparateur;
groupe de paramètres = nom du groupe, "[", 0-9, "]", séparateur;
accès au DATAMODEL = racine, famille, [  paramètre | groupe de paramètres
];
```

TABLE 4.3 – Expression d'accès au DATAMODEL en EBNF

Nous méta-modélisons des expressions en EBNF comme pour les expressions utilisées dans les PDDs (voir tableau 4.2). Ces expressions permettent de faire appels aux fonctionnalités de contrôle **GET**, **SET** et **ACT**, en y associant des paramètres dynamiques, qui sont automatiquement évalués lors de l'exécution. Un expression particulière permet de parcourir le modèle générique de données utilisé par XSStreaMWare (le DATAMODEL).

Dans le tableau 4.2, nous voyons que l'appel de **GET** est défini sur une variable de type **SENSOR** et prend comme paramètre un accès au DATAMODEL, éventuellement suivi d'un opérateur et d'une valeur. Il permet soit de récupérer une valeur si la partie avec l'opérateur et la valeur n'est pas présente, soit de tester une condition dans le cas contraire. S'il récupère une valeur, celle-ci peut permettre à un noeud de décision situé juste après l'appel de créer différents chemins d'exécution suivant la valeur retournée. De même, s'il s'agit d'un test conditionnel, les deux chemins possibles peuvent être définis avec les valeurs de retour booléennes "[true]" ou "[false]".

L'appel de **SET** quant à lui prend comme paramètre un accès au DATAMODEL suivi d'une valeur. L'ancienne valeur du DATAMODEL située à l'endroit indiqué sera écrasée par la nouvelle valeur définie.

L'appel de **ACT** prend comme paramètre le nom d'une opération définie dans le DATAMODEL et éventuellement plusieurs paramètres.

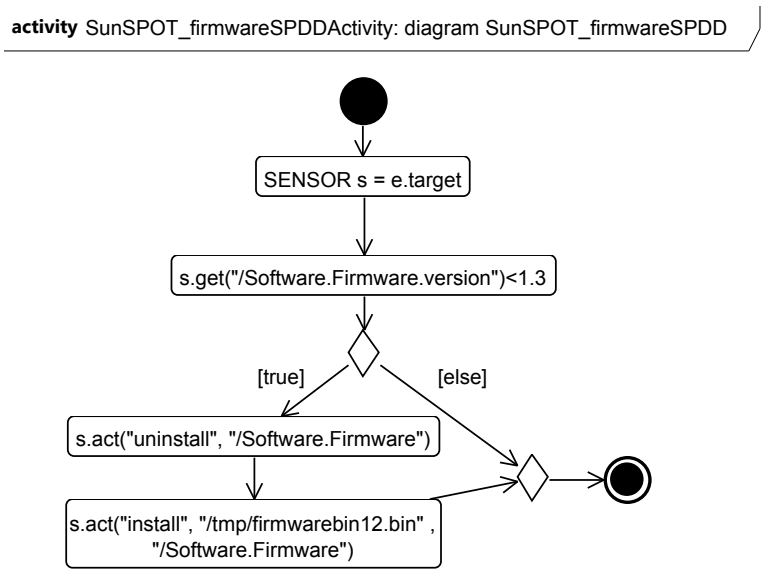


FIGURE 4.3 – PDD pour le "self-configuring" des capteurs SunSPOT : mise à jour automatique

L'accès au DATAMODEL est défini par le tableau 4.3. Par exemple, sur la représentation du DATAMODEL en figure 2.3, pour accéder à l'unité (*unit*) des types de mesures du capteur (*MeasurementTypes[i]*) se trouvant dans la famille des paramètres de configuration, il faut utiliser l'expression :

`/Configuration.MeasurementType[1].unit`

4.2.2 Validation et exemples de politiques de gestion de réseaux de capteurs

4.2.2.1 "Self-configuring" des micrologiels ou "firmwares" des capteurs

Pour cet exemple, nous disposons d'un parc de capteurs SunSPOT et nous voulons être sûrs que tous les capteurs disposent de la dernière version du "firmware" internes. Afin de rendre toutes les opérations de maintenance (comme la mise à jour) totalement transparente (pour la propriété de "self-configuring"), nous déployons des PDD décrivant des politiques de mise à jour. Quand un nouveau capteur est détecté, le système de gestion doit vérifier si la version du "firmware" utilisé est à jour et doit déployer la nouvelle version si nécessaire.

La figure 4.3 montre un exemple de **PDD** pour la gestion des mises à jour. Ce **PDD** est déployé au niveau de site de contrôle (*Control-site*) au plus haut niveau de la hiérarchie XSStreaMWare et associé au type d'événement prédéfini *HELLO*. De ce fait, cette politique est appliquée à tout le système. En effet, lors du rajout de cette politique au plus haut niveau, le site de contrôle souscrit aux événements des services inférieurs. Cette souscription récursive arrive jusqu'en bas de la hiérarchie d'XSStreaMWare qui fait alors remonter les événements correspondants au type *HELLO* jusqu'au site de contrôle. Suivant le cycle décrit par la figure 4.2, quand un nouveau capteur est détecté, un événement système **SE** spécifique est intercepté par les **proxy** en charge de ce type de capteur. Cet

événement est capturé par l'écouteur d'événement **Event Listener** et transite par le gestionnaire d'événement **Event Manager** de l'**adaptateur** du proxy qui le transforme en événement de gestion **ME**. Cet événement de gestion est ensuite diffusé à la passerelle **Gateway** de niveau hiérarchique supérieur puis au site de contrôle **Control site**.

Au niveau du moteur de règles **Rule Engine** du site de contrôle, le gestionnaire de règles **Rule Manager** récupère la règle décrite par le **PDD** de la figure 4.3 et la donne à l'évaluateur de règle **Rule Evaluator**. Ce dernier exécute le **PDD** à l'aide du même moteur d'exécution utilisé précédemment. La première action crée une variable de type *SENSOR* nommée *s* dans l'environnement du **PDD**. Cette variable fait référence au capteur nouvellement détecté par la récupération de l'attribut *target* de l'événement. En effet, cet événement étant généré par le proxy propriétaire des capteurs, l'attribut *source* représente le proxy générateur de l'événement tandis que l'attribut *target* représente le capteur lui-même. La deuxième action récupère avec un **GET** la version du "firmware" installé sur le capteur et teste si sa valeur est inférieure à la valeur fixe "1.3". Cette action renvoie donc un booléen qui est testé grâce à un noeud de décision. Si la version n'est effectivement pas à jour, le "firmware" est désinstallé à l'aide d'une action **ACT** qui fait appel à l'opération *uninstall* sur le "firmware" (accès par `/Software.Firmware`). La nouvelle version du "firmware" (stockée localement au site de contrôle dans "/tmp/firmwarebin12.bin") est ensuite déployée avec une action **ACT** qui fait appel à l'opération *install*. Notons ici que l'action **GET** est utilisée et interroge dynamiquement le capteur. On parle alors d'une politique qui utilise le mécanisme de **pulling** car c'est le système de gestion qui "tire" les informations du capteur. Dans l'exemple suivant, nous étudions le cas inverse du **pushing** dans lequel le capteur est entièrement déclencheur et "pousse" ses données à la politique.

4.2.2.2 "Self-configuring" de paramètres internes des capteurs

L'énergie embarquée dans les capteurs sans fils est une ressource importante qui doit être gérée de manière efficace. Par exemple, quand le niveau d'une batterie d'un capteur approche un certain seuil, des mesures adéquates peuvent être prises comme la réduction du taux d'échantillonnage pour économiser de l'énergie. C'est ce type de reconfiguration que nous voulons décrire avec une politique de "self-configuring" des paramètres internes des capteurs concernant la fréquence d'échantillonnage.

La figure 4.4 montre un exemple de **PDD** pour décrire une politique de "self-configuring" qui change des paramètres interne des capteurs ou qui dans certain cas peut éteindre les capteurs. Ce **PDD** est déployé au niveau des adaptateurs, c'est à dire très proche des capteurs. Ceci permet de déployer ce PDD à une granularité fine qui spécifie des lots de capteurs concernés par cette politique. Pour cet exemple, le type d'événement associé à cette politique est *PERIODIC*. Cet événement périodique est créé quand le capteur envoie périodiquement ses niveaux de batterie. Comme pour le premier exemple, l'attribut *target* de l'événement est une référence vers le capteur qui doit être configuré. Il est à noter que cet événement contient aussi les données envoyées par le capteur dans l'attribut *datamodel*. Cet attribut est une instanciation du modèle générique d'XSStreaMWare qui peut être parcouru avec une expression d'accès au DATAMODEL. L'action `e.datamodel("/GeneralInfor.batteryLevel")` permet de récupérer la valeur du niveau de batterie que le capteur a renvoyée. Cette valeur est ensuite testée par un noeud

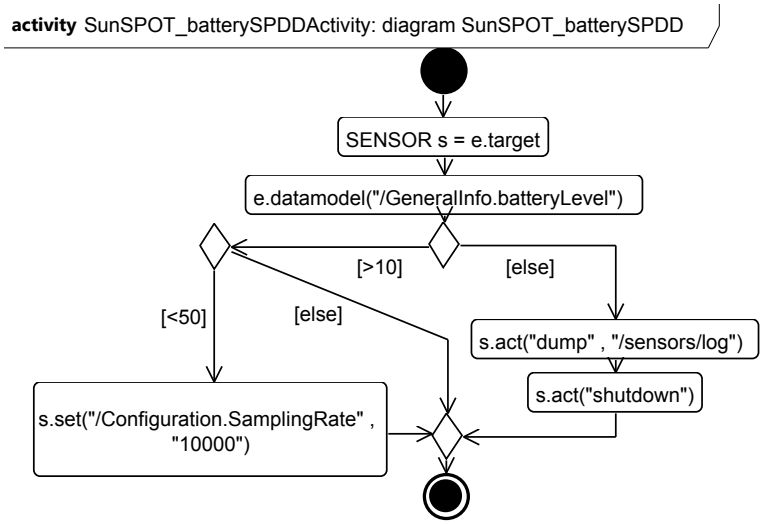


FIGURE 4.4 – PDD pour le "self-configuring" de paramètres internes d'un capteur

de décision qui crée trois chemins d'exécution possibles. Si le niveau de batterie est entre 10% et 50%, alors le taux d'échantillonnage est réduit à 10 secondes avec une action **SET**. Si le niveau est inférieur à 10%, alors l'action spécifique au capteur *dump* est déclenchée et permet de récupérer les fichiers de logs au niveau des adaptateurs, puis le capteur est éteint avec l'action *shutdown*. Notons qu'ici, contrairement à l'exemple précédent, aucune action **GET** n'est utilisée car il s'agit d'un mécanisme de **pushing**. En effet, cette politique est déclenchée sur envoi spontané des capteurs de leur niveau de batterie.

4.3 Conclusion pour l'étude de cas de "self-configuring" appliqué aux réseaux de capteurs

Nous avons proposé d'utiliser les concepts de l'autonomic computing pour la gestion des réseaux de capteurs. Notre approche de gestion des réseaux de capteurs se base sur l'utilisation du logiciel de contrôle manuel XSStreaMWare et est indépendante du domaine fonctionnel, du domaine architectural ou du type de matériel utilisé par ces réseaux de capteurs. Nous avons introduit des PDDs pour la gestion des capteurs. Ces PDDs utilisent les opérations de gestion GET, SET et ACT déjà présentes dans XSStreaMWare et manipulent les données de la hiérarchie générique du DATAMODEL. Cette approche de haut niveau permet de gérer les réseaux de capteurs dans les différents domaines fonctionnels (gestion logicielle, gestion de la configuration et gestion de la performance) en reconfigurant les capteurs suivant quatre familles de paramètres internes définies dans le DATAMODEL. Cette reconfiguration s'effectue à plusieurs niveaux dans la hiérarchie d'XSStreaMWare. En effet, nous avons introduit un service ECA (Event Condition Action) qui peut s'intégrer au niveau du site de contrôle global, au niveau d'une région de gestion ou au niveau d'un lot de capteurs d'une région. Ce service ECA réutilise le moteur de déchiffrement XMI précédemment utilisé.

Ainsi, la gestion des capteurs s'est grandement simplifiée par le rajout de la propriété de "self-configuring". En effet, tous les capteurs gérés sont automatiquement mis à jour

et paramétrés suivant des politiques que les administrateurs peuvent décrire avec un haut niveau d'abstraction. Ces reconfigurations ou mises à jour s'effectuent de manière transparente, par exemple quand un capteur s'allume ou rentre dans une zone de gestion.

Dans une seconde partie de ce chapitre, nous nous intéressons à la problématique de "self-configuring" appliquée aux applications scientifiques distribuées sur les grilles de calcul.

4.4 Problématique de "Self-configuring" pour une simulation électromagnétique sur grille

Les scientifiques sont de plus en plus intéressés dans l'utilisation des grilles de calcul pour exécuter des applications spécifiques à leurs domaines de compétence. Néanmoins, la difficulté pour un non-expert du domaine informatique de démarrer et de gérer correctement ces applications sur un grand nombre de noeuds peut vite le décourager.

De plus, les applications distribuées présentent différents modèles qui ont chacune leur spécificité de configuration. On peut en général classifier quatre modèles d'applications [60] :

- **Les applications parallèles** : elles sont composées d'exécutions simultanées qui communiquent toutes entre elles (mono-programme multi-flux de données[3]). Les technologies utilisées sont par exemple les threads [71] pour la parallélisation d'une exécution sur une machine, OpenMP [72] et MPI [11] pour la communication entre des exécutions sur des machines différentes.
- **Les applications paramétriques** : elle sont composées d'exécutions simultannées qui ne communiquent pas entre elles, c'est-à-dire qui sont indépendantes. En général, une telle application comporte un serveur appelé maître ou lanceur et des clients appelés esclaves ou codes paramétriques. Les clients sont exécutés plusieurs fois en série ou en parallèle (ou les deux) et communiquent uniquement avec le serveur pour recevoir les données à traiter et retourner les résultats. Un exemple est la famille d'applications basées sur BOINC[4] [73] tels que SETI@home [74].
- **Les applications réparties** : elles comportent des exécutions qui communiquent entre-elles mais qui ne sont pas forcément inter-connectées directement. Une exécution peut alors avoir besoin de passer à travers un ou plusieurs autres programmes pour pouvoir atteindre l'exécution distante (multi-programme multi-flux de données[5]). Les exemples sont les appels hiérarchiques à des technologies comme les appels de procédure à distance[6] [75], CORBA [76] ou les services web imbriqués [77] avec les protocoles d'accès aux objets à distance tel que SOAP [78].
- **Les applications en flots de travail** : elles sont organisées en "workflow" qui partage les activités de lecture des données d'entrée, de traitement de ces données et d'écriture des données de sortie. Une exécution peut alors avoir besoin en entrée d'une ou plusieurs autres sorties d'autres exécutions. Par exemple, le langage d'exécution de services web WS-BPEL [79] ou le moteur d'orchestration ODE [80] définissent des applications en flots de travail. Le premier uniquement pour le modèle graphique et le deuxième comme un cadriciel[7] (API) à implémenter au sein du programme.

Pour notre problématique de "self-configuring", nous voyons qu'en fonction du modèle utilisé, l'application doit être configurée d'une certaine manière. Par exemple, les applications parallèles ont besoin d'être déployées et configurées de telle sorte que leurs exécutions connaissent toutes leurs adresses respectives puisqu'elles communiquent po-

3. En anglais : Single Program Multiple Data, SPMD
4. Berkeley Open Infrastructure for Network Computing, BOINC
5. En anglais : Multiple Program Multiple Data, MPMD
6. En anglais : Remote Procedure Call RPC
7. En anglais : framework

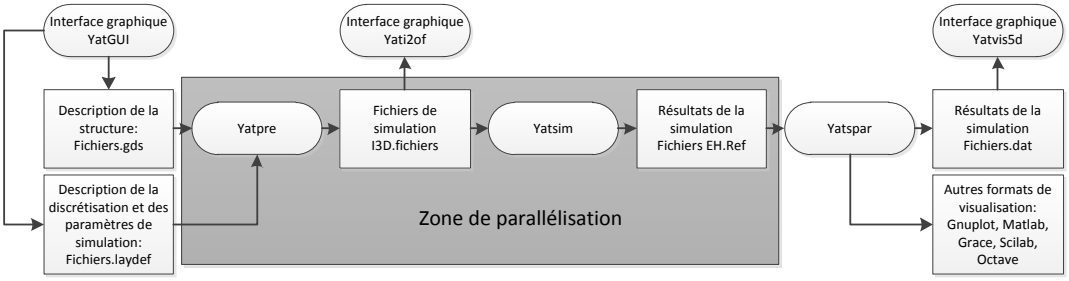


FIGURE 4.5 – Flot de travail pour la simulation YatPac

tentiellement toutes entre elles. Les applications réparties sont un cas particulier des applications parallèles qui limitent certaines inter-connections. Les applications paramétriques n'ont quant à elles besoin de connaitre qu'un seul point de communication (le serveur). Enfin, les applications en flots de travail ont un besoin supplémentaire de synchronisation par rapport aux autres modèles (attente de la fin d'exécution de certains éléments du système). Parfois, une application distribuée peut faire intervenir plusieurs modèles, c'est le cas de celle présentée dans notre approche.

Nous allons prendre comme étude de cas le déploiement et la gestion de l'application de simulation électromagnétique YatPac. Nous rajoutons la propriété de "self-configuring" en décrivant des politiques de démarrage et de configuration de cette application en suivant les différents modèles qu'elle fait entrer en jeu et nous mesurons la performance de notre approche pendant les phases de déploiement, de démarrage et de désinstallation.

4.4.1 Simulation électromagnétique et grille de calcul

Les simulations électromagnétiques font intervenir des modèles d'application paramétriques (plusieurs simulations indépendantes) ou des modèles d'application en flots de travail (suite de processus pour atteindre les résultats de la simulation).

Le besoin en calcul étant parfois considérable, la parallélisation massive de ces simulations rajoute des besoins de configuration dynamiques, ce qui peut devenir rapidement ingérable de façon manuelle pour des environnements à grande échelle. L'idée est de fournir un moyen de rajouter la propriété de "self-configuring" pour ce genre d'application scientifique sur une grille de calcul, qui respecte les différents modèles d'applications mis en jeu. En utilisant des politiques définies par des PDDs, nous voulons montrer qu'il est possible de déployer, de démarrer, d'arrêter et de récupérer les résultats de simulations qui font intervenir les modèles d'applications paramétriques et en flots de travail sur une grille de calcul. Notre étude de cas porte sur le "self-configuring" du simulateur YatPac déployé sur la grille Grid'5000.

4.4.1.1 Flot de travail et parallélisation de la simulation

Le simulateur YatPac est basé sur la modélisation par lignes de transmission **TLM**. Il permet de faire des simulations dans le domaine temporel, c'est à dire de simuler le

comportement électromagnétique dans le temps pour une fréquence donnée et une structure donnée. Le but est ici de lancer plusieurs simulations qui ont chacune une suite de processus à lancer (modèle d'application en flot de travail), le tout en parallèle avec des simulations qui soient indépendantes entre elles (modèle d'application paramétrique). L'idée est de modifier le paramètre de fréquence ou la forme de la structure pour chaque simulation lancée, c'est le paramètre dynamique pour le modèle d'application paramétrique. Pour ce faire, la simulation doit être lancée plusieurs fois avec des fichiers d'entrée différents. Les résultats sont ensuite récupérés et centralisés.

La figure 4.5 représente le flot de travail de la simulation tel qu'il est utilisé par les ingénieurs électroniciens utilisant YatPac. Toutes les actions de déploiement, de configuration et de démarrage de tous les modules étaient initialement effectuées à la main. Face au travail considérable que cela représentait, une première tentative d'automatisation du déploiement a été considérée [81] à l'aide de scripts. Ces scripts permettaient de copier les fichiers d'entrées sur chacune des machines réservées et de lancer les différents modules suivant le flot de travail voulu. Néanmoins, cette approche restait statique car les différentes ressources matérielles étaient réservées manuellement et en fonction des résultats de la réservation des ressources, les informations étaient rentrées manuellement dans des scripts.

D'après la figure 4.5, nous voyons que l'interface graphique YatGUI permet de décrire les structures à simuler dans un fichier de type *gds* et la discrétisation de la simulation, ainsi que les paramètres de simulation (tels que la fréquence et la durée de simulation) dans un fichier de type *laydef*. Le préprocesseur génère un fichier de simulation à partir de ces deux fichiers et les résultats de la simulation peuvent être convertis en différents formats pour visualisation ultérieure. Nous avons défini une zone de parallélisation possible qui permet de lancer de multiples simulations indépendantes car elles se basent sur des entrées différentes et fournissent des résultats différents. Il s'agit donc du modèle d'application paramétrique. Cette zone encadre le préprocesseur *yatpre* et le simulateur *yatsim*. Le but est donc de pouvoir définir une politique de "self-configuring" avec un PDD pour déployer les modules du simulateur yatpac sur plusieurs machines de la grille de calcul. Le modèle d'application paramétrique doit être respecté en exécutant les simulations indépendantes de manière parallèle. De plus, le modèle d'application en flot de travail doit synchroniser les différents modules du simulateur au sein d'une simulation.

4.4.1.2 HDD de la simulation

Pour la description de la plateforme matérielle, nous utilisons un **HDD** représenté par la figure 4.6, décrivant la plateforme Grid'5000. Ce **HDD** est comparable à la figure 3.3 utilisé pour la validation des propriétés de "self-healing" avec DIET. C'est en effet toujours l'ordonnanceur de Grid'5000 OAR qui est utilisé à travers le plugin OARTUNe (attributs *type=oargrid* et *protocole=oarsh*) La différence est qu'ici nous spécifions le nombre de machines utilisées pour chacun des sites de la grille afin de contrôler le temps mis pour le déploiement à chaque fois qu'un site de la grille est rajouté à l'expérience. Pour les expériences menées, nous validons notre approche en mesurant les temps de déploiement sur un site puis en rajoutant un site supplémentaire à chaque fois.

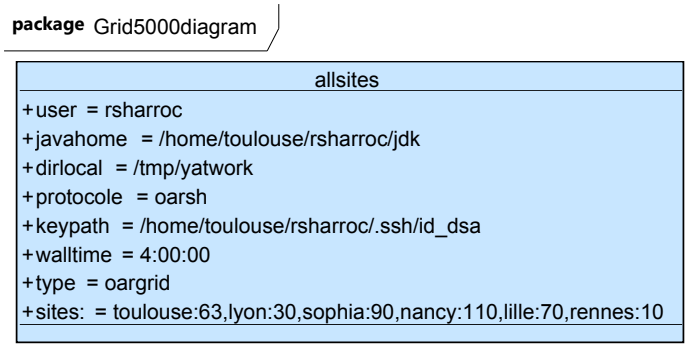


FIGURE 4.6 – HDD pour la simulation YatPac

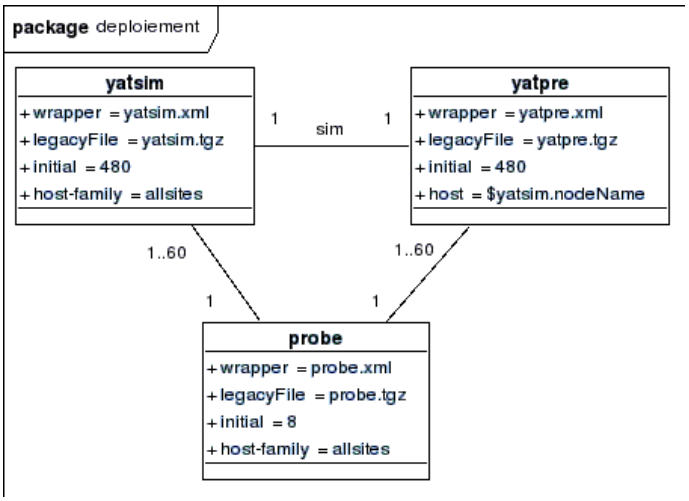


FIGURE 4.7 – SDD pour la simulation YatPac

4.4.1.3 SDD de la simulation

Pour la description de la plateforme logicielle, nous utilisons un **SDD** représenté par la figure 4.7. Pour les applications suivant un modèle en flot de travail, il est parfois judicieux de placer les exécution faisant partie du même flot de travail sur la même machine (pour limiter les transferts de fichiers, par exemple), or il n'existe aucun moyen de spécifier cette contrainte dans les HDD et SDD actuels. Notre approche consiste à prendre en compte cette contrainte et donner une possibilité de l'exprimer dans le SDD de l'application qui suit un modèle de flot de travail.

Nous remarquons que deux modules du simulateur *yatpac* sont utilisés : *yatsim* (le simulateur) et *yatpre* (le préprocesseur). Ces deux modules font partie du même flot de travail (figure 4.5) et il est donc pertinent de les placer sur la même machine pour éviter de transférer les fichiers de communication sur le réseau. Nous introduisons ici l'expression d'une contrainte qui permet de spécifier le placement d'un module logiciel sur la même machine qu'un autre module.

Pour respecter le modèle d'application en flot de travail et pour faciliter la transmission des résultats entre le préprocesseur et le simulateur, nous voulons forcer les modules *yatpre* à être déployés sur le même noeud que les modules *yatsim* respectifs. Pour cela nous introduisons un nouvel attribut **host** qui remplace l'attribut **host-family** et qui permet

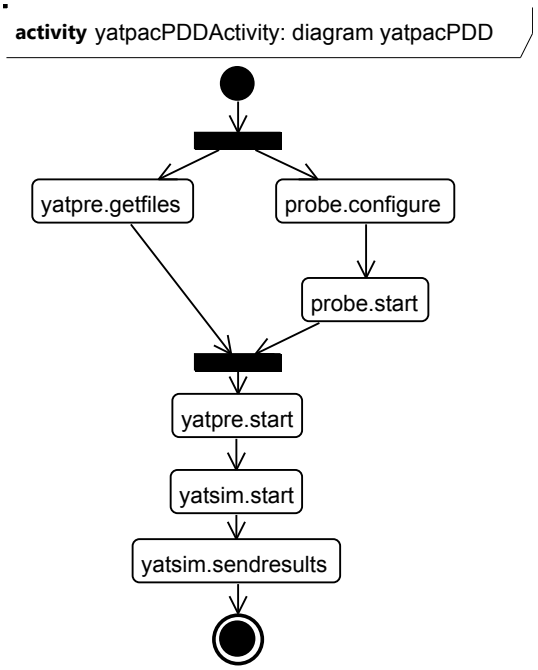


FIGURE 4.8 – PDD pour la simulation YatPac

de spécifier soit de manière statique l'adresse du noeud sur lequel déployer les instances, soit de manière dynamique en utilisant la notation `"$nom_de_classe_SDD.nodeName"`. Ici, chaque instance de la classe *yatpre* est liée à une instance de *yatsim* par l'association nommée *sim*. L'attribut `"host = $yatsim.nodeName"` spécifie que chaque instance de *yatpre* doit être déployée sur le même noeud que l'instance de *yatsim* à laquelle elle est liée.

Les binaires des modules sont inclus respectivement dans les archives *yatsim.tgz* et *yatpre.tgz*. Dans cet exemple, nous déployons initialement jusqu'à 480 simulations en parallèle (attribut *initial=480*). Ce nombre étant supérieur au nombre de noeuds dans la réservation spécifiée par le **HDD**, certains noeuds feront tourner plusieurs simulations en parallèle. Ceci permet de respecter le modèle d'application paramétrique en exécutant toutes les simulations indépendantes en parallèle.

Enfin, nous utilisons des sondes génériques fournies avec TUNe (classe *probe*) pour contrôler le déploiement et le démarrage des simulations. Nous déployons 8 sondes qui peuvent gérer chacune 60 simulations (cardinalités *1..60* des liaisons *probe-yatsim* et *probe-yatpre*).

4.4.1.4 PDD de la simulation

Pour décrire le flot de travail d'une simulation, nous utilisons un **PDD** représenté par la figure 4.8. Ce diagramme ne contient que des noeuds de type *fork/join* et des appels de méthodes SWD pour le module *yatpre* (SWD *yatpre.xml*) et *yatsim* (SWD *yatsim.xml*). En premier, les fichiers qui décrivent les différentes structures (`fichiers.gds`) et les paramètres de fréquence et de discrétisation (`fichiers.laydef`) sont préparés et envoyés sur la machine d'accès à la grille de calcul. L'action *yatpre.getfiles* permet de récupérer les fichiers correspondants à une simulation depuis la machine d'accès. Nous pouvons

```
<wrapper name='yatsim'>
  .....
  <method name="sendresults" key="extension.GenericCommands"
   method="start" >
    <param value="scp $dirLocal/outputs/*
    frontale.toulouse.grid5000.fr:~/results"/>
    <param value="$dirLocal"/>
    <param value="$node.user"/>
    <param value="$node.keypath"/>
    <param value="$nodeName"/>
  </method>

  <method name="start" key="extension.GenericCommands"
   method="start" >
    <param value="./yatsim $dirLocal/I3Dinputs $dirLocal/outputs"/>
    <param value="$dirLocal"/>
    <param value="$node.user"/>
    <param value="$node.keypath"/>
    <param value="$nodeName"/>
    <param value="LD_LIBRARY_PATH=$dirLocal"/>
  </method>
......
</wrapper>
```

FIGURE 4.9 – SWDL utilisés pour la simulation YatPac

voir qu'en parallèle de la récupération des fichiers d'entrée, une sonde est démarrée avec les appels *probe.configure* et *probe.start*. Une fois les fichiers récupérés et la sonde démarrée, le préprocesseur *yatpre* génère le fichier servant à la simulation (`I3D.fichier`) à partir des deux fichiers d'entrée avec l'action *yatpre.start*. Ce fichier de simulation est donné au module de simulation *yatsim* qui démarre le calcul TLM requis grâce à l'action *yatsim.start*. Enfin, les fichiers résultats `EH.Ref` sont envoyés sur la machine d'accès avec l'action *yatsim.sendresults*. Le modèle d'application en flot de travail peut donc être décrit simplement en utilisant des appels de méthode SWD séquentiels. En effet, une transition entre un appel de méthode SWD et un autre appel s'effectue quand le premier appel se termine. Dans notre cas, nous utilisons un SWD fourni par défaut par TUNe qui permet de lancer l'exécution d'une commande à distance et qui se termine une fois l'exécution de celle-ci terminée (d'autres SWD permettent de lancer des commandes et de ne pas attendre la fin de celles-ci). Ceci permet de garantir que les fichiers créés par le préprocesseur sont effectivement disponibles pour le simulateur, car celui-ci n'est démarré qu'une fois que le préprocesseur a terminé son exécution.

4.4.1.5 SWD de la simulation

La figure 4.9 donne une partie du fichier *yatsim.xml* montrant les méthodes *sendresults* et *start* du SWD pour le simulateur. Nous pouvons voir que la méthode *sendresults* lance

une commande *scp* qui va copier les fichiers résultats sur la frontale du site toulousain (*frontale.toulouse.grid5000.fr*) dans le dossier *~/results*. Nous pouvons voir que le nom du dossier source des fichiers est récupéré de manière dynamique avec *$dirLocal/outputs/**. Les paramètres suivants représentent l'endroit où exécuter cette commande (*$dirLocal*), avec quel nom d'utilisateur et quelle clé SSH il est possible de se connecter au noeud *$nodeName* concerné pour la copie (*$node.user* et *$node.keypath*). Tous ces paramètres étant interprétés dynamiquement lors de l'exécution de l'appel de la méthode, la gestion de la configuration est prise en compte par notre approche.

Lors du démarrage des différents modules de la simulation, la problématique est de disposer de librairies nécessaires et compatibles avec le matériel sur lequel chaque module est déployé. Il s'agit de trouver l'adéquation entre le matériel et le logiciel pour assurer une compatibilité. Cette adéquation est effectuée en amont au moment de la création du SDD et du HDD. En effet, chaque famille de matériel étant décrite par une classe du HDD, il est possible de spécifier qu'une famille se définit comme un ensemble de machines ayant la même compatibilité pour le logiciel qui doit être déployé. Dans notre cas, les librairies pour le simulateur YatPac sont compatibles avec toutes les machines de la grille Grid'5000 mais ne sont pas pré-installées sur celles-ci. Elles doivent donc être déployées en même temps que les binaires des modules les utilisant.

En reprenant le SWD de la figure 4.9, nous voyons que la méthode *start* exécute le binaire *yatsim* en lui passant en paramètres le dossier des fichiers d'entrée *$dirLocal/I3Dinputs* et de sortie *$dirLocal/outputs*. De même, cette commande est exécutée dans le dossier *$dirLocal* sur le noeud *$nodeName* en utilisant l'utilisateur *$node.user* et la clé SSH *$node.keypath*. De plus, le simulateur devant utiliser des librairies spécifiques, elles sont incluses dans l'archive *yatsim.tgz*, ces dernières sont alors disponibles lors du déploiement directement dans le répertoire de travail. Le paramètre qui permet de faire pointer la variable d'environnement est *LD_LIBRARY_PATH=$dirLocal*. Il spécifie l'emplacement de ces librairies vers ce répertoire de travail.

4.4.2 Validation de l'approche de "self-configuring" pour YatPac sur Grid'5000

4.4.2.1 Conditions d'expérimentations

En premier lieu, nous listons les trois étapes nécessaires au lancement d'une simulation YatPac sur la grille Grid'5000 :

- Préparation locale de la simulation : les archives *tgz* contenant les modules de YatPac et les librairies doivent être créées. Nous fournissons des outils permettant de créer ces archives de manière automatisée avec un script ldd récursif. En pointant un fichier binaire, toutes les librairies ainsi que les dépendances de ces librairies sont intégrées dans l'archive automatiquement. Les archives pour les sondes sont fournies par TUNe. Les ingénieurs électroniciens préparent tous les fichiers d'entrée dans un dossier `inputs`.
- Création des HDD, SDD, SWD et PDD pour TUNe : les quatre descriptions sont créées à l'aide de l'outil Topcased. Les SWD peuvent en particulier se baser sur des squelettes déjà existants et les HDD, SDD et PDD peuvent être intégrés dans un seul fichier xmi.

Expérience	Rennes	Nancy	Nice	Toulouse	Lyon	Lille	Bordeaux	Paris
Expérience 1	20	20	20	20	20	20	20	20
Expérience 2	100	110	90	63	30	70	0	0

TABLE 4.4 – Localisation des noeuds pour les expériences

– Copie et lancement de TUNe sur la grille : Tous les fichiers créés précédemment et le dossier contenant les entrées de simulation sont copiés sur la machine d'accès à la grille de calcul et TUNe est lancé avec la commande `java -jar TUNe.jar`. Il suffit ensuite de demander le déploiement à TUNe avec la commande `deploy <nom du dossier contenant tous les fichiers> <nom du fichier xmi>`.

Pour valider notre approche, nous avons effectué deux expériences avec un nombre de noeuds différents sur la grille de calcul (voir Tableau 4.4). La première expérience est un petit déploiement concernant 8 sites et 20 noeuds sur chaque site. La deuxième expérience est un déploiement à plus grande échelle avec 6 sites et entre 30 et 110 noeuds sur chaque site. Pour ces expériences, nous voulons mesurer la vitesse de déploiement, de démarrage et d'arrêt des simulations. Pour cela, nous augmentons graduellement le nombre de sites (un site de plus à chaque fois), en relançant TUNe à chaque fois que nous ajoutons un site. Sur les figures 4.10 à 4.13 le premier point d'une expérience représente le déploiement sur un site et le dernier point sur tous les sites de l'expérience. Pour choisir quels sites sont concernés, nous devons seulement modifier l'attribut *sites* du HDD (figure 4.6). Notons que ceci est un moyen simple de rajouter ou supprimer un site pour une simulation donnée.

4.4.2.2 Déploiement

Le déploiement consiste en trois phases internes à TUNe :
– La lecture des SDD, HDD et la réservation des noeuds sur la grille de calcul,
– La création complète de la représentation du système interne SR,
– Le déploiement effectif de l'application sur les machines.
Nous détaillons ces trois phases :

Lecture des SDD, HDD et réservation des noeuds

Quand TUNe reçoit la commande `deploy`, le moteur de déchiffrement XMI est démarré et lit les SDD, HDD et PDD. Le HDD utilisant le plugin OARTUNe, il est démarré et effectue la réservation des ressources auprès d'OAR en exécutant la commande `oargrid`.

Création complète du SR

La figure 4.10 montre combien de temps TUNe passe à créer sa représentation interne du système à déployer (la couche SR). Ce temps inclut l'instanciation du SDD et du HDD. Pour chaque instance des classes du SDD, TUNe crée un composant Fractal de type logiciel. Tous les composants sont ensuite liés entre eux conformément aux liens du diagramme SDD et à leurs cardinalités. De même pour chaque noeud réservé sur la grille

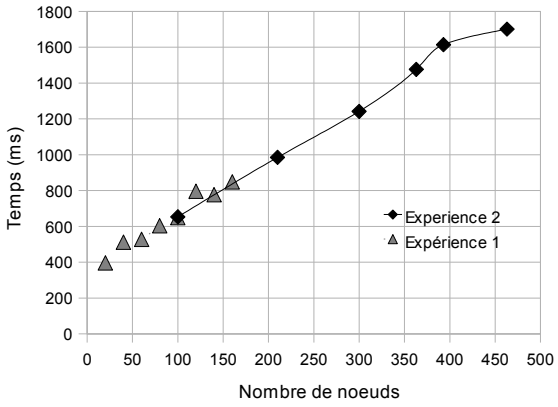


FIGURE 4.10 – Temps de création du SR interne à TUNe

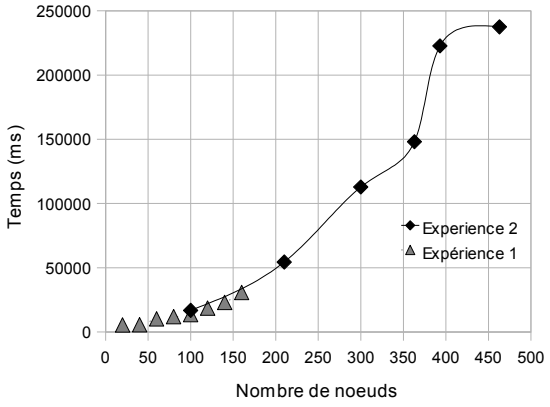


FIGURE 4.11 – Temps de déploiement de la simulation yatpac sur Grid'5000

de calcul, TUNe crée un composant Fractal. Tous les composants de type noeud sont liés aux composants de type logiciel suivant les attributs `host-family` et `host`. Notons ici que lors du rajout de l'attribut `host`, un ordre de création doit être respecté. En effet, une instance de classe SDD utilisant un attribut `host` au lieu d'un attribut `host-family` doit être reliée à la même instance du HDD spécifiée par l'attribut. Pour cela, il faut que cette instance du HDD existe déjà. Si ce n'est pas le cas alors la création est reportée (mise à la fin de la liste de création à traiter).

La figure 4.10 montre clairement que le temps requis pour cette procédure de création du SR varie de manière linéaire en fonction du nombre de noeuds impliqués. Ce temps varie de 400 ms pour 20 noeuds jusqu'à 820 ms pour 160 noeuds sur 8 sites (expérience 1) et de 650 ms pour 110 noeuds à 1700 ms pour 460 noeuds sur 6 sites (expérience 2).

Installation complète sur les machines

La figure 4.11 montre le temps utilisé par TUNe pour déployer l'application sans la lancer. Ce temps inclut : la connexion au noeud distant concerné, la création du répertoire de travail, la création des liens de communication en sockets RMI (mécanisme d'appels de procédures distantes Java utilisé par TUNe) entre le noeud et TUNe, la copie de l'archive

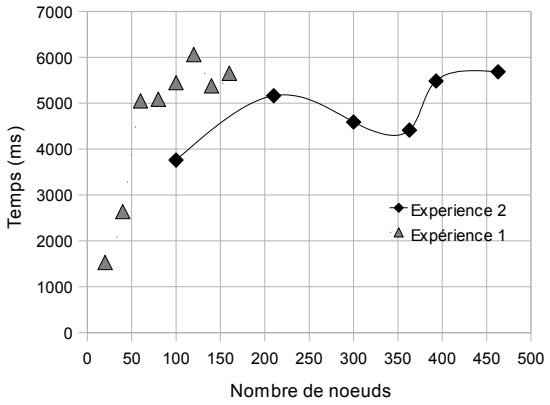


FIGURE 4.12 – Temps pour démarrer la simulation yatpac sur les noeuds Grid'5000

concernée sur le noeud et sa décompression. La proportion du temps passé entre la mise en place de TUNe sur le noeud et la copie/décompression de l'archive dépend de la taille de l'archive. Pour des tailles d'archive inférieures à 1 Mo, 20% du temps est utilisé pour les copier et les décompresser. Pour des tailles d'archive de 100 Mo, la part monte à 95 % du temps utilisé. Pour le simulateur yatpac, la taille des archives contenant les librairies n'excède pas 750 Ko (entre 15 et 20% du temps de déploiement est utilisé pour copier ces archives et les décompresser).

Sur la figure 4.11, nous pouvons voir que le temps total de déploiement sur 20 noeuds sur 1 site est de 5500 ms et monte jusqu'à 30800 ms pour 160 noeuds sur 8 sites (expérience 1). Ce temps est de 16800 ms pour 100 noeuds sur 1 site et 237500 ms (4 minutes) pour 463 noeuds sur 6 sites (expérience 2). Nous observons aussi que les deux expériences ont des temps confondus entre 100 et 160 noeuds et qu'en moyenne le temps de déploiement augmente de manière exponentielle. Ceci est dû au fait que tous les fichiers sont au départ centralisés sur la machine d'accès à la grille. Le déploiement étant multithreadé, l'ensemble des copies en `scp` s'effectue de manière parallèle mais depuis un seul noeud de départ. De plus, de multiples initiations de sockets TCP sont effectuées en parallèle. Nous proposons des solutions d'amélioration de cette performance dans les perspectives de cette section de chapitre.

4.4.2.3 Démarrage de la simulation

La figure 4.12 montre le temps passé par TUNe pour envoyer les commandes de démarrage et de configuration, c'est à dire la somme des temps d'exécution des actions du PDD de "self-configuring" (figure 4.8). Ce temps inclut la génération des commandes depuis les SWDL (incluant le temps d'interprétation des variables précédées d'un $ avec leur valeur dynamique), la génération des fichiers de configuration et le lancement des commandes à distance. Notons que nous n'avons pas inclus les temps de simulation électromagnétique (temps d'exécution des instances de `yatsim`) ou de préprocessing (temps d'exécution des `yatpre`), mais uniquement les temps rajoutés par les mécanismes inhérents à TUNe. Ce temps varie de 1500 ms pour 20 noeuds sur 1 site à une moyenne stable de 5000 ms si le nombre de noeuds est supérieur à 60. Ceci veut dire que ce temps n'augmente pas considérablement en fonction du nombre de noeuds, contrairement au temps de déploiement.

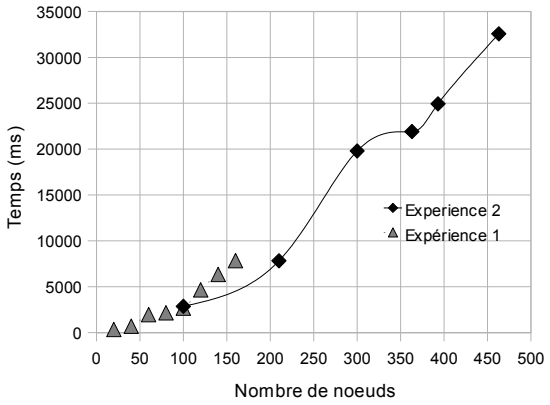


FIGURE 4.13 – Temps pour arrêter et nettoyer la simulation yatpac sur les noeuds Grid'5000

Nous pouvons cependant observer que pour le même nombre de noeuds, les temps de lancement des commandes sont moins importants quand les noeuds sont concentrés sur un plus petit nombre de sites de la grille de calcul. Par exemple, avec 100 noeuds sur 1 site (premier point de l'expérience 2), ce temps est de 3760 ms alors que pour 100 noeuds sur 5 sites (cinquième point de l'expérience 1) ce temps est de 5440 ms.

Les temps de démarrage pour toutes les simulations dans le cas d'un modèle d'application paramétrique dépend donc de la distance géographique de chacune des exécutions.

4.4.2.4 Arrêt de la simulation

La figure 4.13 montre les différents temps utilisés par TUNe lorsque des simulations sont terminées. Ce temps inclut le temps de nettoyage des répertoires de travail, l'arrêt des connections TCP utilisées et éventuellement la désinstallation des démons TUNe sur les noeuds (s'il ne reste pas d'autres simulations sur le même noeud). Notons encore une fois que nous ne prenons pas en compte les temps passés pour la copie des résultats car ils sont trop variables en fonction de la taille des fichiers résultats (pouvant varier de quelques secondes à plusieurs minutes). La figure 4.13 montre que le temps d'arrêt engendré par TUNe augmente avec le nombre de noeuds concernés, démarrant à 349 ms pour 20 noeuds sur 1 site jusqu'à 7845 ms pour 160 noeuds sur 8 sites (expérience 1) et démarrant à 286 ms pour 100 noeuds sur 1 site à 32568 ms pour 463 noeuds sur 6 sites (expérience 2). Comme pour le déploiement initial, notons qu'un nombre important de connections TCPs sont initiées pendant cette phase.

4.4.3 Conclusion pour l'étude de cas de "self-configuring" appliqué aux simulations électromagnétiques sur grille de calcul

Les grilles de calcul offrent une grande quantité de ressources de calcul pour les scientifiques, mais en même temps les applications distribuées dans un grand environnement sont de plus en plus complexes à gérer. En effet, les configurations des applications doivent

s'adapter à la multiplicité des modèles d'applications distribuées (parallèles, réparties, paramétriques ou en flots de données).

Nous avons rajouté la propriété de "self-configuring" à une application de simulation électromagnétique qui suit le modèle paramétrique et le modèle en flot de données. Notre approche a permis d'exécuter de manière parallèle des centaines de simulations indépendantes sur les noeuds de la grille de calcul en suivant le modèle paramétrique. Elle a permis de décrire le flot de données de la simulation en synchronisant les divers modules d'une simulation pour obtenir les résultats.

Notre approche a finalement simplifié le déploiement, la configuration, le démarrage et la récupération des résultats du simulateur YatPac sur Grid'5000 pour les ingénieurs en électronique du LAAS-CNRS. Les temps de déploiement et de démarrage ont été mesurés et sont négligeables si on les compare aux temps de simulation qui durent entre une et trois heures chacune. Notons également qu'une interface graphique a été créée pour une utilisation totalement intégrée de TUNe avec Grid'5000.

4.5 Conclusion et perspectives

Dans ce chapitre, nous avons rajouté la propriété de "self-configuring" à deux types de systèmes : une simulation électromagnétique sur grille de calcul et des systèmes de réseaux de capteurs. Cette propriété permet au système de s'installer, de se paramétrer et de se mettre à jour tout seul. La **gestion du déploiement** a grandement simplifié l'installation, le démarrage, la désinstallation et l'arrêt des modules logiciels pour la simulation électromagnétique sur la grille de calcul. L'introduction des descriptions de politiques à l'aide des **PDD** a permis d'appliquer la gestion aux réseaux de capteurs déjà déployés sur le terrain. Ces politiques mettent à jour les firmwares des capteurs ou reconfigurent des paramètres internes aux capteurs en fonction des politiques choisies.

Les PDDs ont montré la faisabilité d'ajout de la propriété de "self-configuring" à des systèmes déjà existants sans avoir à les modifier, ce qui garantit la généricité de notre approche.

Certains points restent à améliorer : pour un déploiement massif d'une application distribuée, TUNe est limité car le déploiement est contrôlé et centralisé à partir du noeud sur lequel il s'exécute. Ce noeud doit éventuellement avoir une limite du nombre maximum de connections TCP en parallèle augmentée. Cette limite se configure dans le kernel du système d'exploitation de la machine de départ. Nous notons aussi que cette machine doit avoir un "timeout" TCP augmenté pour passer les possibles congestions quand un nombre important de connections TCP sont initiées en même temps.

Un moyen possible d'augmenter la performance de déploiement de TUNe serait de trouver automatiquement le meilleur nombre de copies en parallèle à exécuter pour maximiser le débit sortant. Un autre moyen serait d'utiliser plusieurs noeuds sources pour la copie des fichiers. Il est aussi possible d'envisager l'intégration d'un outil de copie parallèle comme Taktuk [82].

Chapitre 5

Auto-optimisation - "Self-optimizing"

Sommaire

5.1 Introduction 95
5.2 "Self-optimizing" appliqué aux reconfigurations matérielles 96
5.2.1 Rappel sur la Qualité de Service au niveau logiciel 97
5.2.2 Environnement considéré par l'approche 100
5.3 Modèle mathématique pour la gestion autonomique de l'énergie et de la qualité de service des réseaux filaires. 101
5.3.1 Formalisation pour la partie réseau 102
5.3.2 Formalisation pour la partie application 104
5.3.3 Formalisation du critère global de minimisation 107
5.3.4 Validation et expériences 108
5.3.5 Comparaison du cas optimal et de l'utilisation de l'heuristique 114
5.4 Conclusions du chapitre "self-optimizing" 117

5.1 Introduction

L'externalisation des données et des services informatiques (sites webs, bases de données, applications distribuées) vers des "datacenters" spécialisés est de plus en plus adoptée pour confier la gestion complexe de l'infrastructure à des experts. Certains de ces "datacenters" fournissent des services de "cloud computing" qui permettent d'accéder à une infrastructure matérielle ou un environnement logiciel, parfois en l'espace de quelques secondes. Côté client, ces services permettent d'adapter le nombre de serveurs ou de services applicatifs dynamiquement, en fonction de la demande fluctuante. Côté administrateur du "datacenter", le mélange de tous les clients et de toutes les applications sur l'infrastructure peut nécessiter une optimisation de l'utilisation des ressources matérielles, par exemple pour réduire la consommation électrique du "datacenter", tout en garantissant une certaine qualité de service pour les clients.

L'optimisation reste un sujet très vaste qui peut potentiellement faire intervenir un ensemble important de paramètres internes aux logiciels ou aux matériels considérés. Nous

constatons néanmoins qu'en règle générale pour un système distribué, plus le nombre de machines constituant le coeur du système est important, plus les performances du service rendu sont bonnes (par exemple un temps de latence réduit pour des requêtes vers un serveur web, un débit plus élevé ou un temps de calcul globalement plus court). Mais ceci est fait au détriment d'une forte consommation de ressources matérielles. Inversement, plus la consommation de ressources est économe (par exemple en énergie), plus les performances du service sont faibles. Il y a donc un compromis à trouver entre, d'une part, la quantité de ressources nécessaires à un service et d'autre part les performances de ce service. D'une manière générale, l'administrateur humain d'un tel système teste manuellement, pour une charge particulière du système, plusieurs configurations de ce système avant de trouver la plus appropriée. Ceci reste une tâche fastidieuse qui doit être répétée à chaque modification de la charge du système.

La problématique principale vient du fait que la gestion humaine de cette optimisation coûte de plus en plus cher aux entreprises prenant en charge l'externalisation des ressources matérielles et logicielles. Ceci est dû au fait que les actions d'administration étant initiées par des humains, le temps de réponse pour une gestion de l'optimisation des infrastructures est particulièrement lente. De plus, la mise en place de l'adaptation du nombre de serveurs en fonction des fluctuations des demandes comporte un risque d'erreur si elle est effectuée manuellement. Dans notre approche, nous voulons introduire des propriétés de "self-optimizing" qui appliquent les actions de reconfiguration de manière autonome, sans intervention humaine.

Dans ce chapitre, nos contributions de "self-optimizing" sont appliquées aux reconfigurations matérielles, notamment des routeurs dans un réseau filaire.

5.2 "Self-optimizing" appliqué aux reconfigurations matérielles

L'idée est de fournir à l'administrateur un moyen de "self-optimization" autonomique pour la gestion du réseau du "datacenter" qui prend en compte, d'une part, l'optimisation des coûts énergétiques liés aux équipements réseaux et d'autre part, la qualité de service fournie pour les applications utilisatrices du réseau du "datacenter".

Nous introduisons la problématique générale de "self-optimizing" appliquée aux réseaux filaires. La gestion du réseau est confiée à un gestionnaire autonomique qui se charge d'optimiser un compromis entre la consommation électrique du réseau et la qualité de service fournie aux applications.

Nous introduisons un critère qui est fonction de la puissance électrique globale consommée par le réseau et d'un paramètre représentant la perte en qualité de service par les applications utilisatrices de ce réseau. Le problème d'optimisation consiste à minimiser ce critère. Cette minimisation doit aussi prendre en compte certaines contraintes comme la topologie dynamique (extinction des liens, des équipements réseau ou des machines), les capacités des liens du réseau, la politique d'équilibrage de charge et la politique de routage (les deux étant liées) entre tous les noeuds du "datacenter".

Dans les deux sous-parties suivantes, nous positionnons notre approche au niveau logiciel et matériel en faisant tout d'abord des rappels sur les notions de QoS et en décrivant l'environnement de notre approche.

5.2.1 Rappel sur la Qualité de Service au niveau logiciel

La qualité de service (QdS/QoS [1]) est la capacité à estimer dans de bonnes conditions des caractéristiques pour un type de trafic donné, en termes de disponibilité, sécurité, débit, délais de transmission, gigue ou taux de perte de paquets par exemple.

Les caractéristiques de QoS reflètent les aspects qualitatifs d'une application qu'un utilisateur désire contrôler. En d'autres termes elles expriment les besoins de QoS que l'utilisateur veut obtenir. Ces exigences de QoS peuvent aussi être dictées par la nature de l'application. Par exemple, une vidéoconférence a besoin d'un débit élevé et d'un délai de transit défini entre des parties. A l'opposé, un système bancaire en ligne exige une transmission fortement fiable et sécurisée, alors que le débit peut être moins significatif. Le réseau quant à lui, est responsable de garantir la QoS pour les applications en utilisant par exemple, différents protocoles de routage et des mécanismes pour gérer ou éviter les congestions.

5.2.1.1 Différence entre profil de QoS et caractéristique de QoS.

Il est important de différencier le profil de QoS[83] d'une application et les caractéristiques de QoS. En effet, l'un agit sur l'autre. Le profil de QoS est un ensemble de valeurs internes à l'application qui influence les caractéristiques de QoS. Par exemple, une application vidéo peut avoir deux types de compression possibles pour les images[84], l'une utilisant l'algorithme BZIP2 et l'autre Lempel-Ziv. Le paramètre interne est l'algorithme de compression et il influence (entre autres) le délai de transmission car les deux algorithmes ont des temps de compression différents. On peut donc dire que le profil de l'application est décrit par l'algorithme de compression et une des caractéristiques de QoS influencée est le délai de transmission. Par exemple, en utilisant l'algorithme BZIP2, l'application a un délai de 20 ms, tandis qu'avec l'algorithme Lempel-Ziv le délai est de 40 ms. Il faut donc distinguer deux niveaux pour une application : les différents modes de fonctionnement internes : les profils et les caractéristiques de QoS influencées. Une application intelligente choisirait automatiquement son mode de fonctionnement interne pour satisfaire une liste de caractéristiques de QoS, ce qui est un problème complexe d'optimisation en soi.

5.2.1.2 Expression de la Qualité de Service

Les caractéristiques de QoS sont des valeurs mesurables (exprimées comme des nombres réels positifs, des fractions ou des valeurs booléennes) et un ensemble de contraintes sur ces valeurs. Par exemple, une certaine application de type vidéo conférence[85] a besoin de deux caractéristiques QoS : une bande passante minimum garantie égale à 10Mbps et une gigue inférieure à 20ms.

Les caractéristiques de QoS sont majoritairement classées dans les groupes suivants [86], appelés les noyaux des caractéristiques de Qualité de Service :

caractéristiques **temporelles**, les propriétés relatives au temps :

- **Délai de transit** : l'intervalle entre la transmission d'un message et sa réception ;
- **Délai de réponse** : l'intervalle entre une demande et sa réponse ;

1. Quality of Service

- **Délai d'établissement** : l'intervalle entre la demande de connexion et son ouverture ;
- **Gigue** : la variation entre les temps de réception des messages successifs.

caractéristiques **de capacité**, le volume des données :

- **Cadence d'entrée**, **débit et cadence de sortie** : la cadence des données soumises à un service, transférées par un service ou livrées par un service ;
- **Cadence des données utilisateur** : la cadence de transfert pour les données finalement utilisées (excluant les overheads protocolaires) ;
- **Cadence des données de contrôle** : la cadence de transfert pour les paramètres de service ou de protocole et la signalisation y compris relative à la QoS.

caractéristiques **d'intégrité**, le niveau d'altération ou de destruction volontaire ou accidentelle des données ;

- **Probabilité de panne de connexion** ;
- **Probabilité de perte**, c'est à dire la destruction d'un message.

caractéristiques **de sécurité**, le contrôle d'accès et l'authentification :

- **Protection** de connexion : empêchement d'utilisation pour les connexions sans autorisations ;
- **Authentification** de l'émetteur et du récepteur.

Dans notre approche, nous introduisons un modèle mathématique qui prend en compte ces caractéristiques de QoS. Pour chaque application utilisant le réseau du "datacenter", ce modèle associe un ensemble de caractéristiques QoS. L'optimisation consiste alors à choisir la caractéristique de QoS qui satisfait au mieux le critère de minimisation.

5.2.1.3 Echelle de temps et granularité de contrôle

L'échelle de temps considérée par notre approche est de l'ordre de quelques secondes à quelques minutes pour le calcul de la reconfiguration (en comparaison à plusieurs heures en cas de reconfiguration manuelle). L'optimisation induite peut se calculer sur une durée de plusieurs jours ou mois, c'est-à-dire que les effets de l'optimisation sont mesurables sur un plus long terme. Pour positionner notre approche, nous nous servons des critères de réalisation d'une architecture de QoS [87]. Ces critères déterminent son domaine d'application : son échelle, sa grandeur et ses limites. On distingue l'**échelle de temps**, la **granularité de contrôle** et la **localisation des mécanismes de contrôle** ainsi que des informations de contrôle correspondantes.

En premier lieu, pour l'**échelle de temps** de contrôle de QoS, il faut trouver un compromis entre la complexité de traitement et la précision du contrôle :

- **Paquets individuels** (1-100 μ) : Mécanismes de classification, contrôle de trafic, scheduling ;
- **Délai aller-retour** (1-100ms) : Mécanismes de contrôle de flux et de congestion ;
- **Sessions d'utilisateurs** (sec.-min.) : Mécanismes de réservation de ressources, de contrôle d'admission et d'optimisation d'énergie ;
- **Planification de réseau** (jours, mois etc.) : Optimisation de la capacité du réseau.

Pour notre approche, nous considérons donc que le calcul de l'optimisation lui-même doit s'effectuer à l'échelle *sessions d'utilisateurs*, car nous voulons appliquer les propriétés de "self-optimizing" pour l'optimisation d'énergie dans un réseau filaire.

En ce qui concerne la **granularité de contrôle**, elle peut se faire par micro-flux identifié (adresse IP souce et destination, port source et destination, protocole) ou par flux agrégés (par plusieurs classes de service, par destinations). Le traitement de micro-flux est en général trop complexe sur les noeuds à haut débit.

Notre approche traite les flux entre les applications du "datacenter" (couples d'applications émettrices/réceptrices) mais une simplification est possible en faisant l'agrégation des applications ayant le même couple premier routeur rencontré/ dernier routeur rencontré. Il s'agit donc d'une granularité de contrôle par flux agrégés.

Enfin, en ce qui concerne les **mécanismes de contrôle**, ils peuvent être localisés dans les systèmes terminaux (les machines clientes), en bordure du réseau ou dans le coeur du réseau (les noeuds haut débit). Les informations de contrôle correspondantes sont situées soit dans les en-têtes des paquets, soit dans les routeurs, soit dans un organisateur centralisé du réseau.

Pour notre approche, c'est un gestionnaire centralisé qui commande les mécanismes de contrôle du coeur du réseau. En effet, l'économie d'énergie est effectuée par l'extinction de certains éléments des noeuds du réseau, par exemple l'extinction de routeurs, de certains modules des routeurs ou de certains ports des routeurs. De plus, c'est ce gestionnaire centralisé qui calcule l'optimisation en étant au courant de l'état de l'ensemble des applications tournant sur le réseau du "datacenter".

5.2.1.4 Capacité de gestion QoS des applications

En ce qui concerne la capacité de gestion QoS des applications propriétaires tournant dans un "datacenter", nous considérons deux cas possibles : l'application a des capacités de changement de QoS intégrées ou l'application n'a pas de notion de QoS.

Application avec changement de QoS intégré : Si l'application a des mécanismes de gestion de la QoS en interne, elle a la capacité de modifier son fonctionnement pour émettre ou recevoir des trafics différents (compressions de données différentes). La modification de fonctionnement peut s'effectuer **manuellement** (demande de l'utilisateur ou d'un programme externe) ou **automatiquement** (elle sait choisir son fonctionnement interne pour la QoS par négociation entre le client et le serveur ou en fonction de l'environnement réseau local). Le fonctionnement peut aussi être **statique** (fixé au démarrage) ou **dynamique** (re-négociation périodique, changement en cours d'exécution).

Application sans notion de QoS : Pour une application sans notion de QoS, on ne peut pas changer son fonctionnement ni au démarrage, ni en cours d'exécution.

Pour notre approche, on ne considère que les applications avec changement de QoS intégré dont la modification de fonctionnement s'effectue manuellement. En effet, le changement de configuration QoS (c'est à dire la reconfiguration) sera initié par le gestionnaire autonomique en fonction des résultats de l'optimisation. Ceci permet de prendre des décisions de reconfiguration au niveau global, en tenant compte de toutes les applications.

5.2.2 Environnement considéré par l'approche

5.2.2.1 Le domaine DiffServ

Nous situons notre approche au niveau d'un domaine DiffServ[88]. Le modèle DiffServ est mis en oeuvre dans un domaine DiffServ (**domaine DS**) qui correspond à une zone contiguë de noeuds conforme à un modèle DiffServ et ayant une politique commune de qualité de service. C'est en général le cas pour un réseau dans un "datacenter". En effet, un "datacenter" est dans la grande majorité des cas géré par une seule entité administrative qui intègre le réseau dans un domaine DS appartenant à l'entité. Notre approche s'inscrit dans ce cadre avec un seul gestionnaire de "self-optimizing" et délimite le problème de "self-optimizing" au sein d'un seul domaine DS.

Une telle architecture QoS DiffServ s'articule autour de deux types de traitements : les traitements complexes et Les traitements simples.

- Les traitements complexes effectués dans les noeuds d'extrémité, appelés aussi **routeurs de bordure** [2]. Ces noeuds sont les premiers équipements réseau auxquels sont reliés les applications émettant les flux réseaux. Ils comportent un ensemble de modules appliquant des mécanismes complexes de QoS (classification, marquage, "metering", "shaping", "dropping"). Les **Routeurs de bordure** traitent du trafic entrant [3] ou sortant [4] par rapport au domaine DS. Dans le cas du trafic entrant, les paquets devront tous recevoir le marquage (aussi appelé la coloration) qui leur est destiné et, par la même occasion, se voir attribuer un traitement spécifique valable dans tout le domaine DS. Plus qu'une simple liaison, le routeur doit également permettre l'interfaçage avec les autres réseaux ne supportant pas DiffServ et se charger de l'éventuelle destruction ou réduction de la priorité pour les trafics transitant par le domaine DS. Dans notre approche et pour raison de simplification, nous considérons que le trafic transitant par le domaine DS est nul. Nous faisons donc abstraction des problématiques de collaboration entre domaines DS différents.
- Les traitements simples des noeuds intermédiaires, appelés aussi **routeurs de coeur** [5]. Ils ont des comportements plus statiques mais aussi plus simples que l'on regroupe sous le terme de PHB [6]. Les **Routeurs de coeur** doivent transmettre les paquets conformément aux paramètres appliqués dans DiffServ. Ils doivent s'appuyer sur le marquage et employer le mode de traitement de prioritisation adéquat.

5.2.2.2 Le routage du réseau

En général, les machines d'un "datacenter" sont regroupées en clusters. Un cluster peut éventuellement être physiquement connecté à plusieurs routeurs de bordure. Ceci veut dire que toutes les machines du cluster sont reliées à plusieurs routeurs et ont donc plusieurs cartes réseau. Deux cas de figure se présentent alors : soit la machine se base sur la table de routage (statique ou dynamique) pour choisir la carte réseau appropriée, soit elle fait une agrégation de plusieurs interfaces réseaux en une interface logique (procédé

2. edge routers
3. ingress traffic
4. egress traffic
5. core routers
6. Per-hop behaviour

appelé Channel bonding[89]). Notre approche considère pour le moment uniquement le cas des tables de routage dynamiques et ne prend pas en compte le cas du Channel Bonding.

De même, pour les routeurs de bordure et de coeur, le routage est soit statique, soit il dépend du protocole de routage utilisé à l'intérieur du domaine DS, c'est à dire qu'il dépend de l'IGP [7]. Les rôles d'un IGP sont :

- d'établir les routes optimales entre un point du réseau et toutes les destinations disponibles d'un domaine délimité ;
- d'éviter les boucles ;
- en cas de modification de topologie (déconnexion d'un lien physique, arrêt d'un routeur), d'assurer la convergence du réseau, c'est à dire le rétablissement de la connectivité optimale sans boucle dans les plus brefs délais.

On distingue généralement :

- les protocoles de routage à états de lien [8] qui établissent des tables de voisinage et emploient l'algorithme de Dijkstra pour calculer les meilleures routes. Deux exemples de tels protocoles sont *IS-IS* [9][90], *OSPF* [10][91] ;
- les protocoles de routage à vecteur de distance [11] : *RIP* [12], *IGRP* [13][92], *RIPv2* [14][93] ;
- les protocoles hybrides, qui ont des caractéristiques des deux premiers : *EIGRP* [15][94].

Notre approche prend en compte le cas des différents protocoles de routage. Nous donnons un premier exemple d'heuristique avec une table de routage dynamique générée par le protocole de routage OSPF. Le choix d'OSPF comme protocole de routage a pour conséquence de créer des tables de routage suivant une métrique définie sur les liens. Par exemple, le coût des routes peut être calculé suivant les capacités de tous les liens de la route. Un équilibrage de charge est effectué sur les routes ayant le même coût et les routes ayant un coût supérieur ne sont pas du tout utilisées. Il est alors possible d'éteindre certains liens ou routeurs non utilisés.

5.3 Modèle mathématique pour la gestion autonomique de l'énergie et de la qualité de service des réseaux filaires.

Dans cette section, nous introduisons la formalisation du modèle mathématique de "self-optimizing". En premier, la formalisation pour la partie réseau est introduite puis la formalisation pour la partie applicative.

7. Interior Gateway Protocol
8. link state protocols
9. Intermediate system to intermediate system
10. Open Shortest Path First
11. distance vector protocols
12. Routing Information Protocol
13. Interior Gateway Routing Protocol
14. Routing Information Protocol version 2
15. Enhanced Interior Gateway Routing Protocol

5.3.1 Formalisation pour la partie réseau

5.3.1.1 Noeuds et représentation du réseau par un graphe

Nous rappelons que l'ensemble des noeuds considérés appartiennent à un même domaine DS, représentant le domaine d'administration de la taille d'un réseau de "datacenter". La topologie du réseau est représentée sous forme d'un graphe orienté $G = \{N, E\}$ où **N** est l'ensemble des noeuds (Nodes) et **E** est l'ensemble des arrêtes ou arcs (Edges), c'est à dire un ensemble de couples (i, j) représentant le lien réseau entre le noeud i et le noeud j. Un noeud est soit une machine qui exécute une application (réceptrice ou émettrice de trafic), soit un routeur de bordure, soit un routeur de coeur.

$$\mathbf{N} = \{n_1, \dots n_i \dots, n_{n_N}\}, \mid \mathbf{N} \mid = n_N$$

$$\mathbf{E} = \{e_1, \dots e_h \dots, e_{n_E}\}, \mid \mathbf{E} \mid = n_E$$

Pour chaque arc e_h, le compteur h représente un couple (i, j), on peut donc écrire $e_{i,j}$ pour représenter un arc. Par abus, nous noterons parfois l'arc $e_(i, j)$ plutôt (i, j).

5.3.1.2 Ports, modules et châssis des routeurs

Pour chaque arc $e_{i,j}$, on associe une capacité $c^e_{i,j}$, une puissance électrique $p^e_{i,j}$ (puissance électrique utilisée par le lien de i vers j, c'est à dire la puissance électrique consommée par la carte réseau du noeud i uniquement) que nous appellerons par la suite port et une variable $z^e_{i,j}$ qui définit si le port partant de i vers j est allumé électriquement :

$$z^e_{i,j} = \begin{cases} 1 & \text{si l'arc } (i,j) \text{ est allumé électriquement,} \\ 0 & \text{autrement} \end{cases}$$

Et on peut exprimer la puissance totale consommée par les ports p^e_t comme :

$$p^e_t = \sum_{(i,j) \in \mathbf{E}} p^e_{i,j}.z^e_{i,j}$$

En général, les **ports** d'un routeur sont regroupés en **modules** insérables dans un **châssis**. Pour un noeud machine, on considère que le nombre de modules est égal à 1 (en général une machine a seulement quelques cartes réseaux).

Soit l'ensemble des modules pour un noeud i :

$$\mathbf{M_i} = \{m_1, \dots m_l \dots, m_{n_{M_i}}\}, \mid \mathbf{M_i} \mid = n_{M_i}$$

La fonction NodeModules donne l'ensemble des modules pour un noeud.

$$\text{NodeModules} : \mathbf{N} \rightarrow \mathbf{M}$$

$$\forall i \in \mathbf{N} : \mathbf{M_i} = NodeModules(i)$$

La fonction ModuleCards donne l'ensemble des ports (ou l'ensemble des arcs sortants) pour un module.

$$\text{ModuleCards} : \mathbf{M} \rightarrow \mathbf{E}$$

Soit l'ensemble des arcs $\mathbf{E_{i,l}}$ sortant des ports du module l du noeud i.

$$\forall i \in \mathbf{N}, \forall l \in [1, n_{M_i}] : \mathbf{E_{i,l}} = ModuleCards(M_i)$$

L'ensemble $\mathbf{E_i}$ des ports d'un noeud i peut donc s'écrire comme l'union de tous les ports de tous les modules du noeud :

$$\forall i \in \mathbf{N} : \mathbf{E_i} = \bigcup_{l \in [1, n_{M_i}]} \mathbf{E_{i,l}}$$

De plus, on associe des puissances électriques au module $p^m_{i,l}$ et au châssis p^c_i, respectivement la puissance électrique consommée par le module l du noeud i et la puissance du châssis du noeud i. Si tous les ports d'un module sont éteints, on peut se permettre d'éteindre le module. De même, si tous les modules d'un noeud sont éteints, on peut se permettre d'éteindre le chassis du noeud. On introduit donc une variable $z^m_{i,l}$ qui définit si le module l du noeud i est allumé et z^c_i qui définit si le châssis du noeud i est allumé électriquement.

La condition *Si tous les ports d'un module sont éteints, on peut éteindre le module* peut s'écrire sous forme d'une implication :

$$\forall i \in \mathbf{N}, \forall l \in [1, n_{M_i}] : \sum_{(i,j) \in \mathbf{E_{i,l}}} z^e_{i,j} = 0 \Rightarrow z^m_{i,l} = 0$$

Afin de conserver un critère linéaire on peut écrire cette implication comme un ensemble d'inégalités :

Contrainte 1 *Si tous les ports d'un module sont éteints, alors le module est éteint.*

$$\forall i \in \mathbf{N}, \forall l \in [1, n_{M_i}] : z^m_{i,l} - \sum_{j \in \mathbf{E_{i,l}}} z^e_{i,j} \leq 0$$

Contrainte 2 *Si un des ports du module est allumé, alors le module est allumé.*

$$\forall i \in \mathbf{N}, \forall l \in [1, n_{M_i}], j \in \mathbf{E_{i,l}} : z^e_{i,j} - z^m_{i,l} \leq 0$$

De même, la condition *Si tous les modules d'un noeud sont éteints, on peut se permettre d'éteindre le châssis du noeud* peut s'écrire avec le même type d'implication que précédemment :

$$\forall i \in \mathbf{N} : \sum_{(i,j) \in \mathbf{E_i}} z^m_{i,j} = 0 \Rightarrow z^c_i = 0$$

Là aussi pour garder un système linéaire, on transforme cette implication en un ensemble d'inégalités :

Contrainte 3 *Si tous les modules sont éteints, le châssis est éteint.*

$$\forall i \in \mathbf{N} : z^c_i - \sum_{l \in [1, n_{M_i}]} z^m_{i,l} \leq 0$$

Contrainte 4 *Si un des modules du châssis est allumé, le châssis est allumé.*

$$\forall i \in \mathbf{N}, l \in [1, n_{M_i}] : z^m_{i,l} - z^c_i \leq 0$$

Nous rajoutons aussi les contraintes suivantes :

Contrainte 5 *Un lien éteint dans un sens est éteint dans l'autre sens.*

$$\forall i, j \in \mathbf{N} : z^e_{i,j} - z^e_{j,i} = 0$$

5.3.1.3 Expression de la puissance globale

La puissance consommée par le réseau du domaine DS du "datacenter" s'écrit comme la somme pour tous les routeurs de la consommation du châssis, de ses modules et de ses ports :

$$P_{total} = \sum_{i \in \mathbf{N}} \{p^c_i.z^c_i + \sum_{l \in [1, n_{M_i}]} [p^m_{i,l}.z^m_{i,l} + \sum_{j \in \mathbf{E_{i,l}}} p^e_{i,j}.z^e_{i,j}]\}$$

Ce qui donne en regroupant par famille de composant :

$$P_{total} = \sum_{i \in \mathbf{N}} p^c_i.z^c_i + \sum_{i \in \mathbf{N}, l \in [1, n_{M_i}]} p^m_{i,l}.z^m_{i,l} + \sum_{i \in \mathbf{N}, j \in \mathbf{E_i}} p^e_{i,j}.z^e_{i,j}$$

5.3.2 Formalisation pour la partie application

5.3.2.1 L'ensemble des applications du "datacenter"

Soit l'ensemble des applications utilisant le réseau du domaine DS :

$$\mathbf{A} = \{a_1, \ldots a_k \ldots, a_{n_A}\}, | \mathbf{A} | = n_A$$

Pour générer du trafic réseau, il doit y avoir un émetteur de trafic (sender) et un récepteur de ce trafic (receiver). On considère une application comme un couple émetteur-récepteur. Nous pouvons noter a^s_k la partie émettrice de l'application et a^r_k la partie réceptrice. On considère qu'une partie émettrice n'envoie la majorité de son trafic que dans un sens (abstraction de la signalisation TCP de retour par exemple) et à une seule partie réceptrice (pas de broadcast, multicast etc) ; ceci afin de différencier les flux.

5.3.2.2 Les besoins en caractéristiques QoS des applications

Nous introduisons l'ensemble **NE** des besoins [16] en QoS possibles pour toutes les applications. Chaque élément de cet ensemble est lui-même constitué d'un ensemble de valeurs exprimant les caractéristiques de QoS demandées par l'application (débit, gigue, temps de réponse etc) qui sont regroupés dans un tuple. Ce dernier est constitué de valeurs, d'intervalle ou de toutes autres représentations permettant de caractériser un besoin élémentaire en terme de qualité de service. Il est à noter que chaque caractéristique QoS n'est présente qu'une fois dans un tuple.

16. NE pour *needs* en anglais

$$\mathbf{NE} = \{n_1, \dots n_s \dots, n_{n_{NE}}\}, \mid \mathbf{NE} \mid = n_{NE}$$

Pour chaque application a_k, on associe un ensemble $\mathbf{NE^k}$ qui permet de spécifier pour une application les différents besoins possibles pour cette application. L'ensemble $\mathbf{NE^k}$ est donc une partie de **NE**. En parallèle de cet ensemble, nous introduisons l'ensemble de même taille $\mathbf{B^k}$ de variables binaires.

$$\mathbf{NE^k} = \{n_1^k, \dots n_s^k \dots, n_{n_{NE^k}}^k\}, \mid \mathbf{NE^k} \mid = n_{NE^k}$$

$$\mathbf{B^k} = \{b_1^k, \dots b_n^k \dots, b_{n_{B^k}}^k\}, \mid \mathbf{B^k} \mid = n_{B^k}$$

De fait on a : $n_{NE^k} = n_{B^k}$
Soit $b_n^k \in \mathbf{B^k}$ cette variable binaire est définie comme suit :

$$b_n^k = \begin{cases} 1 & \text{si le besoin } n_s^k \ (s = n) \text{ est choisi pour l'application } a_k, \\ 0 & \text{autrement} \end{cases}$$

Pour une application, un seul besoin peut être choisi à un instant donné, ce qui s'écrit par la contrainte suivante :

Contrainte 6 *Un seul besoin est choisi par application.*

$$\forall k \in [1, n_A] : \sum_{n=1}^{n_{B^k}} b_n^k - 1 = 0$$

On suppose qu'il existe une fonction métrique M permettant de mesurer la qualité d'un besoin. Cette dernière permet de définir une relation d'ordre total notée $<$ entre les différents besoins d'une application. Afin de simplifier par la suite la notation, on supposera que les besoins sont rangés dans l'ensemble $\mathbf{NE^k}$ suivant l'ordre $<$ défini avec la métrique M (ceci est toujours réalisable à une permutation prêt sur les indices). Ce qui revient à pouvoir écrire :

$$M(n_1^k) < M(n_2^k) < \ldots < M(n_{n_{NE^k}}^k)$$

5.3.2.3 Expression de la perte de qualité de service

On définit une perte de qualité de service (Quality Loss) QL pour l'application a_k ayant choisi le besoin de qualité de service c (c'est à dire $b_c^k = 1$, tous les autres b_n^k étant égaux à zéro pour $n \neq c$) comme étant :

$$\forall k \in [1, n_A] : QL_{a_k} = M(n_{n_{NE_k}}^k) - \sum_{c=1}^{n_{B^k}} M(n_c^k).b_c^k$$

La perte de qualité de service totale pour le réseau du domaine DS du "datacenter" s'écrit donc :

$$QL_{Total} = \sum_{a_k \in \mathbf{A}} QL_{a_k}$$

5.3.2.4 Propagation des flots des applications

Pour une application a_k, on considère le noeud émetteur N_s^k (send) du trafic de a_k^s et le noeud récepteur N_r^k (receive) du trafic arrivant à a_k^r. On définit un flot $x_{i,j}^k$ comme étant le débit utilisé sur le lien (i,j) par l'application a_k.

Nous introduisons la fonction AF^k qui donne pour une application a_k et un besoin choisi n_n^k le débit moyen produit par l'application [17]. Notre approche suppose donc que le débit moyen produit par l'application en fonction de ses besoins est connu.

On a donc la contrainte suivante :

Contrainte 7 *Le flot sortant du noeud émetteur est égal au débit moyen produit par l'application et suit la route en fonction du protocole de routage.*

$$\forall k \in [1, n_A], j \in \mathbf{E} \text{ tels que } \exists e_{N_s^k,j} \in \mathbf{E} : x_{N_s^k,j}^k - \sum_{n=1}^{n_{B^k}} (AF(n_n^k).b_n^k) = 0$$

Nous rajoutons aussi les contraintes suivantes qui permettent le calcul de la propagation des flots de a_k^s à a_k^r :

Contrainte 8 *Pour tous les noeuds qui ne sont pas des machines, la différence entre les débits entrants et sortants est nulle*

$$\forall k \in [1, n_A], \forall i \in \mathbf{E} : \sum_{\substack{j \in \mathbf{E}, \exists e_{i,j} \\ j \neq N_s^k, N_r^k}} x_{i,j}^k - \sum_{\substack{u \in \mathbf{E}, \exists e_{u,i} \\ u \neq N_s^k, N_r^k}} x_{u,i}^k = 0$$

Contrainte 9 *Le débit qui sort de N_s^k rentre dans le noeud N_r^k :*

$$\forall k \in [1, n_A], \forall i \in \mathbf{E} \text{ tels que } \exists e_{N_s^k,i} \in \mathbf{E}, \forall l \in \mathbf{E} \text{ tels que } \exists e_{l,N_r^k} \in \mathbf{E} : x_{N_s^k,i}^k - x_{l,N_r^k}^k = 0$$

Les deux contraintes suivantes permettent de spécifier que les débits entrant dans N_s^k et sortant de N_r^k sont nuls :

Contrainte 10 *Ce qui sort (en débit) des machines réceptrices est nul.*

$$\forall k \in [1, n_A], \forall i \in \mathbf{E} \text{ tels que } \exists e_{i,N_s^k} \in \mathbf{E} : x_{i,N_s^k}^k = 0$$

Contrainte 11 *Ce qui rentre (en débit) dans les machines émettrices est nul.*

$$\forall k \in [1, n_A], \forall l \in \mathbf{E} \text{ tels que } \exists e_{N_r^k,l} \in \mathbf{E} : x_{N_r^k,l}^k = 0$$

Enfin, la minimisation doit respecter les capacités des liens en tenant compte des liens éteints :

Contrainte 12 *Les flots respectent les capacités des liens, s'ils sont éteints.*

$$\forall (i,j) \in \boldsymbol{E} : \sum_{k=1}^{n_A} (x_{i,j}^k - c_{i,j}^e . z_{i,j}^e) \leq 0$$

$$\forall (i,j) \in \boldsymbol{E}, \forall k \in [1, n_A] : -x_{i,j}^k \leq 0$$

17. En anglais nous choisissons le nom de la fonction **AF** pour désigner **Average Flow**

5.3.3 Formalisation du critère global de minimisation

De manière globale, le problème peut s'écrire comme la minimisation d'un critère, en parcourant les possibilités pour les différentes variables $z = [0; 1]$ (l'allumage/l'extinction des éléments qui constituent le réseau géré), $b = [0; 1]$ (le choix des besoins QoS des applications) et $x \in \mathbb{R}^+$ (la valeur des flots sur les liens du réseau) :

$$\underset{x\in\mathbb{R}^+;z=[0;1];b=[0;1]}{\text{Min}} \alpha.P_{total} + \beta.(1-\alpha).QL_{total}$$

Nous introduisons ici $\alpha \in [0; 1]$ qui représente le compromis entre la puissance totale du réseau géré et la perte de qualité de service engendrée. C'est l'administrateur du réseau du "datacenter" qui définit α.

Ce qui peut s'écrire sous la forme développée :

$$\underset{x^k_{i,j};z^c_i,z^m_{i,l},z^e_{i,j},b^k}{\text{Min}} \alpha.\{\sum_{i\in\mathbf{N}} p^c_i.z^c_i + \sum_{i\in\mathbf{N},l\in[1,n_{M_i}]} p^m_{i,l}.z^m_{i,l} + \sum_{i\in\mathbf{N},j\in\mathbf{E_i}} p^e_{i,j}.z^e_{i,j}\}+$$

$$\beta.(1-\alpha).\sum_{k=1}^{n_A}(M(n^k_{n_{NE_k}}).b^k_{n_{B^k}} - M(n^k_n).b^k_n)$$

Avec les contraintes définies plus haut (**Contraintes 1 à 13**).

Notons que α et $(1-\alpha)$ prennent en compte une éventuelle normalisation par β, pour ramener les deux critères sur une échelle comparable. Cette remise à l'échelle se fait par rapport aux bornes minimales et maximales de P_{Total} et QL_{Total}. Dans un premier temps et pour simplification, nous décidons de calculer β à l'aide d'une moyenne arithmétique sur ces bornes (il serait aussi envisageable d'utiliser une moyenne géométrique ou une médiane) :

$$\beta = (P_{min} + P_{max})/(QL_{min} + QL_{max})$$

Pour le calcul de P_{max}, nous mettons tous les $z = 1$ et calculons P_{total}. Pour le calcul de P_{min}, nous faisons une première minimisation du critère global uniquement sur les variables z en mettant $\alpha = 1$ (la partie concernant la qualité de service est ignorée). De même, pour le calcul de QL_{max} nous calculons QL_{Total} avec tous les $b^k_{n_{B_k}} = 1$ et nous minimisons le critère global avec $\alpha = 1$ pour QL_{min}.

5.3.4 Validation et expériences

5.3.4.1 Problème d'optimisation

Pour la validation de notre approche, nous décidons de comparer des résultats dans le cas d'une recherche de solution optimale et dans le cas d'utilisation d'un protocole de routage qui rajoute des contraintes supplémentaires sur les flots. En effet, une solution optimale maximise les flots et l'équilibrage de charge sur tous les liens du réseau. En revanche, elle n'est pas réaliste car c'est un protocole de routage spécifique qui est en général utilisé pour calculer les tables de routage du réseau dynamiquement.

Nous définissons tout d'abord la fonction *RoutingNodes* pour le réseau, qui prend un noeud de départ et un noeud d'arrivée et crée l'ensemble des noeuds de tous les chemins possibles calculés par la fonction de routage du réseau (OSPF, RIP, RIPv2 ...). Cette fonction prend en compte l'extinction et l'allumage des équipements réseau (les $z^X_{i,j} = 0, 1$).

$$\text{RoutingNodes} : \mathbf{NxN} \rightarrow \mathbf{N}$$

On définit l'ensemble $\mathbf{RN^k}$ correspondant aux noeuds calculés tels que :

$$\mathbf{RN^k} = \text{RoutingNodes}(N^k_s, N^k_r)$$

On définit aussi, pour un noeud i, les ensembles $\mathbf{RN^{k,i}_{input}}$ et $\mathbf{RN^{k,i}_{output}}$ suivants :

$$\mathbf{RN^{k,i}_{input}} : \forall j \in \mathbf{N}, \exists e_{i,j} \in \mathbf{E}$$

$$\mathbf{RN^{k,i}_{output}} : \forall j \in \mathbf{N}, \exists e_{j,i} \in \mathbf{E}$$

En fonction du routage du réseau, les flots des applications peuvent être divisés par des mécanismes d'équilibrage de charge. Par exemple, dans le cas d'utilisation du protocole OSPF, il faut rajouter des contraintes qui spécifient que les flots sont divisés équitablement sur les chemins de même coût (le coût étant calculé par la métrique OSPF, en général la capacité des liens). Ces contraintes d'équilibrage de charge sont donc spécifiques au protocole choisi et nous choisissons ici de les exprimer pour le protocole OSPF, qui est le plus largement répandu.

Contrainte 13 *Les flots respectent les contraintes d'équilibrage de charge en fonction du protocole de routage choisi. Pour OSPF ces contraintes définissent une division des flots équitable entre les chemins de même coût :*

$$\forall k \in [1, n_A]; \forall i \in \mathbf{RN^k}, Card(\mathbf{RN^{k,i}_{output}}) > 1, i \neq N^k_s; \forall j \in \mathbf{RN^{k,i}_{output}} :$$

$$x^k_{i,j} = \frac{1}{Card(\mathbf{RN^{k,i}_{output}})} \sum_{l \in \mathbf{RN^{k,i}_{input}}} x^k_{l,i}$$

Il est important de noter que pour calculer la solution optimale il faut désactiver cette contrainte. Dans ce cas, la solution optimale représente une maximisation des flots, difficilement atteignable dans la réalité, même avec les protocoles de routage dont les coûts des liens sont calculés de manière dynamique [95]. Dans nos expériences, nous comparons

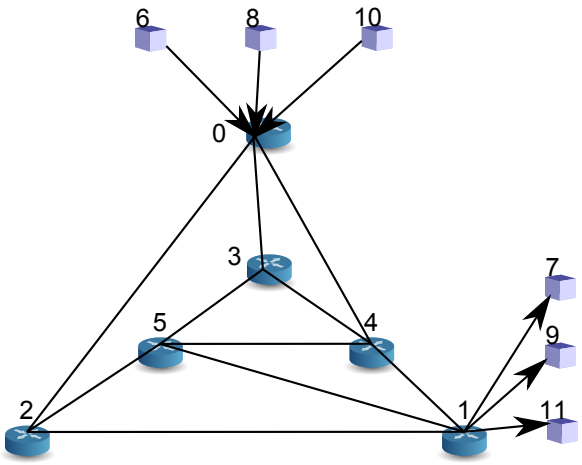


FIGURE 5.1 – Topologie de l'architecture Grid'MIP

la solution optimale avec la solution utilisant le protocole de routage OSPF, largement répandu à l'heure actuelle.

Une limite apparait si la **Contrainte 13** est utilisée. En effet, cette fonction dépend de l'allumage et de l'extinction des châssis, des modules et des ports car la fonction *RoutingNodes* prend en compte l'extinction et l'allumage des équipements réseau. Ceci implique que pendant la minimisation, à chaque changement de variable z, la fonction de routage change. La **Contrainte 13** devient alors dépendante d'une variable à minimiser et il faut introduire une heuristique pour parvenir à calculer une solution.

Dans les parties suivantes, nous présentons le contexte de résolution et nous comparons la solution optimale et la solution obtenue avec l'heuristique pour le cas d'utilisation d'OSPF.

5.3.4.2 Contexte de résolution

Pour valider notre approche, nous choisissons d'utiliser la topologie de Grid'MIP (voir section 2.3.3) décrite par la figure 5.1. Les trois routeurs de coeurs sont numérotés 0, 1, 2 et disposent de deux modules : le premier de 48 ports Gigabit-ethernet (chaque lien connecté a une capacité $c_{i,j} = 1000$, soit 1000 Mbit/s) et le deuxième de quatre ports Fibre Channel 10 Gigabit (chaque lien connecté a une capacité $c_{i,j} = 10000$, soit 10000 Mbit/s). Le premier module est relié aux machines qui exécutent les applications et le deuxième module aux autres routeurs. Les trois routeurs de coeur sont numérotés 3, 4, 5 et disposent chacun du module Fibre Channel avec quatre ports 10 Gigabit. Pour l'initialisation des constantes de puissance ($p_i^c, p_{i,l}^m, p_{i,j}^e$) nous utilisons des valeurs mesurées sur la plateforme Grid'MIP à l'aide de wattmètres (Annexe D).

En ce qui concerne le placement des applications, nous avons décidé d'utiliser une disposition simple qui permette de mettre en valeur la différence d'allumage des équipements réseaux dans deux cas précis : le cas optimal et le cas d'utilisation du protocole de routage OSPF. C'est pour cela que pour toutes les applications, nous avons choisi de connecter toutes les machines exécutant la partie émettrice sur le routeur de bordure 0 et les machines exécutant la partie réceptrice sur le routeur de bordure 1 ($\forall k \in [1, n_A] : N_s^k = 0, N_s^k = 1$). Ainsi, la machine 6 émet du trafic à la machine 7, la

machine 8 à la machine 9, la machine 10 à la machine 11 etc. En considérant le cas optimal, pour aller du routeur 0 au routeur 1, il existe trois chemins possibles : 0-2-1, 0-4-1 et 0-3-5-1. Tous les liens des trois chemins étant différents et ayant une capacité de 10 Gbit/s, le total du débit pouvant transiter du routeur 0 vers le routeur 1 est de 30 Gbit/s. En considérant le cas du routage OSPF avec la capacité des liens comme métrique pour le coût, il ne reste que deux chemins possibles : 0-2-1 et 0-4-1, le total du débit pouvant transiter est alors de 20 Gbit/s. Enfin, pour les constantes $AF(n_n^k)$ nous considérons toutes les applications uniformes pour une meilleure lisibilité des résultats. Nous associons à ces applications cinq besoins de QoS ($\mid \mathbf{B^k} \mid = n_{B^k} = 5$) et pour raison de simplification nous considérons que la métrique M définissant la qualité de chaque besoin pour ces applications est égale au débit moyen résultant du besoin choisi, c'est à dire $M(n_n^k) = AF(n_n^k)$. Les cinq débits moyens résultants des cinq besoins sont fixés en Mbit/s à $AF(n_0^k) = 200$, $AF(n_1^k) = 400$, $AF(n_2^k) = 600$, $AF(n_3^k) = 800$ et $AF(n_4^k) = 1000$. Enfin, pour nos expériences, nous faisons varier le nombre d'applications entre 1 et 60 ($n_A \in [1..60]$) et α par pas de 0,5.

5.3.4.3 Résolution dans le cas optimal

Les variables du modèle d'optimisation n'étant jamais multipliées entre elles, il s'agit d'un problème de programmation linéaire (**PL**). Il est en revanche nécessaire d'utiliser des variables discrètes dans la modélisation du problème, par exemple pour les valeurs associées aux variables binaires du problème. Dans ce cas particulier, le modèle rajoute des contraintes dites d'intégrité, on parle alors de programmation linéaire en nombres entiers (**PLNE**), nettement plus complexe à résoudre que les **PL** à variables continues.

Pour le premier calcul optimal nous utilisons JOpt[96], un outil java open source qui fournit un mécanisme d'encapsulation de **PL**. JOpt permet de créer des objets java pour définir des **PL** de manière générique, sans se soucier du solveur linéaire utilisé. En effet, JOpt n'est pas un solveur mais une surcouche d'accès à distance aux solveurs tels que CPLEX[97], LPSolve[98] ou Hooks[99]. JOpt permet aussi de faire de l'équilibrage de charge dans le cas de la résolution de problèmes en parallèle sur plusieurs solveurs. En annexes B.1 et B.2 nous donnons les étapes nécessaires pour la construction d'un **PL** avec JOpt.

Pour la résolution du critère, quatre étapes sont nécessaires pour le calcul de la solution d'optimisation :

Etape 1 Initialisation : La première étape initialise les entrées du problème : le graphe représentant le réseau (châssis, modules, ports, liens), le placement des applications sur le graphe (pour chaque a_k création du noeud émetteur et récepteur, création des besoins et calcul des débits moyens) et les constantes.

Etape 2 Préparation : La deuxième étape prépare les critères pour l'objectif et les contraintes. Il s'agit du calcul des coefficients et de la construction des objets JOpt en respectant le nommage : les variables, les termes, les critères, les contraintes et l'objectif.

Etape 3 Normalisation : La troisième étape permet de calculer la normalisation à effectuer sur l'objectif. Elle fait deux appels au solveur linéaire pour le calcul de P_{min} et de QL_{min} et un calcul simple de P_{max} et de QL_{max}.

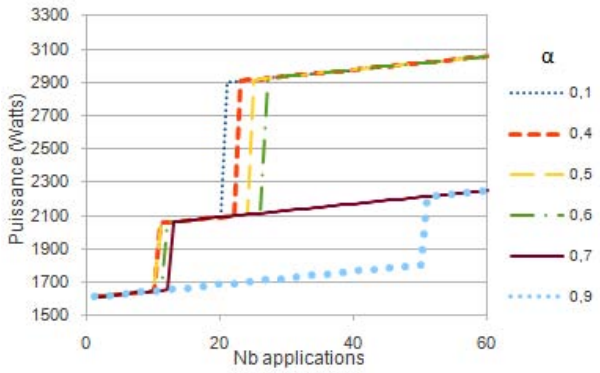


FIGURE 5.2 – Puissance en fonction du nombre d'applications et d'α dans le cas de self-optimization optimal

Etape 4 Minimisation : La quatrième étape consiste à lancer la minimisation sur le solveur distant et récupérer les solutions.

Le calcul des coefficients et la construction des objets JOpt (variables, termes, critères, contraintes et l'objectif) s'effectue au niveau d'un client JOpt qui a été codé en java et qui transfère ensuite ces objets au solveur linéaire distant. Dans notre cas, nous utilisons le solveur CPLEX sur un serveur distant avec quatre processeurs dual-core Intel Xeon à 3,20GHz. Dans nos expériences nous calculons le temps **preparation_time** qu'il faut pour calculer tous les coefficients et préparer toutes les contraintes et l'objectif et le temps **network_time** qu'il faut pour transmettre les objets JOpt et récupérer les résultats. L'algorithme en annexe 3 donne un exemple de création des objets JOpt pour la **Contrainte 2**.

Les figures 5.3.4.3 et 5.3.4.4 représentent la puissance du réseau en fonction du nombre d'applications et de certaines valeurs d'α sélectionnées.

Dans le cas optimal (Figure 5.3.4.3), entre 0 et environ 10 applications pour $\alpha < 0,8$ il n'y a qu'un chemin allumé (sur la figure 5.1 le chemin 0-2-1 ou 0-4-1) et la consommation augmente légèrement entre 1600 et 1700 Watts en fonction du nombre de ports (environ 4 ports par applications : 2 pour l'émetteur et 2 pour le récepteur, donc environ 4 Watts par application). Pour $\alpha = 0,9$ la consommation augmente ainsi jusqu'à 50 applications. Entre 11 et 13 applications pour $\alpha < 0,8$, on peut constater un saut de puissance indiquant qu'un routeur (châssis, modules et ports concernés) supplémentaire est allumé. Il faut attendre 50 applications pour observer ce saut pour $\alpha = 0,9$. En effet, au delà de 50 applications, le débit moyen résultant des besoins minimums étant fixé à 200 Mbit/s, la capacité du seul chemin allumé à 10 Gbit/s est dépassée ce qui force l'allumage d'un deuxième chemin.

Nous remarquons donc que plus α augmente, plus la puissance est prise en compte dans la minimisation, ce qui retarde l'allumage de nouveaux routeurs au détriment de la qualité de service, comme on peut le voir sur la figure 5.3.4.3. Entre ce premier saut et le deuxième après 20 applications, on a donc les routeurs 0,1,2 et 4 allumés et les chemins 0-2-1 et 0-4-1 utilisés. A partir de 20 applications et pour $\alpha < 0,7$, on remarque un deuxième saut en puissance plus important que le premier car il concerne l'allumage des routeurs 3 et 5. Il s'étale de 21 applications pour $\alpha = 0,1$ à 27 applications pour $\alpha = 0,6$.

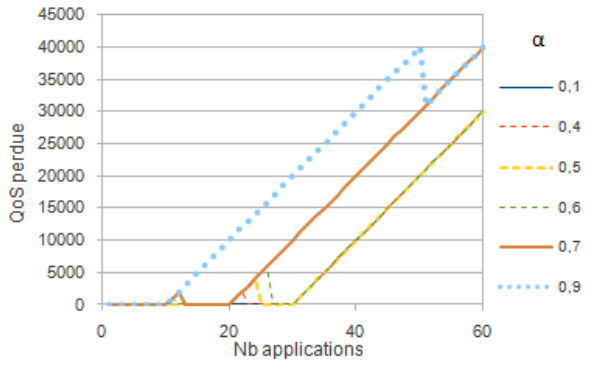


FIGURE 5.3 – Qualité de service perdue en fonction du nombre d'applications et d'α dans le cas de self-optimization optimal

On remarque aussi expérimentalement que pour $\alpha = 0,7$ ce saut n'a pas lieu jusqu'à 60 applications. Avec les routeurs 3 et 5 allumés, le réseau atteint une consommation de plus de 2900 Watts et les trois chemins sont utilisés (0-4-1, 0-2-1 et 0-3-5-1).

En ce qui concerne la qualité de service (figures 5.3.4.3 et 5.3.4.4), nous remarquons globalement que plus α augmente, plus il y a de perte en qualité de service. Nous remarquons aussi que les différents seuils de démarrage de la perte de qualité de service coïncident avec les sauts en puissance des figures 5.3.4.3 et 5.3.4.4. Par exemple pour $\alpha = 0,9$, c'est la puissance qui prédomine dans l'objectif de minimisation. Dans ce cas on voit que la perte en qualité de service commence à partir de 11 applications. En effet, jusqu'à 10 applications, le débit moyen total pour les 10 applications ne peut pas excéder la capacité du seul chemin allumé (soit 0-2-1, soit 0-4-1). Les 10 applications utilisent donc les besoins dont la qualité mesurée par M est maximum et qui ont un débit moyen résultant $AF(n_4^k) = 1000$ Mbit/s. A partir de 11 applications, la qualité de service est alors dégradée pour au moins une application. Ce phénomène est bien visible juste avant l'allumage de nouveaux routeurs. On peut en effet observer une légère augmentation de la perte en qualité de service, visible notamment entre 20 et 27 applications pour $\alpha < 0,7$ dans le cas optimal (figure 5.3.4.3).

5.3.4.4 Résolution dans le cas de l'utilisation de l'heuristique

L'utilisation d'une heuristique quand la fonction de routage du réseau est OSPF est nécessaire, car pendant la minimisation, à chaque changement de variable z, la fonction de routage change. Comme la **Contrainte 13** devient dépendante d'une variable à minimiser, nous introduisons l'heuristique proposée par l'algorithme 1. L'idée consiste à calculer une première solution qui utilise tous les chemins OSPF pour toutes les applications. Ceci est fait au début de l'heuristique à l'aide de l'appel à l'algorithme donné en annexe 4. Cet algorithme rajoute des contraintes qui forcent les flots qui utilisent des liens sur des chemins non-OSPF à être nuls. Avant de démarrer la boucle d'itérations de l'heuristique, une première solution (stockée dans la variable **meilleure_solution**) est calculée en prenant en compte ces contraintes. Une variable d'historique de parcours de l'heuristique **solutions_explorées** permet de mémoriser une association entre l'ensemble des solutions pour les variables et l'ensemble des contraintes rajoutées par l'heuristique afin

Algorithm 1 Heuristique utilisée avec OSPF comme fonction de routage

```
Initialiser les constantes, contraintes et exécuter l'algorithme 4
Résoudre P_min (LP avec α = 1), QL_min (LP avec α = 0)
Résoudre directement P_max, QL_max
β = (P_min + P_max)/(QL_min + QL_max)
Créer Solution meilleure_solution ← Résoudre LP
Créer Liste solutions_explorées ← meilleure_solution
while nb_iterations < max_iter & objectif_bouge do
  N^trie ← trier N par nombre inverse d'applications a_k les utilisant
  for i ∈ N^trie do
    M_i^trie ← trier M_i par nombre inverse d'applications a_k les utilisant
    for l ∈ M_i^trie do
      E_i,l^trie ← trier E_i,l par nombre inverse d'applications a_k les utilisant
      for e_i,j ∈ E_i,l^trie do
        exécuter l'algorithme 2 avec z^e_i,j = 0
        nb_iterations ← nb_iterations + 1
      end for
      exécuter l'algorithme 2 avec z^m_i,l = 0
      nb_iterations ← nb_iterations + 1
    end for
    exécuter l'algorithme 2 avec z^c_i = 0
    nb_iterations ← nb_iterations + 1
  end for
end while
return meilleure_solution
```

de ne pas recalculer une solution sur une topologie déjà explorée.

L'heuristique est arrêtée si le nombre d'itérations maximum est atteint ($nb_iterations < max_iter$) ou si l'objectif de la meilleure solution n'a pas bougé depuis un certain nombre d'itérations ($objectif_bouge$). Nous trions les châssis (Noeuds), les modules et les ports par nombre inverse d'applications les utilisant, c'est à dire du plus petit nombre de flux au plus grand nombre de flux les utilisant. Nous essayons d'éteindre en premier les ports, puis en second les modules et en dernier les châssis (Noeuds), en respectant pour chacun l'ordre du tri. En effet, cela minimise le nombre d'applications impactées par l'extinction de l'équipement concerné. A chaque extinction nous vérifions à l'aide de l'algorithme 2 s'il existe au moins un chemin OSPF pour toutes les applications et s'il existe une solution. Cet algorithme 2 est le même pour l'extinction des ports ($z^e_{i,j} = 0$), des modules ($z^m_{i,l} = 0$) ou des châssis ($z^c_i = 0$) et nous le montrons de manière généralisée avec la variable z. Si cet algorithmique trouve qu'il existe au moins un chemin OSPF pour toutes les applications et que la solution trouvée est meilleure, alors elle est stockée dans **meilleure_solution**, sinon nous continuons d'explorer les solutions pendant max_essais itérations pour essayer d'en trouver une meilleure. A chaque fois qu'une meilleure solution est trouvée, $max_essais = 0$. Si nous arrivons à max_essais sans trouver de meilleure solution alors nous arrêtons l'heuristique ($objectif_bouge =$ **false**).

Dans le cas de l'utilisation de l'heuristique (figures 5.3.4.4 et 5.3.4.4) nous remarquons

Algorithm 2 Fonction de vérification pour l'heuristique

Require: z comme entrée ; **solutions_explorées** & **meilleure_solution** comme entrée/sortie
  if $z = 0 \notin$ **solutions_explorées then**
    if $\forall k \in [1, nA]$ & $z = 0$ & $\exists chemin_OSPF(N_s^k, N_r^k)$ **then**
      Solution **s** $\leftarrow$ exécuter l'algorithme 4 et résoudre **LP** avec **meilleure_solution** et $z = 0$
      if $\exists$**s then**
        solutions_explorées $\leftarrow$ **s**
        if s est meilleure **then**
          meilleure_solutions $\leftarrow$ **s**
          $nb_tries \leftarrow 0$
        else if $nb_essais < max_essais$ **then**
          $nb_essais \leftarrow nb_essais + 1$
          solutions_explorées $\leftarrow$ **s**
        else
          $objectif_bouge =$ **false**
        end if
      else
        solutions_explorées $\leftarrow$ **s**
        arrêter la boucle for
      end if
    else
      solutions_explorées $\leftarrow$ **s**
      arrêter la boucle for
    end if
  end if

que les résultats suivent ceux du cas optimal mais sans jamais allumer les routeurs 3 et 5. Comme expliqué dans le contexte de résolution (section 5.3.4.2), dans le cas d'utilisation de la fonction de routage OSPF, le chemin 0-3-5-1 ne peut pas être utilisé pour l'équilibrage de charge. En effet, en utilisant comme métrique de calcul OSPF les débits des liens (tous égaux ici) le coût du chemin 0-3-5-1 est plus important et il ne sera donc jamais utilisé pour les routes allant du routeur 0 au routeur 1.

Pour ce qui est de la qualité de service, elle est globalement plus dégradée que dans le cas optimal. Ceci s'explique par le fait que la troisième route 0-3-5-1, contrairement au cas optimal, n'est jamais utilisée. Le débit pouvant transiter par le réseau étant moindre, la qualité de service s'en trouve plus dégradée.

5.3.5 Comparaison du cas optimal et de l'utilisation de l'heuristique

Les figures 5.3.5 et 5.3.5 représentent les temps de résolution du critère respectivement dans le cas optimal et dans le cas de l'utilisation de l'heuristique. Nous avons fait varier le nombre d'applications jusqu'à 100 pour faire apparaître des temps de calculs plus

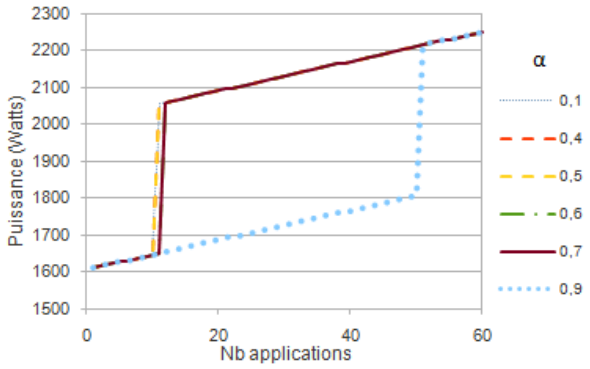


FIGURE 5.4 – Puissance en fonction du nombre d'applications et d'α dans le cas de routage OSPF avec utilisation de l'heuristique

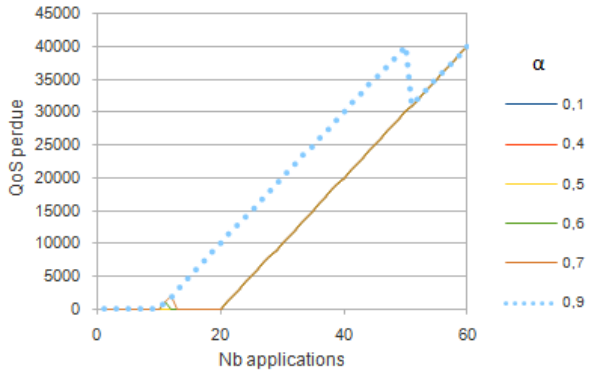


FIGURE 5.5 – Qualité de service perdue en fonction du nombre d'applications et d'α dans le cas de routage OSPF avec utilisation de l'heuristique

importants dans le cas de saturation du réseau. Ces temps sont décomposés en trois :

- Le temps de préparation : calcul de l'objectif, des variables, des termes et des contraintes JOpt (en forme développée).
- Le temps de transfert de l'ensemble des paramètres d'optimisation au serveur cplex par le réseau et le temps de réception des résultats.
- Le temps de calcul du critère par le serveur cplex.

Chaque résolution fait trois appels au solveur cplex distant. En effet, deux appels sont nécessaires au calcul de β (pour le calcul de la puissance minimale P_{min} et de la perte de qualité de service minimale QL_{min}) et un appel pour la résolution du critère. Les temps de préparation et de calcul englobent les temps pour P_{min}, P_{max}, QL_{min}, QL_{max} et pour le problème final. Les temps de transfert réseau englobent les temps pour les trois appels distants à CPLEX.

Nous remarquons que les temps de préparation et de calcul sont sensiblement les mêmes dans le cas optimal et de l'utilisation de l'heuristique. Ces temps restent aussi presque confondus et varient entre une vingtaine de millisecondes pour une application et une seconde pour 80 applications. Au delà de 80 applications, ces temps peuvent monter jusqu'à 4 secondes, ce qui s'explique par le fait qu'il est plus difficile de trouver une solution car tous les liens du réseau considérés dans l'expérience arrivent à saturation. Dans le cas de 100 applications il n'y a pas de solution possible ce qui fait redescendre

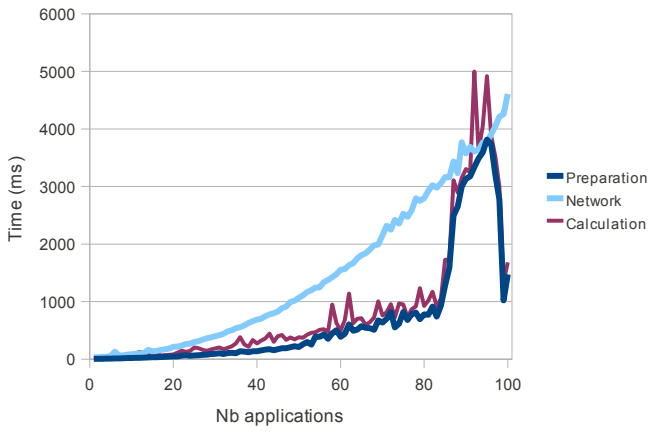


FIGURE 5.6 – Temps de résolution pour le cas optimal

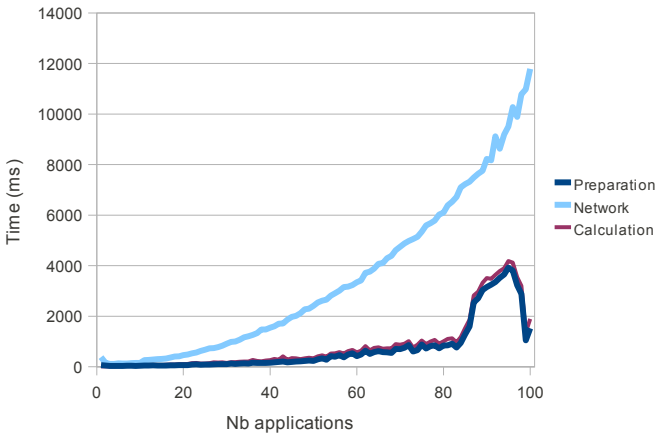


FIGURE 5.7 – Temps de résolution pour le cas de l'heuristique

ces temps à environ une seconde.

Pour ce qui est du temps de transfert des données JOpt au serveur CPLEX, nous remarquons que dans le cas de l'utilisation de l'heuristique, il est plus important que dans le cas optimal. En effet, l'heuristique rajoute des contraintes supplémentaires qui sont liées à la fonction de routage du réseau. Le fait de rajouter ces contraintes augmente le nombre de variables et de termes à transmettre au serveur pour la résolution. Dans le cas optimal, ce temps varie exponentiellement de 360 ms pour une application à environ 12 secondes pour 100 applications.

Compte tenu de ces résultats, nous pouvons conclure que l'utilisation de l'heuristique n'engendre pas un surplus de temps de calcul ou de préparation des contraintes, mais par contre le temps de transfert des contraintes rajoutées par l'heuristique est multiplié par 2,6. Globalement, ces temps restent néanmoins raisonnables dans le cas de la reconfiguration dynamique de matériels réseaux comme des routeurs. En effet, lors de nos tests, l'arrêt et le démarrage des routeurs peut prendre jusqu'à plus de six minutes (Voir l'annexe D). Une optimisation de réseau se planifie donc plutôt avec une granularité de l'ordre de l'heure, ce qui rend les temps de résolution de l'optimisation négligeables.

5.4 Conclusions du chapitre "self-optimizing"

Les administrateurs humains ne peuvent plus faire face à l'augmentation de la complexité de gestion de l'infrastructure et des applications déployées dans les datacenters. Que se soit au niveau matériel ou au niveau logiciel, le processus d'optimisation est une tâche délicate et coûteuse en temps. Dans ce chapitre, nous avons introduit la propriété de "self-optimizing" au niveau matériel en l'appliquant à l'optimisation des coûts énergétiques des datacenters.

Nous avons introduit une approche permettant de décrire un compromis entre, d'une part, la consommation électrique de l'infrastructure réseau et, d'autre part, la dégradation de la qualité de service des applications utilisant ce réseau. Le fait de pouvoir contrôler la reconfiguration dynamique à deux niveaux : au niveau applicatif en reconfigurant dynamiquement des profils de qualité de service et au niveau du matériel réseau avec l'extinction et l'allumage de liens, de modules ou de châssis, permet d'avoir une gestion globale et centralisée du datacenter. En effet, l'utilisation d'un gestionnaire autonomique centralisé permet à l'administrateur de jouer sur le coût énergétique ou sur la performance globale à l'aide d'un seul paramètre contrôlant la politique de haut niveau.

En ce qui concerne les limites de notre approche, on suppose qu'il existe une relation d'ordre entre les besoins (à une permutation prêt) en utilisant la fonction M. Cet ordre n'est pas à confondre avec la qualité d'expérience utilisateur ou "Quality of Experience" (QoE)[100]. Par exemple, pour une application de streaming vidéo, un serveur peut diffuser une vidéo en définition standard (SD) ou en haute définition (HD). Du point de vue serveur, la QoS fournie par une diffusion HD est supérieure à celle d'une diffusion SD. Par contre, du point de vue client, la QoE du service HD peut être inférieure à la QoE du service SD (le débit trop important en HD fait que le son ou l'image est hachée).

Dans certains cas, la relation d'ordre par la fonction métrique M est difficile à établir car elle dépend de la QoE. Par exemple, pour une communication voix sur IP (VoIP), un besoin favorisant uniquement le délai de transmission peut dégrader le taux de perte. Dans ce cas la communication sera hâchée (mais peut-être compréhensible). Un autre besoin favorisant uniquement le taux de perte peut dégrader le délai de transmission. Dans ce cas les réponses de l'interlocuteur mettent du temps à venir mais le son est de bonne qualité (cas bien connu des interview télé avec retransmission satellite). Dépendant de l'usage, la relation d'ordre entre ces deux besoins s'inverse. Une solution consisterait à créer un seul besoin avec les deux caractéristiques pondérées par l'utilisateur en fonction de l'usage.

Chapitre 6

Auto-protection - "Self-protecting"

Sommaire

6.1 Introduction de la problématique 119
6.1.1 Cohérence, stabilité et contraintes ACID 120
6.1.2 Règles de cohérence internes à TUNe 121
6.2 Solutions de récupération de la cohérence 122
6.2.1 Vérification des liaisons utilisées par les SWD et création automatique de liaisons en cours d'exécution d'un PDD 123
6.2.2 Vérification de cohérence des créations automatiques de liaisons et mécanisme de création automatique d'instances en cours d'exécution d'un PDD 125
6.2.3 Interruption automatique en cours d'exécution d'un PDD . . . 127
6.2.4 Vérifications de cohérences finales et procédures de résolution d'incohérences ou d'annulation 129
6.2.5 Vue globale de la gestion des cohérences 130
6.3 Mise en oeuvre du "self-protecting" pour l'architecture DIET 132
6.3.1 Présentation du cadre de mise en oeuvre 132
6.4 Conclusion du chapitre "self-protecting" 136
6.4.1 Contributions au "self-protecting" des systèmes autonomiques . 136
6.4.2 Limites de notre approche 137

6.1 Introduction de la problématique

La propriété de "self-protecting" permet à un système d'assurer sa survie en se protégeant de deux manières : grâce à une protection interne qui évite les phénomènes d'explosions, ainsi qu'avec une protection externe face aux attaques malicieuses. La protection interne apporte une cohérence et une stabilité globale au système à tout instant. La protection externe, quant à elle, fait face à des événements dit exogènes comme les attaques malicieuses, les infiltrations et autres infections par des virus.

D'après Chess D.M. [101], pour satisfaire la propriété de "self-protecting" les systèmes autonomiques doivent être capables de contrôler la sécurité du système à plusieurs niveaux. D'abord la sécurité doit être assurée pour se protéger contre les attaques de

manière pro-active. Ensuite, ces systèmes doivent pouvoir éventuellement se restaurer après une attaque. Enfin, la sécurité doit être assurée au sein même de tous les autres aspects de reconfiguration engendrés par le rajout des quatre propriétés "self-*". En effet, les systèmes autonomiques doivent rester cohérents pour toutes les configurations qu'ils peuvent atteindre. Aussi, ces systèmes doivent rester dans un état stable pour toutes les optimisations éventuellement déclenchées en cours d'exécution. D'après ce même auteur, la propriété de "self-protecting" n'est souvent réduite qu'à la protection externe contre les virus et les attaques malicieuses. Les principales contributions sont donc celles retrouvées en sécurité générale informatique (détection d'intrusion et d'attaque et parades pour palier à ce genre d'attaques). Selon ce même auteur, l'aspect de "self-protecting" interne est quant à lui toujours mis à l'écart.

Dans notre approche nous ne nous penchons que sur la problématique de "self-protecting" interne, c'est-à-dire du maintien de la cohérence et de la stabilité du système géré à tout instant. Nous introduisons en premier le concept de cohérence et de stabilité, puis les règles de cohérence internes dans le cas du "self-protecting" avec TUNe.

6.1.1 Cohérence, stabilité et contraintes ACID

Pour un système informatique, un **état cohérent** est défini comme une organisation logique entre les différents éléments du système. Nous définissons un état stable comme un état dans lequel les éléments ont une variation de certains de leurs paramètres (par exemple, la charge CPU) qui se stabilise autour d'une valeur désirée.

Le concept de cohérence est en général utilisé en bases de données et notamment dans le domaine des transactions [102]. Une transaction est définie comme une suite d'opérations sur les données, ce qui modifie l'état de la base de données. En particulier, elle doit respecter les quatre contraintes suivantes dites **ACID** [103] :

- **atomique** : la suite d'opérations est indivisible, en cas d'échec en cours d'une des opérations, la suite d'opérations doit être complètement annulée (rollback) quel que soit le nombre d'opérations déjà bien réalisées.
- **cohérente** : le contenu de la base de données à la fin de la transaction doit être cohérent sans pour autant que chaque opération durant la transaction donne un contenu cohérent. Un contenu final incohérent entraînera l'échec et l'annulation de toutes les opérations de la transaction.
- **isolée** : lorsque deux transactions `A` et `B` sont exécutées en même temps, les modifications effectuées par `A` ne sont ni visibles par `B`, ni modifiables par `B` tant que la transaction `A` n'est pas terminée et validée (*commit*).
- **durable** : d'un point de vue technique, une transaction terminée ne peut pas être remise en cause, annulée ou recouverte. Lorsque deux transactions sont exécutées en même temps, le résultat de la première transaction ne pourra pas être recouvert par la deuxième. Toute tentative de recouvrement entraînera l'annulation des opérations de la transaction fautive.

L'aspect fondamental de la cohérence repose sur la définition au préalable de **règles de cohérence**. Ces différentes règles permettent de spécifier ce que nous pouvons appeler les limites de la cohérence du système. Elles précisent quelles organisations sont logiques par rapport au système concerné et quelles organisations ne sont à priori pas logiques. Sans délimitation précise des règles de cohérence, il devient impossible de déterminer si

un système est cohérent ou non.

D'un autre côté, la stabilité repose aussi sur la définition de seuils et de bornes dans lesquelles le système doit se trouver (borne maximale et minimale pour la charge CPU moyenne ou charge moyenne souhaitée).

Nous pouvons donc dire que la cohérence et la stabilité d'un système repose d'une part sur l'évolution du système et d'autre part sur la description du cadre d'évolution avec des règles de cohérence et de bornes ou de seuils sur certains paramètres. Si une évolution du système reste dans le cadre d'évolution et dans les bornes définies, alors le système se retrouve dans un état cohérent et stable. Un moyen de parvenir à une évolution restant dans le cadre d'évolution consiste à lui fournir des contraintes ACID avec éventuellement des procédures d'annulation ("rollback") d'une part et définir des politiques spécifiques aux paramètres que l'utilisateur souhaite contrôler d'autre part.

6.1.2 Règles de cohérence internes à TUNe

Nous voulons en premier lieu définir le cadre d'évolution utilisé pendant la phase de gestion de TUNe en se basant sur le patron architectural fourni par le SDD. Nous introduisons différentes règles de cohérence qui doivent être respectées.

Nous avons vu que le SDD est représenté par un diagramme de classes et que la représentation du système se fait par une instanciation de ce diagramme (le modèle à composants). Cette instanciation est aussi appelée "System Representation" ou "SR" en notation abrégée. Le SR est donc une représentation interne du système administré par TUNe. Il désigne du point de vue de l'utilisateur l'ensemble des instances de classes du SDD et leurs liaisons, du point de vue de TUNe l'ensemble des composants Fractal et leurs liaisons et du point de vue application l'ensemble des processus s'exécutant.

Grâce aux différents PDD que nous avons présentés, l'utilisateur est en mesure de modifier le SR en rajoutant/supprimant des instances et en modifiant leurs liaisons. La suite d'actions de modification structurelle constitue une phase d'évolution du SR qui peut donc le mener dans un état incohérent s'il n'est pas conforme au SDD. En effet, les cardinalités de ce diagramme fournissent une contrainte sur les liaisons entre les éléments du système. Ces contraintes spécifient les règles de cohérence du système par rapport au SDD.

En premier lieu, nous introduisons une notation mathématique pour vérifier que le SR est cohérent avec les contraintes du diagramme de déploiement. Nous verrons ensuite les solutions de vérification de cohérence et de rétablissement de cohérence que nous apportons, notamment à l'aide des générations d'actions automatiques.

6.1.2.1 Règles de cohérence au SDD

Le SR est construit initialement à partir des attributs "initial" et des cardinalités minimales du SDD. Notons $initial(A)$ et $initial(B)$ les valeurs des attributs *initial* de deux classes liées A et B. Pour les bornes des cardinalités de chaque lien de la classe : si les classes A et B sont liées par une liaison $A : [t_{min}..t_{max}] - B : [u_{min}..u_{max}]$, cela signifie que chaque instance de A doit être liée au minimum à u_{min} instances de B et au maximum à u_{max} instances de B et chaque instance de B doit être liée au minimum à t_{min} instances de A et au maximum à t_{max} instances de A pour que le SR soit cohérent,

c'est-à-dire conforme au SDD.

Les contraintes initiales peuvent donc s'écrire :

$$(1) : initial(A) * u_{min} <= initial(B)$$
$$(2) : initial(B) * t_{min} <= initial(A)$$

Ces deux contraintes ne sont pas des règles de cohérence au SDD mais plutôt des règles de conformité du SDD. C'est-à-dire qu'elles permettent de vérifier que le cadre d'évolution est correctement défini. Pour qu'un SDD soit conforme, il faut qu'il respecte les contraintes (1) et (2).

La première règle de cohérence concerne l'utilisation des liaisons lors des appels de méthodes SWD. En effet, les méthodes définies en SWDL peuvent utiliser une notation pointée qui permet de naviguer dans le SR. Pour chaque appel de méthode SWD, il faut que toutes les liaisons utilisées par la méthode soient effectivement présentes. Si elles ne sont pas présentes c'est qu'il y a une incohérence dans l'utilisation des liaisons. Nous appelons ce type d'incohérence une **incohérence locale à l'appel SWD**. Le respect de toutes les liaisons utilisées lors de l'appel d'une méthode SWD spécifie la règle de cohérence (3).

Notons maintenant $Nb(A \rightarrow B)$ le nombre d'instances de A ayant une liaison avec B (et inversement).

La règle de cohérence pour les cardinalités minimales peut donc s'écrire (pour une liaison de A vers B à un instant t) :

$$(4) : Nb(A \rightarrow B) >= u_{min}$$
$$Nb(B \rightarrow A) >= t_{min}$$

La règle de cohérence pour les cardinalités maximales s'écrit de même :

$$(5) : Nb(B \rightarrow A) <= t_{max}$$
$$Nb(A \rightarrow B) <= u_{max}$$

Finalement, pour qu'une phase d'évolution soit cohérente par rapport au SDD, il faut qu'elle respecte les contraintes (3) et (4) et (5) citées ci-dessus. Dans le cas contraire, nous introduisons des solutions de récupération de la cohérence qui permettent de faire tendre le système vers un état cohérent. Ces solutions garantissent (à la manière des transactions pour les bases de données) qu'à la fin d'une évolution, le système est bien dans un état cohérent et que par conséquent la propriété de "self-protecting" est bien respectée.

6.2 Solutions de récupération de la cohérence

Afin de garantir que le système géré par TUNe est dans un état cohérent pendant la phase de gestion, nous introduisons des mécanismes de vérification de cohérence. Nous introduisons également des mécanismes d'automatisation de certaines actions pour permettre dans le cas de détection d'incohérences de ramener éventuellement le système dans un état cohérent.

Les vérifications de cohérence sont effectuées à deux niveaux : **pendant l'exécution des politiques** de gestion (PDD) et **à la fin de l'exécution d'une politique**

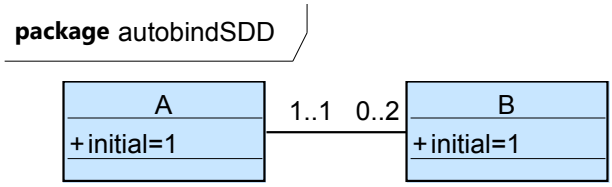


FIGURE 6.1 – SDD pour la création automatique de liaisons en cours d'exécution d'un PDD

```
<wrapper name='B'>
  .....
  <method name="start" key="extension.GenericStart"
   method="start" >
    <param value="$dirLocal/b $A.nodeName"/>
  </method>
......
</wrapper>
```

FIGURE 6.2 – SWDL utilisé pour la classe B

lors de l'arrivée au noeud final d'un PDD. Pour chaque vérification nous associons un certain nombre d'actions automatisées qui sont introduites dynamiquement au cours de l'exécution de la politique.

6.2.1 Vérification des liaisons utilisées par les SWD et création automatique de liaisons en cours d'exécution d'un PDD

La première vérification concerne le respect de la règle de cohérence (3), c'est-à-dire l'utilisation des liaisons lors des appels de méthodes SWD. Nous introduisons une vérification avec les étapes suivantes : avant chaque appel de la méthode, nous listons toutes les liaisons utilisées par la méthode. Nous vérifions ensuite que toutes ces liaisons sont effectivement présentes. Si elles ne sont pas présentes c'est qu'il y a une incohérence locale à l'appel du SWD dans l'utilisation des liaisons.

Dans le cas d'une incohérence détectée avant l'exécution de l'appel de la méthode SWD, la résolution de cette incohérence consiste à associer des actions de création automatique de liaison pour chacune des liaisons manquantes afin de ramener le système dans un état cohérent.

Pour illustrer notre propos, nous considérons le SDD simplifié décrit par la figure 6.1, ainsi que pour la classe B, son SWD représenté par la figure 6.2. Pour cet exemple de cadre d'évolution, nous spécifions qu'une instance de A peut être reliée à zéro, une ou deux instance(s) de B. Une instance de B doit par contre être reliée au minimum à une et une seule instance de A. Lors de la phase de déploiement initiale (c'est-à-dire à l'instanciation initiale du SR), une instance de A est reliée à une instance de B (les deux attributs *initial* sont égaux à un).

Considérons maintenant une politique de gestion qui fait évoluer le système en rajou-

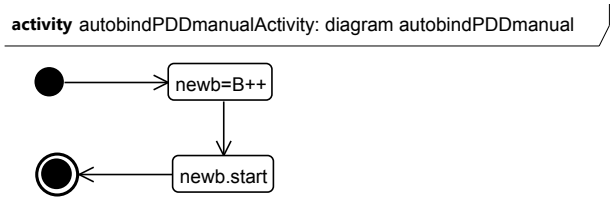


FIGURE 6.3 – PDD avec les actions manuelles - démonstration de la création automatique de liaisons

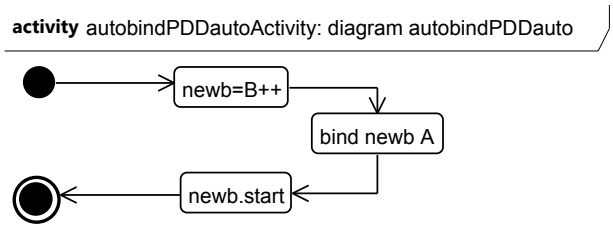


FIGURE 6.4 – PDD avec les actions manuelles et automatiques - démonstration de la création automatique de liaisons

tant une nouvelle instance de B et qui la démarre en faisant appel à la méthode *start* de son SWD. Cette politique est décrite par le PDD représenté par la figure 6.3. La première action `newb = B++` rajoute une instance de B. La deuxième action `newb.start` fait appel à la méthode *start* du SWD (cette méthode est représentée en figure 6.2).

Juste avant l'appel de méthode SWD, notre approche vérifie la cohérence locale de l'appel SWD (règle de cohérence (3)). En premier lieu les différentes liaisons utilisées par la méthode *start* sont listées. Dans notre exemple, la seule liaison utilisée est la liaison entre l'instance *newb* et une instance de la classe A. En effet, cette méthode utilise la liaison entre les classes A et B car la notation pointée `$A.nodeName` essaie d'accéder à l'attribut `nodeName` de l'instance de A reliée à l'instance `newb`.

La règle de cohérence (3) n'est pas respectée car cette liaison n'existe pas. Elle n'a effectivement pas été explicitement créée par le PDD avant l'appel de la méthode. Nous voyons donc qu'une **incohérence locale à l'appel SWD** apparaît lors de l'appel à la méthode `newb.start` et empêche celle-ci de s'exécuter correctement puisque la résolution de la notation pointée `$A.nodeName` ne peut pas s'effectuer.

Dans le cas d'absence d'une liaison utilisée par un appel de méthode du SWD, nous introduisons un mécanisme de création automatique de liaison. Cette création s'effectue de manière dynamique lors de l'exécution d'une politique de gestion. Il s'agit de l'ajout automatique d'une action de création de liaison `bind newb A` représentée par la figure 6.4. Cette action prend toujours comme argument la liste utilisée lors de l'appel de méthode du SWD (ici `newb`) et le nom de la classe spécifiée dans l'expression de navigation (ici A est extrait de la notation pointée `$A.nodeName`). Nous rappelons que dans ce cas, le nom de la classe représente l'ensemble des instances de cette classe. L'action `bind newb A` relie donc en priorité les instances de la liste passée en premier paramètre (*newb*) qui ne sont pas liées à des instances de la liste passée en deuxième paramètre (les instances de la classe A).

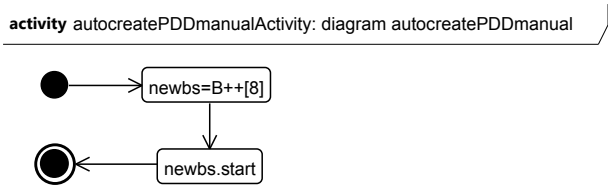


FIGURE 6.5 – PDD avec les actions manuelles - démonstration de la création automatique d'instances

Grâce à la création automatique de cette action, le système devient cohérent localement par rapport aux données nécessaires à l'appel de la méthode.

Notons aussi une deuxième incohérence plus globale mais toutefois acceptable en cours d'exécution d'une politique : le non-respect de la règle (4) au niveau de l'appel de la méthode, c'est-à-dire le non-respect des cardinalités minimales. En effet, l'instance `newb` n'est pas reliée au minimum requis d'une instance de la classe A. Il est important de noter que pour notre approche, cette deuxième incohérence est acceptable si elle est temporaire et qu'elle ne persiste pas jusqu'à la fin de l'exécution de la politique. En revanche, la première incohérence locale à l'appel SWD est par contre inacceptable car elle empêche d'exécuter correctement des actions de gestion sur les éléments du système géré. C'est pour cela que nous traitons en priorité et en cours d'exécution la règle de cohérence (3).

6.2.2 Vérification de cohérence des créations automatiques de liaisons et mécanisme de création automatique d'instances en cours d'exécution d'un PDD

La deuxième vérification concerne la cohérence des créations automatiques de liaisons. Nous avons vu qu'une création automatique de liaison est utilisée pour conserver un état cohérent local à un appel de méthode du SWD (respect de la règle de cohérence (3)). Cependant cette création de liaison peut échouer si toutes les instances de la classe concernée (c'est-à-dire celle utilisée dans le SWD) ont déjà atteint la cardinalité maximale (non-respect de la règle de cohérence (5)). Si aucune action n'est menée, alors l'appel de méthode SWD échoue car la liaison ne peut pas être résolue lors de l'interprétation de l'expression de navigation. Il s'agit toujours d'un cas d'**incohérence locale à l'appel SWD**.

Dans le cas de détection de cette incohérence, nous mettons en oeuvre un mécanisme de plus haut niveau : celui de **création automatique d'instances** qui permet de créer automatiquement autant d'instances que nécessaire pour revenir dans un état cohérent. Cette résolution d'incohérence tend à mener le système dans un état localement cohérent pour l'appel de méthode SWD de manière itérative à l'aide d'une **boucle de création**.

Pour illustrer nos propos, nous considérons la politique de gestion représentée par le PDD de la figure 6.5. Le cadre d'évolution est le même que précédemment (défini par le SDD de la figure 6.1). Cette politique décrit la création et le démarrage de huit instances de B. Lors de l'appel de méthode SWD `newbs.start`, la vérification de cohérence locale aux données du SWD échoue pour les mêmes raisons que précédemment (les liaisons utilisées dans le SWD n'existent pas). Comme expliqué dans la section précédente, les

liaisons manquantes utilisées dans celui-ci sont automatiquement créées pour faire tendre le système dans un état cohérent, mais échouent à partir de la deuxième.

En effet, d'après le SDD, chaque instance de B doit être liée à une instance de la classe A, or il n'y a qu'une instance de A initialement. Chaque instance de A ne pouvant être liée qu'à un maximum de deux instances de B, il faudrait donc créer quatre nouvelles instances de A pour arriver dans un état cohérent.

Dans le cas de la détection de ce type d'incohérence, notre approche consiste à créer autant d'instances que nécessaires et de les démarrer avec la suite d'actions par défaut suivante :

1. Vérification de la possibilité de création automatique d'instance.
2. Création d'un noeud de décision faisant suite à la création automatique de liaison.
3. Création d'une transition sinon (déclenchée si cette liaison échoue).
4. Création d'une transition qui continue sur l'appel de méthode SWD concerné si cette liaison réussit.
5. Création d'une nouvelle instance avec l'action `Nom_de_classe++` à partir de la transition sinon.
6. Création d'un appel de méthode SWD `configure` sur la nouvelle instance. Si cette méthode n'est pas définie dans le SWD cette action n'est pas créée.
7. Création d'un appel de méthode `start` sur la nouvelle instance.
8. Création d'une transition qui reboucle sur l'action de création automatique de liaison.

Notons que nous limitons la création automatique d'instances à un seul niveau de création, c'est-à-dire que les nouvelles instances créées automatiquement n'engendrent pas de nouvelles créations. En effet, les appels de méthode SWD créés utilisent potentiellement des liaisons qui n'existent pas avec la nouvelle instance créée. Il s'agit donc d'éviter de retomber dans un cas d'incohérence local à l'appel SWD (respect de la règle de cohérence (3)). C'est pour cela que la première action est une vérification de la possibilité de création automatique d'instances. Cette vérification consiste à s'assurer que les appels des méthodes *configure* et *start* n'utilise aucune notation pointée et par conséquent aucune liaison.

Ce choix vient du fait qu'il devient difficile de garantir la cohérence du système lors d'une création en chaîne non contrôlée. La chaîne peut potentiellement être infinie et faire exploser le système avec une création en boucle. Dans un premier temps, nous considérons donc que les cas de création en chaîne ou de création en boucle doivent être définies et contrôlées par l'utilisateur de façon manuelle dans la description des politiques. C'est notamment à l'utilisateur connaissant le logiciel géré de définir l'ordre de création et surtout l'ordre de démarrage de façon manuelle dans les politiques de gestion. Enfin, la création et le démarrage peut nécessiter de faire appel à des actions plus complexes qui ne font pas partie de la suite d'appels de méthodes `configure` et `start` utilisée par défaut pour résoudre ce cas d'incohérence. Nous considérons donc que si l'utilisateur ne veut pas de ce comportement par défaut ou s'il fallait enchaîner plusieurs boucles il doit explicitement spécifier le comportement attendu de manière manuelle dans le PDD.

D'après cette limite, l'**incohérence locale au SWD** peut persister si la première vérification échoue.

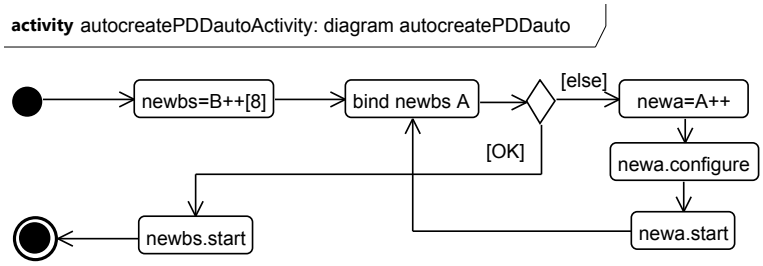


FIGURE 6.6 – PDD avec les actions manuelles et automatiques - démonstration de la création automatique d'instances

Pour illustrer notre propos, la figure 6.6 représente le PDD global résultant de la suite d'actions automatiques, qui met en évidence la boucle de création d'instances. Le même PDD que précédemment est utilisé avec les actions manuelles `newbs=B++[8]` et `newbs.start`, mais cette fois huit instances de B sont créées. Nous voyons qu'un noeud de décision est dynamiquement inséré juste après l'action de création automatique de liaison. Les actions `newa=A++`, `newa.configure` et `newa.start` constituent respectivement la création, la configuration et le démarrage de la nouvelle instance, c'est-à-dire la suite d'actions menée par défaut pour la résolution d'incohérence. Nous voyons que l'exécution reboucle sur l'action de création automatique de liaison. Cette boucle de création s'effectue autant de fois que nécessaire jusqu'à ce que l'action de création automatique de liaison réussisse, ce qui garantit la cohérence locale à l'appel SWD. Le flot d'exécution continue alors sur l'appel de méthode SWD `newbs.start`.

Notons par la même occasion que cette résolution d'incohérence permet de tenir compte des cardinalités minimales du SDD et résout aussi une partie des incohérences dues au non respect des cardinalités minimales (règle (4) de la section 6.1.2.1).

Dans le pire des cas et si une incohérence persiste, nous introduisons une **procédure d'interruption** qui amène l'exécution de la politique directement au noeud final du PDD qui se charge des vérifications de cohérence finales. Les procédures d'interruption sont explicitées dans la section suivante.

6.2.3 Interruption automatique en cours d'exécution d'un PDD

La troisième vérification de cohérence concerne les cas de création de liaisons explicitement définies par l'utilisateur (de façon manuelle dans le PDD) qui ne respectent pas les cardinalités maximales du SDD (règle (5) de la section 6.1.2.1), ainsi que les cas de création automatique d'instances qui ne peuvent pas avoir lieu.

Lors de l'exécution des actions de création de liaisons manuellement explicitées, si l'utilisateur ne spécifie pas la politique en cas d'erreur de la création de liaison (par exemple dans le cas des cardinalités maximales non respectées), tous les appels de méthode SWD suivants peuvent potentiellement échouer s'ils utilisent cette liaison. Il en va de même si la création automatique d'instance n'a pas pu avoir lieu.

Dans ce cas, nous considérons qu'il faut interrompre l'exécution de la politique et forcer la vérification de cohérence finale à l'aide d'une **procédure d'interruption**. Une autre solution serait d'envisager un calcul de toutes les créations d'actions automatiques

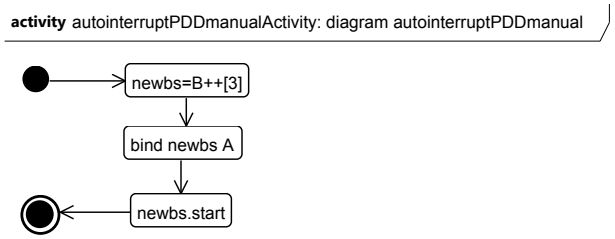


FIGURE 6.7 – PDD avec les actions manuelles - démonstration de l'interruption automatique

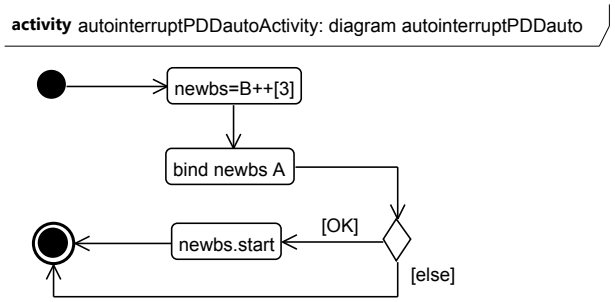


FIGURE 6.8 – PDD avec les actions automatiques - démonstration de l'interruption automatique

nécessaires pour parvenir à satisfaire la création de liaison explicite. Cette solution est néanmoins déjà mise en oeuvre si l'utilisateur ne spécifie pas explicitement la création de liaison (cas traité précédemment).

La figure 6.7 représente une politique décrite manuellement par l'utilisateur qui amène dans un état incohérent par rapport au SDD inital (figure 6.1). En effet, trois instances de B sont créées avec l'action `newbs=B++[3]` et la création de liaison avec l'unique instance de A est spécifiée de manière explicite avec l'action `bind newbs A`. Par conséquent cette action échoue puisque l'unique instance de A ne peut pas être liée à ces trois nouvelles instances (non respect de la règle (5)). Les liaisons n'étant pas créées, l'appel de méthode suivant ne peut pas s'effectuer correctement car l'incohérence de l'action précédente persiste et se propage jusqu'à l'appel SWD. Cette incohérence devient alors une **incohérence d'appel SWD**.

Si l'utilisateur n'avait pas spécifié l'action de création de liaison `bind newbs A` de manière explicite, nous retomberions dans le cas traité précédemment avec la création automatique de liaison et du nombre d'instances de A nécessaires automatiquement. Nous donnons donc le choix d'avoir une vérification de cohérence plus contraignante qui permet d'interrompre l'exécution du PDD en cas d'échec lors d'une création de liaison et de passer directement à la vérification de cohérence finale du PDD à l'aide d'une **procédure d'interruption**.

Pour cela, comme le montre la figure 6.8, un noeud de décision est dynamiquement placé juste après l'action de création de liaison explicite. La transition sinon (`else`) est directement reliée au noeud final du PDD et la transition `OK` au prochain noeud décrit de manière manuelle. Les actions de création automatique d'instances sont bien inhibées

Actions	Annulation
++	- -
- -	++
bind	unbind
unbind	bind

Actions	Annulation
start	stop
stop	start
configure	X

FIGURE 6.9 – Tableaux de transformation des actions en actions d'annulation. A gauche pour les actions de modification structurelle, à droite pour les appels de méthode SWD.

dans ce cas, rendant la vérification de cohérence plus stricte avec une interruption du flot d'exécution.

6.2.4 Vérifications de cohérences finales et procédures de résolution d'incohérences ou d'annulation

A la fin de l'exécution d'une politique de gestion, une dernière vérification globale de conformité établit toutes les incohérences issues du non respect des cardinalités minimales pour toutes les instances du SR. C'est-à-dire que pour chaque instance du SR, la règle (4) de la section 6.1.2.1 est vérifiée.

Les vérifications des cardinalités maximales (règle (5) de la section 6.1.2.1) ne sont effectivement pas nécessaires car toutes les actions de création de liaison explicitement décrites par l'utilisateur et menant au non respect des cardinalités maximales échouent et n'aboutissent donc pas au dépassement de la cardinalité maximale. Par contre, la vérification pour les cardinalités minimales est nécessaire car certaines instances ne respectent potentiellement pas les cardinalités minimales. En effet, les résolutions précédentes ne portaient que sur les incohérences locales aux appels SWD et non sur l'incohérence globale du SR. Les cardinalités minimales ne sont donc potentiellement pas respectées si certaines liaisons sont explicitées au niveau du SDD mais ne sont ni utilisées dans les appels de méthode SWD du PDD, ni créées manuellement dans la politique.

Cette dernière vérification intervient lors de l'arrivée sur le noeud final de la politique et uniquement si le PDD n'a pas été appelé en référence. En effet, un appel en référence à un PDD peut être vu comme un ensemble d'actions à effectuer au niveau de l'action d'appel en référence. Le PDD résultant sur lequel va être appliquée la vérification finale de cohérence est donc considéré comme le développement de tous les appels en référence dans un unique PDD.

La vérification finale de la cohérence consiste donc à appliquer les mécanismes de résolution d'incohérence précédemment introduits, notamment grâce à la création automatique de liaison et la création automatique d'instances pour satisfaire les cardinalités minimales de toutes les instances du SR. De même que précédemment, la création automatique d'instances ne prend en compte que le premier niveau de création (c'est-à-dire qu'elle ignore les créations en chaîne). Dans le pire des cas et s'il reste toujours des instances qui ne satisfont pas les cardinalités minimales, une **procédure d'annulation** (rollback) est initiée pour revenir à l'état cohérent initial du PDD.

Pour la procédure d'annulation, une mémorisation des actions effectuées lors de l'exécution du PDD, qui comprend également toutes les actions automatiques, est introduite. Cette mémorisation est stockée dans l'environnement d'exécution du PDD et contient un

horodatage de toutes les actions.

La procédure d'annulation consiste à parcourir toutes les actions dans l'ordre inverse d'exécution. Chaque action est transformée en son action inverse (qui doit annuler l'action concernée) puis exécutée. Les transformations par défaut sont présentées par le tableau 6.9.

Concernant les actions de modification structurelles, la création d'instances est transformée en destruction d'instances et inversement. La création de liaisons est transformée en destruction de liaisons et inversement. Pour les appels de méthode SWD, la méthode start est annulée grâce à la méthode stop et inversement. La méthode configure n'a pas d'équivalent d'annulation (représenté par un X dans le tableau). Les éventuels fichiers de configuration créés sont néanmoins supprimés lors de la suppression d'instance qui nettoie les machines concernées (suppression du répertoire de travail).

Nous rappelons que lors de la mise en oeuvre des interruptions automatiques en cours d'exécution (voir section précédente), la procédure d'annulation est directement appelée. En effet, il n'est pas nécessaire d'essayer de créer des liaisons ou des instances manquantes automatiquement puisque les appels de méthode SWD ont été potentiellement interrompus.

Il faut toutefois noter que des chemins d'exécution créés par des noeuds de type fork sont toujours potentiellement en attente au niveau des noeuds de type join dans le cas des interruptions automatiques en cours d'exécution (si cette interruption est survenue entre un noeud de type fork et un noeud de type join). La procédure d'annulation prend en compte toutes les actions y compris celles effectuées dans des chemins n'ayant pas atteint le noeud final (toujours en attente).

6.2.5 Vue globale de la gestion des cohérences

La figure 6.10 représente la vue globale des vérifications et des différents mécanismes mis en oeuvre pour rétablir le système dans un état cohérent.

A l'issue de ces différentes résolutions d'incohérences finales à l'aide de création automatiques de liaisons, de création automatiques d'instances ou de cette procédure d'annulation, toutes les instances du SR respectent les règles de cohérence définies en section 6.1.2.1. De ce fait, le système peut être considéré comme dans un état cohérent par rapport à la description de l'architecture donnée par le SDD. Le fait de maintenir le système dans un état cohérent lors des différentes phases de gestion garantit que la propriété de "self-protecting" interne est respectée pour ce système.

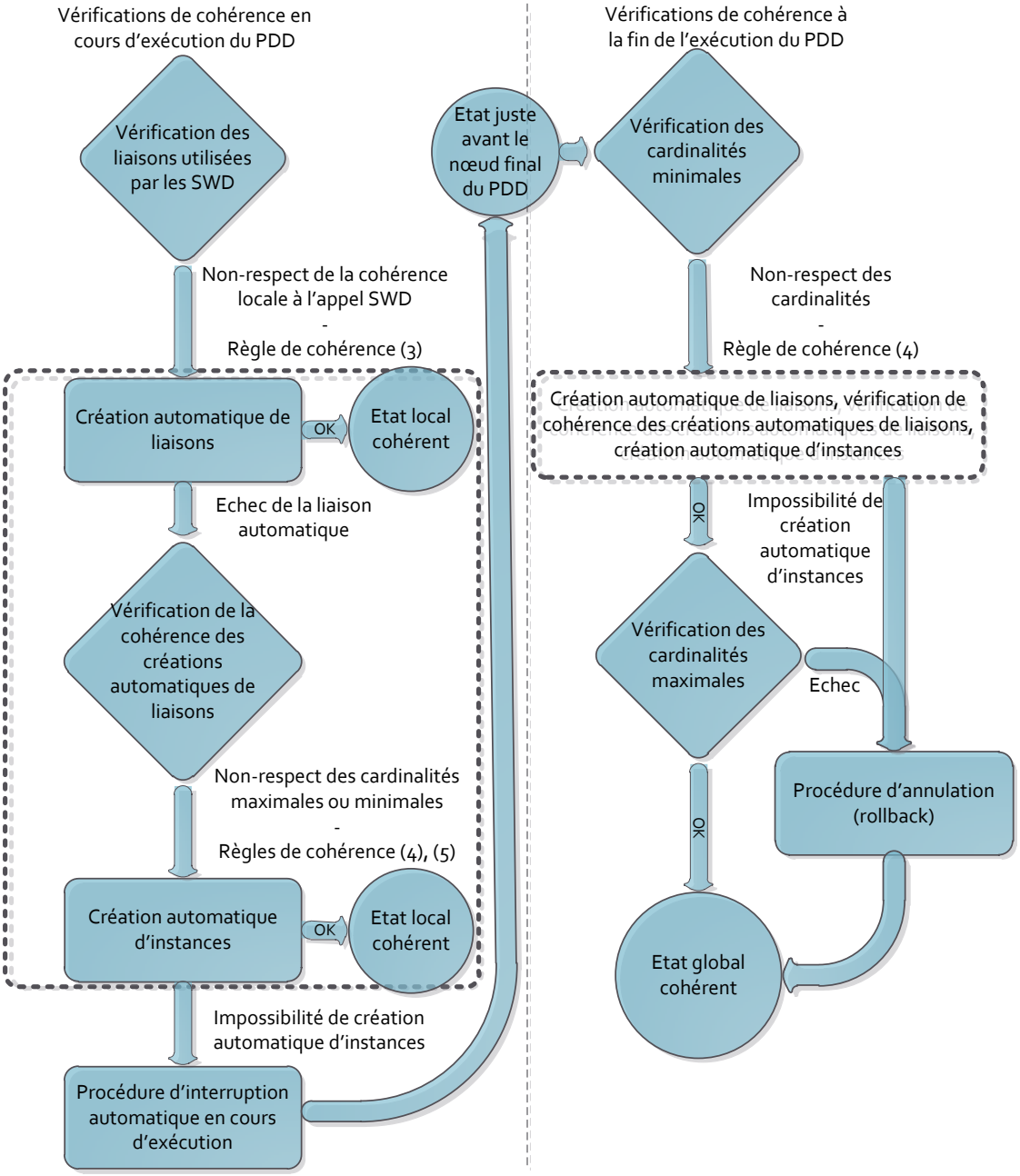


FIGURE 6.10 – Vue globale de la résolution d'incohérences

6.3 Mise en oeuvre du "self-protecting" pour l'architecture DIET

6.3.1 Présentation du cadre de mise en oeuvre

Par cette mise en oeuvre, nous voulons montrer que le rajout de la propriété de "self-protecting" permet au système DIET de rester conforme à la spécification donnée par le SDD lors des reconfigurations spécifiées par les PDDs. Nous utilisons pour cela des PDDs qui définissent des reconfigurations de l'architecture DIET en rajoutant et en supprimant des instances des agents DIET (SED, LA, MA). En fonction du cadre d'évolution défini par le SDD de l'architecture DIET et en particulier des cardinalités entre les différentes classes des agents DIET, nous montrons les vérifications de cohérence et les différentes solutions mises en oeuvre pour résoudre les problèmes d'incohérence.

Nous voulons également que pendant la phase de gestion, le nombre de SED à rajouter ou à supprimer soit calculé pour atteindre une charge moyenne des SEDs cible prédéfinie. C'est à dire que nous voulons montrer qu'il est possible d'arriver à une certaine stabilité du système. En effet, l'architecture DIET est conçue pour être une plateforme multi-clients, ce qui veut dire que les clients arrivent à tout moment pour demander un service de calcul. De ce fait, les composants logiciels serveurs de calculs SEDs peuvent rencontrer d'importantes fluctuations de leurs charges qui dépend, entre autres, du nombre de clients qu'ils doivent traiter en parallèle. De plus, les LAs qui doivent répartir les requêtes vers les SEDs peuvent aussi rencontrer une charge fluctuante qui dépend, entre autres, du taux d'arrivée des requêtes clientes et du nombre de SEDs dont ils ont la charge. Il s'agit donc aussi d'une optimisation du nombre de SED au cours de l'exécution de DIET, ce qui rajoute également la propriété de "self-optimizing".

6.3.1.1 PDD de gestion globale de l'architecture DIET

Pour ces expériences, nous utilisons le SDD de la figure 6.11. Ce SDD est une version simplifiée pour une meilleure lisibilité. Nous voyons que pour être conforme au SDD, un système DIET doit avoir un serveur de nom *OMNI* lié jusqu'à 2 *MA*, 10 *LA* et 500 *SED*. De même, un *MA* peut être lié jusqu'à 5 *LA* et un *LA* jusqu'à 50 *SED*. Chaque agent doit être relié à sa propre et unique sonde *PROBEMA*, *PROBELA* ou *PROBESED*.

Ces sondes génériques envoient la charge périodiquement (attribut *refreshrate*, en ms) dans une notification qui porte le nom du PDD : *refreshloads*. Ces sondes récupèrent la charge CPU à l'aide de la commande `uptime`. Sur des systèmes mono-CPU la charge représente le pourcentage d'utilisation du système durant la dernière minute, c'est à dire une évaluation du nombre de processus actifs attendant le processeur (présents dans la file d'attente du processeur). Par exemple, une charge de 200% avec deux processus équivaut à dire que si les deux processus étaient répartis sur deux processeurs, la charge de chacun des processeurs serait alors de 100%. Pour des systèmes multi-processeurs, la sonde divise la charge obtenue par la commande `uptime` par le nombre de processeurs afin d'obtenir un pourcentage comparable.

Nous introduisons une première politique de gestion globale de haut niveau pour l'architecture DIET et en particulier pour la gestion des SEDs. La politique globale est décrite par un PDD représenté par la figure 6.12. Cette politique est exécutée quand

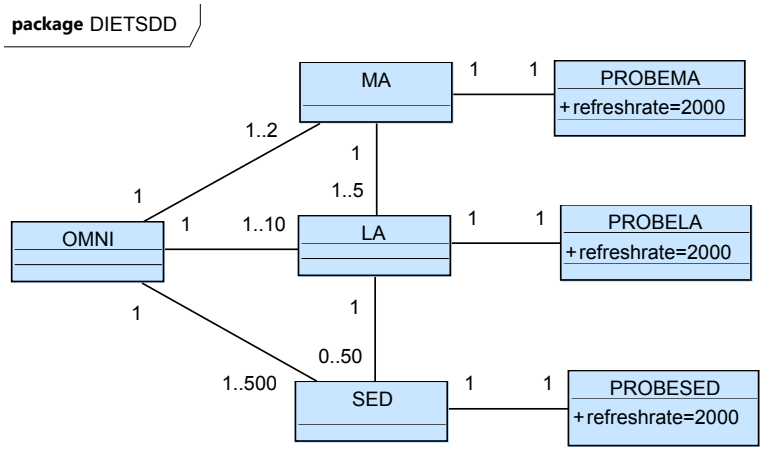


FIGURE 6.11 – SDD simplifié de DIET pour le self-protecting

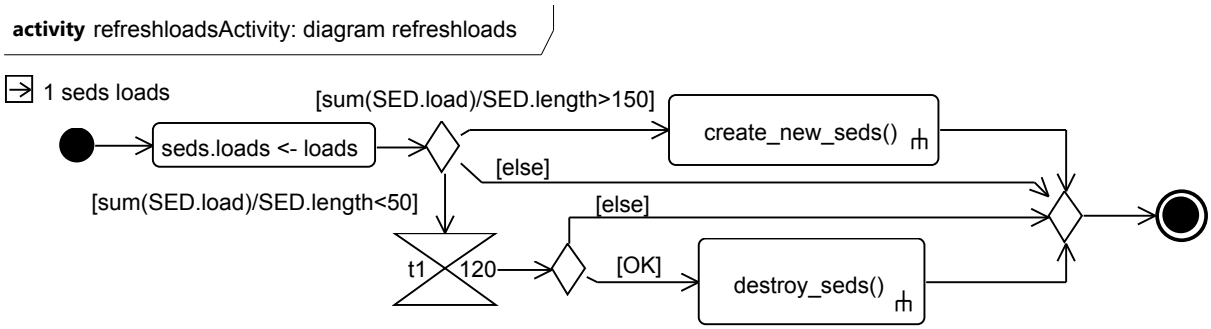


FIGURE 6.12 – PDD global pour l'architecture DIET.

TUNe reçoit une notification dont le nom est *refreshloads*.

Le premier paramètre d'entrée *1 seds loads* contient les références des éléments monitorés et les valeurs respectives de leurs charges. L'action de modification d'attribut $seds.load <- loads$ met à jour les valeurs des attributs load avec ces valeurs fraîchement renvoyées. Si la charge moyenne de tous les SEDs dépasse 150, une politique appelée *create_new_seds* est exécutée pour créer de nouvelles instances de SEDs. Si cette charge moyenne passe en dessous de la valeur 50, le timer t1 est déclenché pour 120 secondes et la politique appelée *destroy_seds* est exécutée pour détruire des instances de SEDs. Le timer assure que le PDD référencé *destroy_seds* n'est exécuté qu'une fois toutes les 120 secondes au maximum. En effet, si le timer du même nom est déjà déclenché et n'est pas arrivé à expiration, le flot d'exécution du PDD continue sur la transition *else* du noeud de décision situé juste après le timer.

Nous définissons ensuite trois PDD : *create_new_sed*, *create_new_la* et *create_new_ma* qui ont la charge de créer et de démarrer chacun des nouveaux agents DIET. Ces différents PDDs s'appellent en référence, notamment la politique de création de SED qui appelle éventuellement la politique de création de LA et la politique de création de LA qui appelle éventuellement la politique de création de MA.

Pour déclencher la politique de création de nouveaux SEDs, nous avons créé un injecteur de charge qui émule des clients. Ces clients envoient des requêtes de calcul (de

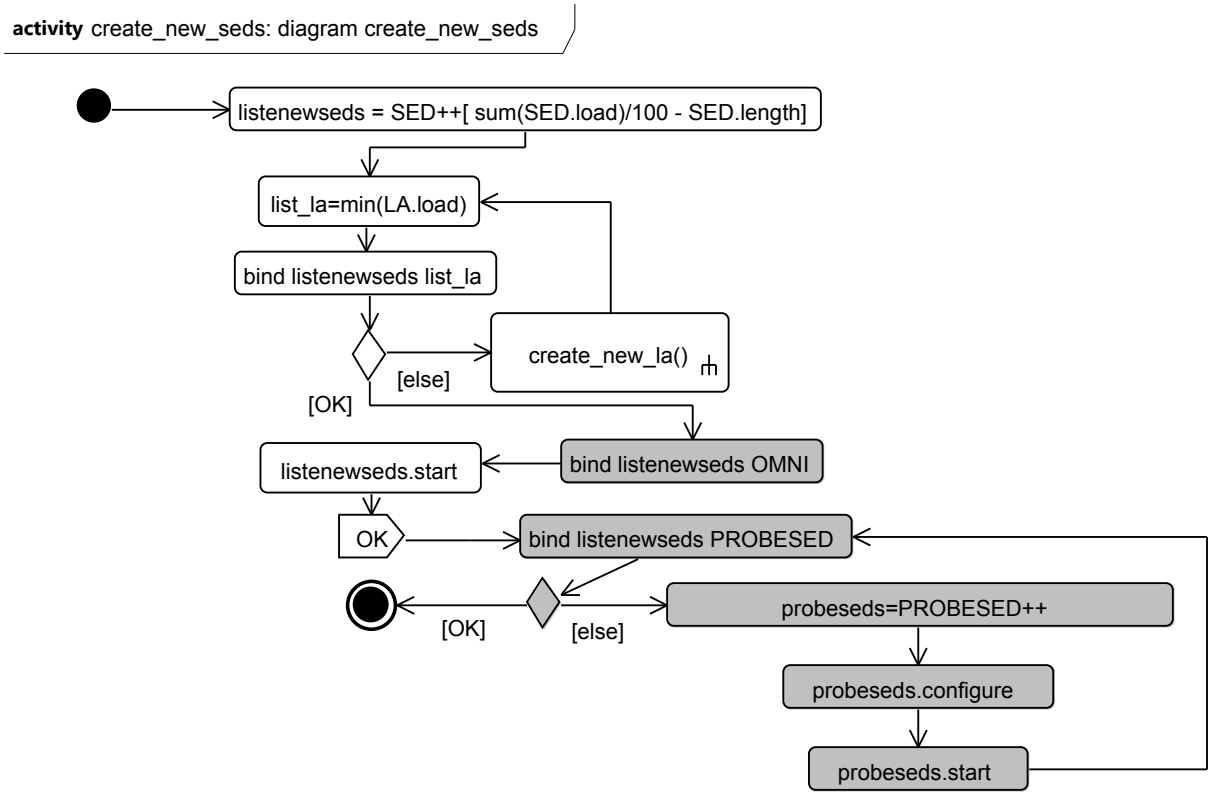


FIGURE 6.13 – PDD avec les actions automatiques en gris pour la création d'un nouveau SED

type multiplication de matrice) à l'architecture DIET. Pour simplifier, nous avons dans un premier temps fixé une requête toutes les 14 secondes et chaque requête prend environ 2 minutes de temps CPU sur un ordinateur disposant de deux processeurs Quad Core Intel Xeon 2.83 GHz.

6.3.1.2 Politique de création de nouvelles instances de SEDs

Nous introduisons une première politique de création de nouvelles instances de SEDs *create_new_seds*. Le PDD de la figure 6.13 représente les actions manuelles sur fond blanc et les actions automatiques sur fond gris. Le but est de calculer le nombre de nouveaux SEDs à créer pour arriver à une charge moyenne cible stable (pour maintenir la stabilité du système), en utilisant l'équation suivante :

$$x = \left\lceil \frac{\sum_{i=1}^{n} C_i}{T} - n \right\rceil$$

avec x : le nombre de SEDs à créer, C_i : la charge du SED numéro i, n : le nombre actuel de SEDs et T : la charge cible.

Cette équation est exprimée dans la première action du PDD. La charge cible est fixée à 100%, c'est à dire un *uptime* stable et autour de 100%. Les nouvelles instances de SEDs sont connectés au LA approprié. En effet, *list_la = min(LA.load)* crée une liste filtrée des LAs en ne sélectionnant que ceux qui minimisent leur charge CPU. Comme

plusieurs LAs peuvent éventuellement avoir la même valeur de charge CPU minimale, au moment de la création de liaison (*bind list_newsed list_la*) ceux qui sont déjà liés à un minimum de SEDs sont choisis. Si tous les LAs sont déjà reliés à leur maximum de SEDs (définit par la cardinalité maximale entre LA et SED), alors d'autres sont créés (l'action de liaison échoue et continue sur le *else*). Dans ce cas de nouveaux LAs sont créés grâce aux appels en référence du PDD *create_new_la*. Le PDD de création du LA est très proche de celui de la création des SEDs et fait appel à la création des MAs en référence (PDD create_new_ma). Ces deux PDDs sont donnés en annexes C.5 et C.6.

Une fois les nouveaux SEDs reliés aux LAs, ils sont tous démarrés avec l'appel de méthode *listenewseds.start*. La méthode de démarrage des SEDs définie dans le SWD a besoin de parcourir les liaisons vers l'OMNI (pour communiquer avec le serveur de nom) et vers le LA (pour se connecter à l'agent Local). Cet appel a donc une incohérence locale à l'appel SWD puisque la liaison vers l'OMNI n'a pas été créée. Notre approche crée donc une action automatique *bind listenewseds OMNI* placée juste avant l'appel de méthode. Cette action permet de rétablir la cohérence locale à l'appel SWD pour que l'exécution de l'appel se déroule correctement. Une fois les SEDs démarrés le PDD renvoie le code de retour *OK* et l'exécution arrive au noeud final.

Avant d'exécuter le noeud final du PDD, notre approche fait les vérifications de cohérences finales et éventuellement les procédures de résolution d'incohérences ou d'annulation. Le premier contrôle de cohérence consiste à vérifier si, pour toutes les instances du SR, les cardinalités sont respectées. Ici, pour les nouveaux SEDs créés, la cardinalité vers les sondes *PROBESED* (1-1) n'est pas respectée puisque la liaison n'a pas été créée. La première procédure de résolution de cette incohérence rajoute l'action de création de liaison *bind listenewseds PROBESED*. Comme chaque sonde PROBESED ne peut être reliée qu'à un seul SED, aucune sonde déjà reliée à un SED n'est disponible pour recevoir une nouvelle liaison. La deuxième procédure de résolution d'incohérence crée donc des instances de PROBESED avec la suite d'actions standard : *probeseds=PROBESED++* , *probeseds.configure* , *probeseds.start*. Cette suite d'actions reboucle sur la création de liaison pour créer autant de sondes que nécessaire. Dans notre cas, il faut exactement le même nombre de sondes que de SEDs nouvellement créés (cardinalité 1-1).

La figure 6.14 (a) montre comment la charge moyenne de tous les SEDs est affectée quand la politique de création des nouvelles instances de SEDs est exécutée. Durant les six premières minutes, la charge moyenne de tous les SEDs est en train d'augmenter rapidement et dépasse le seuil maximum (maximum threshold) trois fois. Un nombre calculé de nouveaux SEDs sont créés pour atteindre la charge optimale CPU de 100%. Trois nouveaux SEDs sont créés la première fois, quatre la deuxième fois et six la troisième fois. Nous pouvons voir que la charge moyenne continue d'augmenter un petit peu au dessus du seuil maximum à chaque fois. Ceci est dû au fait qu'il faut quelques secondes pour créer et démarrer tous les nouveaux SEDs. De t=7 à t=15 minutes, 17 serveurs sont en marche et absorbent les requêtes des clients. Leur charge moyenne se stabilise autour de 60% à t=10 minutes. Le petit dépassement du seuil cible à 6 minutes est dû au fait que l'ordonnancement interne de l'architecture DIET est par défaut un *round robin* qui ne prioritise pas l'utilisation des SEDs nouvellement créés. De ce fait, quelques requêtes clients sont toujours envoyées aux SEDs surchargés jusqu'à ce que l'ordonnancement atteigne les nouveaux.

La figure 6.14 (b) montre un exemple d'exécution de la politique de destruction des

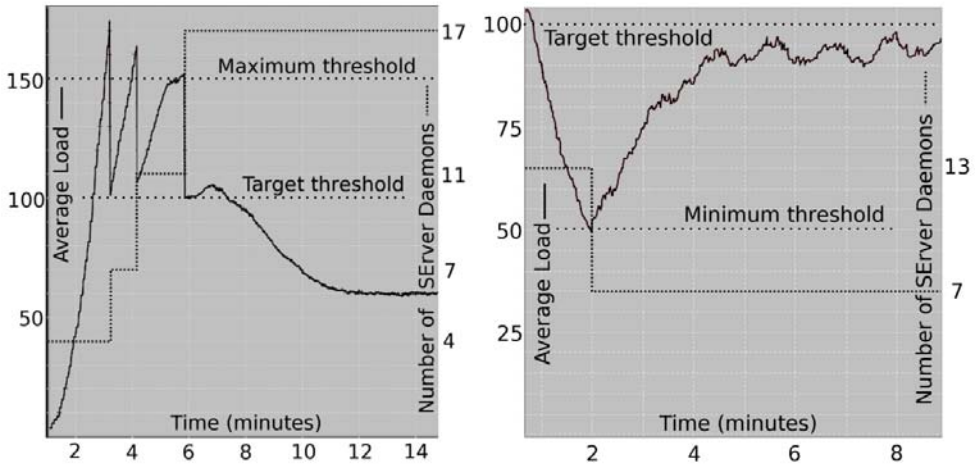


FIGURE 6.14 – Résultat de l'expérience pour (a) gauche : la création de nouvelles instances de SEDs (b) droite : la destructions d'instances de SEDs.

SEDs sous-chargés *destroy_seds*. Pour cela, nous avons d'abord surchargé 13 SEds puis diminué le nombre de requêtes à une requête toutes les secondes, avec un calcul demandant six secondes CPU pour que leur charge moyenne passe en dessous du seuil minimum (*minimum threshold*). Le nombre de SEDs à supprimer est calculé de la même manière que pour la politique de gestion des SEDs suchargés (le PDD est donné en Annexe C.2). Quand un SED est supprimé, il ne reçoit plus de requêtes et termine les calculs en cours. Une fois toutes les requêtes terminées, TUNe arrête tous les processus, nettoie les fichiers de travail et rend les machines disponibles à nouveau. Nous voyons que la charge moyenne se stabilise autour de 95% de t=2 à t=4 minutes. En effet, les requêtes sont redirigées vers les SEDs restants qui ne sont pas supprimés et ce temps de transfert de charge et de stabilisation peut éventuellement prendre plus de temps. Il dépend du taux de requêtes et du temps de calcul de chaque requête. C'est la raison pour laquelle le timer t1 est utilisé. En effet, le timer t1 attendrait deux minutes pour la prochaine exécution du PDD de destruction des instances de SEDs si la charge restait sous le seuil minimum (ce qui n'est pas le cas ici).

Les annexes C.3 et C.4 présentent deux expériences qui sont plus tournées sur la propriété de "slef-optimizing" et qui utilisent le même mécanisme de gestion de cohérence.

6.4 Conclusion du chapitre "self-protecting"

6.4.1 Contributions au "self-protecting" des systèmes autonomiques

Dans ce chapitre, nous avons introduit une approche pour permettre d'ajouter la propriété de "self-protecting" aux systèmes gérés par TUNe. Nous avons focalisé notre approche sur le "self-protecting" interne, c'est-à-dire la protection du système face aux incohérences. Dans TUNe, le cadre de cohérence est donné par le SDD et notamment les cardinalités entre les classes du SDD. Chaque cardinalité fournit une information sur le

nombre minimal et maximal d'instances qui doivent être reliées entre elles. Lors de l'exécution de politiques de reconfigurations exprimées à l'aide de PDD, la modification de la représentation du système interne (SR) peut le mener dans un état incohérent. Nous avons introduit des vérifications de cohérence à deux niveaux : lors de l'exécution des politiques et à la fin d'une politique. Nous avons introduit des règles de cohérence qui permettent de détecter ces incohérences. Si une incohérence est détectée, diverses actions automatiques tentent de ramener le système dans un état cohérent. Ces actions automatiques sont rajoutées aux diverses politiques de façon dynamique. Il s'agit des actions de création automatique de liaisons, de création automatique d'instances et d'interruption de l'exécution de la politique. A la fin de l'exécution d'une politique et de toutes les actions automatiques, une dernière vérification de cohérence globale est effectuée. Si elle échoue, une procédure d'annulation est déclenchée. Cette procédure permet, à la manière des transactions pour les bases de données, de ramener le système dans l'état cohérent initial présent avant l'exécution de la politique. Cette procédure se base sur l'annulation ("rollback") des actions effectuées.

La mise en oeuvre de notre approche sur le système DIET démontre qu'il est possible d'appliquer les propriétés de "self-protecting" sur les systèmes propriétaires. Ainsi, la phase de gestion de TUNe applique les différentes politiques de reconfiguration dans un cadre de cohérence défini par le SDD. Les différentes solutions de résolution des incohérences introduites permettent de rester à l'intérieur du cadre d'évolution et évitent par conséquent les explosions du système.

6.4.2 Limites de notre approche

La problématique soulevée dans les sections précédentes concerne la cohérence du SR par rapport au SDD. Cependant, notre approche ne tient pas compte de la vérification de la synchronisation entre le SR et le monde réel. Les PDD sont une interface d'administration du SR car ils permettent sa modification (avec les actions de modification structurelles). Il sont aussi une interface d'administration du monde réel car ils permettent l'appel de méthode des SWD. Un PDD fait donc évoluer le SR et le monde réel en parallèle. Notre approche ne vérifie pas qu'à la fin de l'exécution du PDD, les deux évolutions sont équivalentes, c'est à dire que chaque action de modification du SR a son équivalent dans le monde réel et inversement. De plus, les possibles actions d'annulation (*rollback*) supposent que le système réel est sans mémoire.

Nous avons néanmoins introduit une aide supplémentaire sous forme d'avertissement pour l'utilisateur qui vérifie un minimum de synchronisation entre le SR et le monde réel. Pour cela, nous associons des actions de modification structurelles avec des appels de méthodes des SWD et nous vérifions que ces couples d'actions sont présents dans les diagrammes d'activité :

- Pour l'action de modification structurelle "++", nous vérifions que l'appel de méthode "start" est aussi effectué par la suite. La vérification inverse n'est pas nécessaire car l'appel de méthode "start" ne peut s'effectuer que sur une instance déjà existante dans le SR.
- Pour l'action de modification structurelle "- -", nous vérifions que l'appel de méthode "stop" est aussi préalablement effectué. La vérification inverse est dans ce cas nécessaire et nous vérifions également qu'après chaque appel de méthode "stop"

l'utilisateur a utilisé l'action de modification structurelle "- -".
Il ne s'agit ici que d'une aide facultative pour l'utilisateur, car il se peut que dans certains cas, il ait fait appel à "stop" suivi de "start" pour redémarrer un processus sans qu'il y ait forcément besoin de répercuter le changement au niveau du SR en faisant un "- -" suivi d'un "++". De plus, une politique peut très bien définir un arrêt de plusieurs éléments qui seront par la suite redémarrés par une autre politique, sans pour autant les supprimer entre temps.

Chapitre 7

Conclusions et perspectives

Sommaire

7.1 Conclusions 139
7.2 Perspectives 142
7.2.1 Poursuite de l'intégration des contributions au gestionnaire autonomique TUNe. 142
7.2.2 Gestion avancée des événements 143
7.2.3 Vers une gestion autonomique complète des "datacenters" . . . 144

7.1 Conclusions

Les systèmes informatiques modernes se densifient et se complexifient d'une telle façon que leur gestion dépassera les capacités humaines dans les années à venir. De nos jours, cette complexité est gérée par des administrateurs hautement qualifiés, mais les coûts de maintenance, le temps de réactivité et les erreurs issues d'une gestion par des humains ne suffit plus à combler la demande croissante en administration. Un des paradoxes de ces systèmes informatiques est qu'ils ont apporté une puissance considérable pour l'automatisation des tâches de calcul mais l'instabilité matérielle et logicielle due à à la diversité et l'hétérogénéité des systèmes engendre des pannes qui deviendront vite insurmontables si leur gestion n'est pas elle-même automatisée. Le problème général de la complexification des systèmes est qu'elle devient un facteur limitant très significatif qui freinera le développement des futurs systèmes. Kephart et Chess [4] résument le problème en mentionnant que le rêve de l'interconnectivité de tous les matériels et systèmes futurs pourrait bien devenir le cauchemar de l'informatique pervasive, dans lequel les architectes seront incapables d'anticiper, de concevoir et de maintenir la complexité des interactions de tous les éléments.

Un des domaines de recherche émergents de ce constat est l'informatique autonomique. Ce concept s'inspire du domaine biologique et notamment du système nerveux autonomique du corps humain, qui se gère lui-même et de manière inconsciente pour le sujet. Cette gestion contrôle pourtant les fonctions critiques du corps humain tels que la respiration, le rythme cardiaque, la pression sanguine etc... L'essence d'un système autonomique vient du fait qu'il se gère tout seul, libérant ainsi les administrateurs des

tâches de gestion de bas niveau, tout en offrant un comportement optimal du système à tout instant, y compris face aux pannes.

Une des solutions proposées par ce domaine de recherche est de permettre aux systèmes informatiques de se gérer eux-mêmes sans intervention humaine directe, on peut les appeler les auto-systèmes ou les systèmes autonomiques. Pour cela, l'opérateur humain joue un nouveau rôle : il ne contrôle pas le système directement. A la place, il définit des politiques générales et des règles de haut niveau qui servent d'entrées au processus de gestion. Ce processus de gestion se décompose en différentes tâches qui forment une boucle de contrôle fermée. Les boucles fermées sont un concept issu du domaine de recherche en théorie de contrôle des processus : le système monitore lui-même ses ressources (logicielles ou matérielles) et exécute des actions de manière autonome en essayant de garder un certain nombre de paramètres dans des bornes définies (nombre de composants, charge, qualité de service, énergie...). Un exemple d'implémentation est la boucle MAPE-K introduite par IBM. Cette boucle décompose les tâches de monitoring M, d'analyse A, de planification P et d'exécution E, en gardant l'ensemble des paramètres et l'historique des événements dans la base de connaissances K.

Dans ce cadre, cette thèse adresse la problématique de gestion autonomique des systèmes propriétaires et introduit des descriptions de politiques de gestion de haut niveau qui peuvent s'appliquer à de nombreux systèmes, sans modification préalable de leur fonctionnement. Notre approche propose de rajouter les quatre propriétés de gestion autonomique aux systèmes patrimoniaux : le "self-healing", le "self-configuring", le "self-optimizing" et le "self-protecting". Nos contributions se situent donc au niveau de ces quatre propriétés fondamentales qui forment la gestion autonomique ou encore le "self-management" d'un système. Nous nous basons sur le gestionnaire autonomique TUNe, développé à l'IRIT et qui implémente la boucle de contrôle MAPE-K.

La première contribution de cette thèse porte sur la définition de méta-modèles pour la description de politiques de haut niveau. Nous avons introduit la méta-modélisation des Policy Description Diagrams **PDD** qui permettent de décrire un processus de gestion de manière graphique et intuitive pour l'utilisateur final. Nous avons spécifié différents types d'actions et de noeuds de contrôle : les **actions de modification structurelles** ont un impact sur l'architecture du système géré. Elles permettent de rajouter ou de supprimer des instances de certains éléments du système en cours d'exécution, ainsi que de rajouter ou supprimer des liaisons entre ces instances. Les **modifications et sélection de composants** permettent de reconfigurer certains attributs des éléments du système de manière dynamique, de les filtrer suivant certain critères et les **actions d'appel aux méthodes** permettent d'agir sur ces éléments en exécutant des actions sur le système réellement déployé. En ce qui concerne les noeuds de contrôle, nous avons spécifié des **noeuds de décision** qui créent des chemins d'exécution différents en fonction de conditions exprimées sur des **transitions** sortantes de ces noeuds. D'autres noeuds tels que les **fork**, **join**, **timer** ou **codes de retour** permettent de contrôler le flot de travail des actions à mener pour la politique de gestion.

Une première validation de notre approche porte sur le rajout de la propriété de "self-healing" au système DIET. Des politiques ont été introduites pour décrire tout d'abord le démarrage et le déploiement initial, puis une politique utilisée pendant la phase de gestion a permis de ramener le système dans un état opérationnel en cas de panne d'un cluster. Pour cela les différentes sondes génériques détectent les événements de pannes

(mort de processus, connections impossibles sur des machines) et renvoient à TUNe des notifications qui permettent d'exécuter les différentes politiques. Nous avons comparé une approche qui consiste à récolter tous les événements de toutes les sondes et une approche qui agrège plusieurs événements au niveau de la sonde. Cette dernière approche s'est montrée plus pertinente dans le cas d'une panne de cluster, l'agrégation des événements résultants de la panne permettant de traiter tous les éléments du système impactés en une seule exécution de politique.

La deuxième contribution se situe au niveau de la propriété de "self-configuring", aux niveaux logiciels et matériels. En effet, l'installation, la configuration, le démarrage ou la mise à jour de systèmes distribués peut vite devenir une tâche fastidieuse si elle n'est pas gérée de manière automatique. Le problème vient du fait qu'un certain ordre dans les actions doit être respecté lors du déploiement de modules logiciels ou lors de la phase de démarrage ou d'arrêt. Aussi, certaines parties du système peuvent démarrer de manière parallèle et d'autres ont besoin d'attendre la fin d'exécution de certaines tâches pour continuer. Notre approche permet de spécifier cet ordre à l'aide des PDDs. Nous avons validé notre approche sur le simulateur électromagnétique yatpac. La gestion de son déploiement à grandement simplifié l'installation, le démarrage, la désinstallation et l'arrêt des modules logiciels pour la simulation électromagnétique sur la grille de calcul Grid'5000 pour les ingénieurs électroniciens du LAAS-CNRS.

De plus, notre approche du "self-configuring" porté au niveau matériel a permis de focaliser notre approche sur la gestion événementielle hiérarchique. En effet, nous avons introduit une gestion autonomique pour les réseaux de capteurs qui distingue différents types d'événements, tout en s'intégrant dans le logiciel de gestion de réseaux de capteurs existant XSStreaMWare. Cette approche consiste à rajouter à tous les niveaux hiérarchiques de ce logiciel de gestion un nouveau service ECA (Event Condition Action). Ce service est décomposé en un moteur d'événements et un moteur de règles basé sur le moteur que nous avons introduit pour l'exécution des politiques de gestion dans TUNe. Les règles sont définies avec les PDDs (Policy Description Diagrams) qui permettent de faire appel aux fonctionnalités spécifiques de gestion des capteurs matériels. Une validation de l'approche menée au National Institute of Informatics de Tokyo montre comment décrire une politique de mise à jour des micrologiciels d'un parc de capteurs SunSPOT ou la reconfiguration dynamique de paramètres internes (par exemple le taux d'échantillonnage) en fonction du niveau de batterie.

Toujours au niveau matériel, nous nous sommes intéressés pour la troisième contribution au "self-optimizing" à la gestion d'énergie et son impact sur la qualité de service dans les réseaux filaires. Cette approche s'applique à l'optimisation de l'utilisation des ressources informatiques au niveau d'un "datacenter", notamment en ce qui concerne leur consommation d'énergie, tout en garantissant une certaine qualité de service pour les applications déployées dans le "datacenter". Nous avons formalisé un modèle mathématique qui prend en compte cette dualité dans un critère à minimiser. Ce critère définit un compromis entre, d'une part, la consommation électrique de l'infrastructure réseau et, d'autre part, la dégradation de la qualité de service des applications utilisant ce réseau. L'administrateur peut alors jouer sur le coût énergétique ou sur la performance globale du "datacenter" à l'aide d'un seul paramètre contrôlant la politique de "self-optimizing". Une heuristique a été introduite pour permettre de prendre en compte des fonctions de routage propres au réseau (par exemple OSPF). En effet, ces fonction rajoutent des contraintes

supplémentaires qui sont elle-mêmes des minimisations de critères (minimisation du coût pour les chemins en fonction des coûts des liens en OSPF).

Enfin, notre dernière contribution concerne l'apport de la gestion des cohérences, notamment lors des modifications structurelles du système géré. Cet apport fait partie de la propriété de "self-protecting" interne des systèmes autonomiques, elle permet notamment de vérifier que l'exécution des politiques de gestion amènent le système dans un état cohérent. Notre approche introduit des règles de vérification de la cohérence du système en cours de l'exécution des politiques et des mécanismes de résolution de cohérence permettant de ramener éventuellement le système dans un état cohérent. A la manière des transactions dans le domaine des bases de données, nous introduisons un mécanisme d'annulation (rollback) qui s'enclenche dans le pire des cas et si aucune des résolutions n'a permis de remettre le système dans un état cohérent à la fin de l'exécution d'une politique.

Toutes ces contributions ont montré que le domaine de recherche en informatique autonomique est très vaste. Nous avons validé nos approches se basant sur les "self-*" et nous avons garanti la généricité de celle-ci. En effet, les différentes applications n'ont pas été modifiées lors de leur gestion autonomique. De nombreuses problématiques restent ouvertes et plusieurs aspects de nos travaux peuvent être poursuivis.

7.2 Perspectives

Les travaux réalisés durant cette thèse ont ouvert de nouvelles possibilités d'utilisation de l'informatique autonomique. Les perspectives proposées concernent à la fois des extensions de nos travaux et la proposition de nouvelles voies.

7.2.1 Poursuite de l'intégration des contributions au gestionnaire autonomique TUNe.

La partie "self-optimizing" de ce document a formalisé un modèle mathématique de gestion autonomique du réseau et des applications déployées dans un "datacenter". Ce modèle introduit un critère à minimiser faisant intervenir un compromis entre la consommation électrique et la qualité de service obtenue pour les applications. Il a ensuite été implémenté pour un calcul des solutions optimales, ainsi qu'avec une heuristique prenant en compte les spécificités de la fonction de routage du réseau. Il reste néanmoins à intégrer ce développement au sein du gestionnaire autonomique TUNe. Pour ce faire, il faudrait améliorer la description matérielle interne à TUNe, c'est-à-dire étendre le HDD (Hardware Description Diagram) pour prendre en compte la topologie réseau complète du "datacenter". L'extension devrait posséder la description des routeurs, des châssis et des modules sous un format générique. Les routeurs pourraient être représentés par des classes et la topologie par des liens entre les classes. L'approche pourrait se baser sur des modèles de description topologiques déjà existants et présentés dans [104]. Il est aussi possible de faire intervenir des mécanismes de découverte et de construction automatique de la topologie [105].

En ce qui concerne la description des applications gérées par TUNe (le Software Description Diagram SDD), une possible extension devrait permettre de spécifier des besoins

en qualité de service de manière générique, par exemple avec un nouveau type de classes utilisant des modèles issus de spécifications détaillées existantes [106]. Ces classes pourraient ensuite être associées aux différentes applications décrites par le SDD.

Dans notre approche, l'optimisation s'effectue avec un placement des applications qui est prédéfini et chaque application a un seul noeud émetteur et un seul noeud récepteur. Une possible amélioration intègrerait le calcul optimal du placement des applications en fonction non seulement de leur besoin de qualité de service, mais aussi du nombre de récepteurs ou d'émetteurs et de leur emplacement géographique. Ce placement ferait intervenir l'extension de la description matérielle avec la possibilité de spécifier des contraintes technologiques (type de réseau, débit...).

Un des aspects qui ouvre une grande perspective est le rajout de la dynamicité des placements des applications au cours de leur exécution. En effet, les modifications structurelles lors de rajouts ou de suppressions d'applications pourrait engendrer une reconfiguration globale à deux niveaux : applicatif et matériel. L'architecture logicielle pourrait alors faire migrer certains processus et l'architecture matérielle pourrait changer la topologie réseau (extinction des routeurs et des liens) pour parvenir au compromis performance-consommation électrique défini par la politique de gestion de haut niveau. Pour ce faire, une possible intégration avec des sondes génériques SNMP [1] pourrait faire remonter des métriques de performance concernant la partie matérielle. Ces métriques seraient utilisées pour un calcul optimal de l'objectif en fonction des conditions réelles observées sur le réseau. De plus, les applications ayant défini un besoin en qualité de service n'utilisent pas forcément tous les critères (débit, délai, gigue) à leur maximum à tout instant. Des reconfigurations matérielles pourraient donc avoir lieu à partir ces mesures réelles des métriques en cours d'exécution, comme la redirection ou l'agrégation de flux réseaux.

Toutes ces reconfigurations matérielles ont néanmoins un prérequis : la création d'actionneurs génériques pour les équipements matériels. Notre approche actuelle utilise le concept d'encapsulation des éléments logiciels afin d'homogénéiser les interfaces de gestion propriétaires. Ce mécanisme pourrait être transplanté au niveau des interfaces de gestion matérielles propriétaires. Les différentes actions de configuration ou de reconfiguration pourraient être exprimées en utilisant la même approche que pour la description des "wrappers" logiciels (le Software Wrapper Description Language SWDL). Ainsi, un nouveau langage de description des "wrappers" matériels pourrait être défini (le Hardware Wrapper Description Language HWDL). Les différentes méthodes seraient alors utilisables directement dans les PDD de la même manière pour les appels de méthode du SWDL.

7.2.2 Gestion avancée des événements

Dans notre approche, nous avons vu deux types de gestion d'événements. Dans TUNe, tous les événements sont détectés par des sondes génériques ou spécialisées. Celles-ci renvoient ensuite une notification à TUNe qui peut alors exécuter la politique de gestion correspondante. Dans XSStreaMWare, les événements sont détectés par des logiciels propriétaires aux capteurs gérés. Ils sont ensuite transformés en événements génériques et individuellement traités par les différents services que nous avons introduits à tous les

1. Simple Network Management Protocol

niveaux hiérarchiques du middleware. De nombreux points peuvent être améliorés concernant la gestion événementielle dans nos approches. Tout d'abord, l'aspect hiérarchique utilisé par XSStreaMWare n'est pas présent dans TUNe. En effet, TUNe comporte un seul gestionnaire centralisé, ce qui limite la scalabilité de l'approche. Dans un premier temps et pour palier à la scalabilité de TUNe centralisé, nous avons introduit une agrégation possible des événements au niveau des sondes. Celles-ci renvoient ensuite une notification à TUNe après un certain temps d'agrégation, ce qui permet de construite des listes d'éléments dont sont issus les événements agrégés au sein d'une seule notification. Dans un futur proche, il faudrait introduire ces mécanismes d'agrégation dans une hiérarchique de TUNe. D'autres travaux menés en parallèle de cette thèse tentent en effet de déployer plusieurs TUNe de manière automatique afin de décentraliser l'approche de gestion autonomique [107]. Ainsi, chaque TUNe serait en mesure de gérer de manière autonomique une région qui lui est propre et pourrait éventuellement transformer (par exemple, par agrégation) et transmettre les événements à un TUNe de plus haut niveau (ou qui s'occupe d'une autre région).

Un autre aspect de la gestion événementielle concerne la prioritisation des événements. A l'heure actuelle, tous les événements sont traités de manière égale. Les notifications reçues par TUNe sont stockées dans une file d'attente avant d'être traités de manière séquentielle. Une possibilité serait de définir des priorités différentes entre les notifications, par exemple en fonction de critères comme la gravité de l'événement dont est issue la notification, sa position géographique, le nombre d'éléments du système impactés, si elle représente une panne matérielle ou logicielle etc... Ces priorités pourraient aussi être définies par les administrateurs humains. Par exemple pour une architecture DIET, la panne d'un LA est à priori plus importante à réparer que la panne d'un SED de plus bas niveau. De même, la panne d'un serveur de noms OMNI serait critique car les éléments de DIET ne pourraient plus communiquer. Il serait envisageable d'introduire un langage de description des événements et des priorités entre les événements afin que l'utilisateur puisse les spécifier en fonction du domaine applicatif.

Enfin, le dernier aspect à améliorer concerne une éventuelle anticipation des événements à venir. Un calcul automatique pourrait prédire l'évolution future de la charge ou du nombre de certains éléments, par exemple en fonction d'un historique passé ou d'une modélisation mathématique du système. Une adaptation plus fine pourrait ensuite avoir lieu à l'aide d'un apprentissage. Cet apprentissage prendrait en compte l'historique des actions menées précédemment pour les ajuster plus précisément face à une montée en charge ou à une panne qui a déjà été traitée auparavant.

7.2.3 Vers une gestion autonomique complète des "datacenters"

A plus long terme, les différents travaux ont pour but de fournir éventuellement une gestion complète et autonomique de plusieurs "datacenters". L'idée est de déployer un centre de décision autonomique permettant de gérer les "datacenters". Un centre serait composé d'une hiérarchie de modules qui pourrait s'occuper de la gestion d'un "datacenter" et pourrait communiquer avec les autres centres. Ainsi et en fonction de contrats passés entre les "datacenters", une possible migration d'applications pourrait avoir lieu entre les "datacenters". De plus, lors de la maintenance d'un "datacenter", celui-ci pourrait migrer les différentes applications avant d'éteindre tous les équipements (machine,

réseau). Une gestion plus fine pourrait aussi faire intervenir une gestion des adresses IP, des VLANs ou VPNs et une adaptation dynamique des pare-feux ou des proxys en utilisant les mécanismes de reconfiguration générique du matériel (reconfiguration des routeurs, des switchs et des serveurs spécialisés).

Le domaine de recherche porterait alors sur la définition de contrats et de politiques de gestion de haut niveau entre les différents "datacenters". Ce domaine est à mettre en relation avec le domaine émergent du "cloud computing", permettant aux utilisateurs de porter leurs données et la gestion de leurs applications au niveau de plusieurs "datacenters". Ces différentes données et applications sont alors disponibles depuis plusieurs plateformes (mobile : sur les téléphones intelligents, au bureau avec la synchronisation des serveurs d'email, de calendriers ou à la maison sur des appareils domestiques comme les consoles de jeux, les média-centers ou la télévision...). Une première vision de cette approche à long terme est donnée par l'article [108].

Annexe A

Fractal, Deployware et Jade

Sommaire

A.1 Le modèle à composants Fractal 147
A.2 DeployWare 149
A.3 JADE 149

A.1 Le modèle à composants Fractal

Le modèle à composants **Fractal** [109] est basé sur les notions de composants, interfaces (avec interfaces de contrôle) et liaisons. Un composant est une entité exécutable conforme au modèle Fractal. Ce modèle distingue deux types de composants : les *composants primitifs* et les *composants composites*. Les composants primitifs encapsulent en général une unité de calcul décrite dans un langage de programmation. Les composants composites encapsulent un groupe de composants primitifs et/ou d'autres composites. Une interface est un point d'accès au composant. Une interface implémente un type d'interface générique qui spécifie les opérations supportées par cette dernière. Il existe deux catégories d'interfaces :

- les *interfaces serveur* qui sont des points d'accès acceptant des appels de méthodes. Elles correspondent donc aux services fournis par le composant.
- les *interfaces client* qui sont des points d'accès émettant des appels de méthodes. Elles correspondent aux services requis par le composant.

Pour contrôler les composants, le modèle dispose aussi d'une *partie de contrôle*. Cette partie de contrôle met à disposition les interfaces du composant et comporte des objets contrôleurs et intercepteurs. Le modèle Fractal ne contraint pas la nature des contrôleurs contenus dans la partie de contrôle. Il est ainsi possible d'exercer un contrôle adapté aux besoins sur les composants. La librairie Fractal fournit quatre contrôleurs :

- Le contrôleur d'attributs permet de modifier les attributs primitifs d'un composant. Ces attributs incluent les types primitifs (booléens, entiers etc.) et les chaînes de caractères.
- Le contrôleur de liaisons permet d'établir ou de rompre des liaisons primitives entre les interfaces. Ces liaisons sont des canaux de communication établis entre une interface client d'un composant et une ou plusieurs interfaces serveurs d'autres compo-

sants. Cette communication est uniquement possible si les interfaces (client/serveur) des composants sont liées.
– Le contrôleur de contenu pour les composites permet d'ajouter et de retrancher des sous-composants à un composant composite.
– Le contrôleur de cycle de vie permet de contrôler le cycle de vie du composant. Le cycle de vie d'un composant est représenté par un automate à deux états : *started* (le composant est dans un état démarré) et *stopped* (le composant est dans un état d'arrêt). Le contrôleur permet de passer d'un état à l'autre.

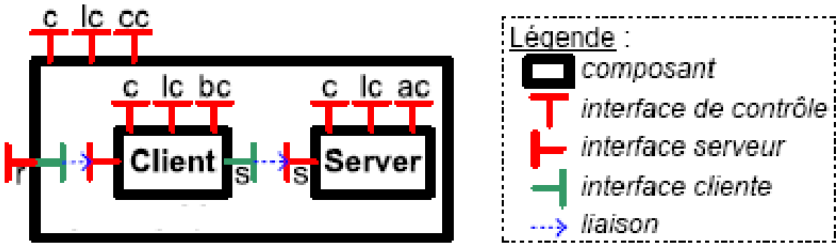


FIGURE A.1 – Modèle à composants Fractal : composant client/serveur

La figure A.1 décrit les différentes entités mises en jeu dans une architecture de type Fractal. Le rectangle noir représente le contrôle du composant alors que l'intérieur du rectangle représente le contenu du composant. Les flèches correspondent aux liaisons tandis que les formes en T attachées aux rectangles correspondent aux interfaces du composant. Les interfaces de contrôle sont représentées par des lettres telles que le contrôleur de composants (c), le contrôleur de cycle de vie (lc [1]), le contrôleur de liaisons (bc [2]), le contrôleur de contenu pour le composant composite (cc) pour le contrôleur d'attributs (ac). Cet exemple montre une application client/serveur qui se compose de deux composants primitifs : *Client* et *Server*. Ces composants sont liés par une interface *Service* nommée *s*. Celle-ci est une interface de type cliente pour le composant *Client* et elle est de type serveur pour le composant *Server*.

Le modèle à composants Fractal fournit un langage de description d'architecture appelé Fractal ADL. C'est un langage déclaratif qui permet la description de la structure de l'application construite à partir des composants Fractal. La base du langage est minimale : Fractal ADL fournit uniquement des constructions de base pour énumérer des composants, des interfaces, des liaisons et des attributs et laisse les concepteurs étendre le langage pour intégrer d'autres informations spécifiques à leur cadre d'utilisation. La syntaxe du langage Fractal ADL est basée sur XML. Pour faciliter la description d'architecture, Fractal ADL donne la possibilité de répartir dans plusieurs fichiers la description d'une architecture. Les relations entre ces fichiers peuvent prendre des formes différentes. Par exemple une définition peut en étendre une autre en y rajoutant certains éléments ou en surchargeant certaines propriétés.

Le principe de représentation de l'architecture du système sous forme de composants Fractal constitue une approche efficace et novatrice. Elle permet d'abstraire la notion d'application, pour la représenter sous forme de composants en interaction. Le choix du modèle Fractal pour les composants est doublement judicieux. Les composants Fractal

1. Lifecycle controller
2. Binding controller

sont légers en terme d'exécution et offrent des mécanismes de reconfiguration dynamique. L'architecture de composants offre un mécanisme de réflexivité pour l'administration des applications réparties.

A.2 DeployWare

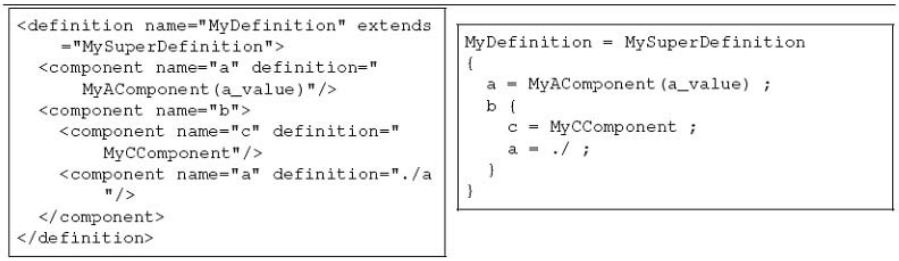


FIGURE A.2 – Transformation d'un fichier Fractal en langage DeployWare

Un exemple de langage DeployWare comparé à Fractal est donné par la figure A.2.

A.3 JADE

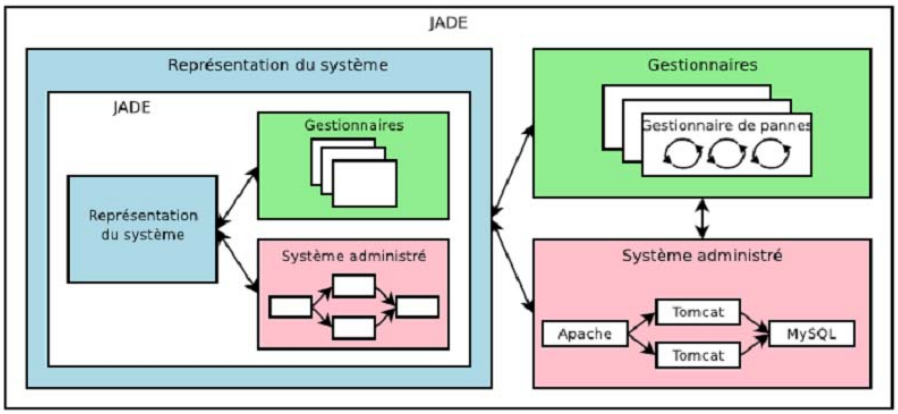


FIGURE A.3 – Architecture de JADE

JADE est une plateforme spécifiquement conçue pour une approche qui offre une gestion d'infrastructures logicielles propriétaires.

La figure A.3 montre une vision globale de JADE. Les trois grandes composantes sont identifiables dans l'architecture de JADE. Nous pouvons distinguer la représentation du système, le système administré et l'ensemble des gestionnaires. Ces gestionnaires autonomiques sont appelés **Autonomic Manager** ou **AM**. La gestion autonomique dans JADE repose sur la répartition des démons ayant la charge d'exécuter les tâches élémentaires de gestion sur le noeud physique. Ces démons sont dénommés éléments administrables (**Managed Element** ou **ME**). Ces différents éléments de gestion (**ME** et **AM**) sont des composants primitifs Fractal.

```
<definition name="HelloWorld">
   <interface name="r" role="server"
      signature="java.lang.Runnable"/>

   <component name="client">
      <interface name="r" role="server"
         signature="java.lang.Runnable"/>
      <interface name="s" role="client"
         signature="Service"/>
      <content class="ClientImpl"/>
   </component>

   <component name="server">
      <interface name="s" role="server"
         signature="Service"/>
      <content class="ServerImpl"/>
   </component>

   <binding client="this.r" server="client.r"/>

   <binding client="client.s" server="server.s"/>

</definition>
```

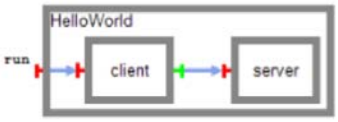


FIGURE A.4 – Architecture de JADE

La figure A.4 illustre un exemple de description d'architecture pour une application simple contenant un composant client qui appelle des opérations sur un composant serveur. La description générale de l'infrastructure matérielle et logicielle pour JADE est effectuée en utilisant le langage Fractal ADL.

Pour déployer une application avec JADE, l'administrateur décrit dans un ou plusieurs fichiers Fractal ADL, les paramètres de configuration de chaque instance d'entité logicielle et l'architecture logicielle de l'application. Un gestionnaire de déploiement centralisé transforme cette description en une architecture à composants (ME) distribuée sur les machines cibles en utilisant le mécanisme de déploiement intégré à Fractal. Ces composants vont agir localement sur les entités logicielles patrimoniales. Un dépôt des paquetages contenant les paquetages disponibles pour installation est utilisé. Un programme déclenche le processus de déploiement en appelant les interfaces appropriées sur les composants au niveau des machines cibles.

Pour gérer l'hétérogénéité des machines dans la version initiale de JADE, les auteurs proposent de disposer pour chaque cluster, d'une version de l'application qui lui est compatible. Ces différentes versions sont mises dans un dépôt (repository) et donc accessibles à partir de ce dernier. Le dépôt indique pour chaque entité logicielle et pour chaque cluster, la localisation (accessible par NFS [3]) de la livraison binaire de ce logiciel exécutable sur ce cluster. Le contenu du dépôt est stocké dans un fichier. L'administrateur doit disposer de différentes versions et tester les logiciels sur chaque cluster, en générant différentes images binaires si nécessaire et mettre à jour le fichier de dépôt en conséquence pour chaque cluster.

La plate-forme JADE a été testée pour les propriétés de "self-healing" [57] et de "self-optimizing" [110]. Pour remédier au problème de panne logicielle ou matérielle, le gestionnaire autonomique de "self-healing" s'appuie sur des sondes permettant de détecter les pannes de composants logiciels ou matériels du système géré ; et sur des actionneurs permettant de redéployer et de remplacer les composants défectueux du système pour le rendre à nouveau opérationnel.

L'architecture de JADE est distribuée. Cependant le processus de gestion est orchestré

3. Network File System

par un gestionnaire centralisé contenant la représentation à composants de l'application. Dans cette approche, les propriétés de "self-management" sont prises en compte par des gestionnaires autonomiques qui peuvent être adaptés et réutilisés pour différentes applications. Néanmoins, l'utilisation de JADE n'est pas une tâche facile. L'utilisateur doit assimiler les concepts d'architecture à composants, en plus des concepts spécifiques de Fractal, pour gérer son application. Le manque de description plus générale pour les grands systèmes engendre le problème de passage à l'échelle lors de la description de l'application. En effet le déploiement d'une architecture incluant un nombre important d'instances logicielles peut nécessiter la création d'un long fichier de description XML.

Annexe B

Utilisation de la programmation linéaire

B.1 Construction d'un Problème Linéaire (PL) avec JOpt

La construction de la définition d'un **PL** avec JOpt nécessite la création de plusieurs éléments :

- Les **variables** : la solution optimale donnera la meilleure valeur pour chacune des variables.
- Les **termes** : les termes sont un **coefficient** associé a une **variable**.
- Le **critère** : c'est un ensemble de **termes** associés par des **opérateurs** mathématiques standards (+,−,∗,/,...). Dans le cas de **PL**, il n'y a que l'opérateur + qui est utilisé.
- L'**objectif** est soit **maximiser**, soit **minimiser** un **critère**. Il doit être écrit sous la forme développée, c'est à dire sans parenthèses.
- Les **contraintes** : de même que pour l'objectif, les contraintes sont définies par un **critère**, un **opérateur** (<,>,=,...) et une **constante**. Elles sont écrites sous la forme développée.
- Les **bornes** : définissent les intervalles de recherche des valeurs pour les variables. Certaines variables peuvent être booléennes, dans ce cas le problème peut être **PLNE**.

B.2 Nommage des variables et des constantes

Dans le cas de l'utilisation d'un nombre important de variables et de contraintes pour le problème d'optimisation JOpt, il est important de définir une syntaxe pour le nommage des variables JOpt afin de retrouver facilement les valeurs des solutions une fois calculées. En effet, JOpt utilise les chaînes de caractères comme identificateur de tous ses objets. Dans notre cas nous nommons les variables comme suit :

- **x__k__i__j** $[x^k_{i,j}]$: flot pour l'application **k** du noeud **i** vers le noeud **j**.
- **zc__i** $[z^c_i]$: châssis du noeud **i** allumé = 1 ; éteint = 0.
- **zm__i__l** $[z^m_{i,l}]$: module **l** du noeud **i** allumé = 1 ; éteint = 0.

– **ze__i__j** $[z^e_{i,j}]$: port du noeud **i** vers **j** allumé = 1 ; éteint = 0.
– **b__k__n** $[b^k_n]$: pour l'application **k** : le besoin **n** est choisi = 1 ; sinon = 0.

B.3 Algorithmes pour JOpt

Algorithm 3 Création de la **Contrainte 2**

for $i \in \mathbf{E}$ **do**
  for $l \in [1, n_{M_i}]$ **do**
    for $j \in \mathbf{E}, \exists e_{i,j} \in \mathbf{E}$ **do**
      Créer (ou récupérer si elle existe déjà) la Variable "**zm__i__l**" $[z^m_{i,l}]$
      Créer (ou récupérer si elle existe déjà) la Variable "**ze__i__j**" $[z^e_{i,j}]$
      Créer le Terme **t1** ← Coefficient **1.0** & Variable "**zm__i__l**" $[z^m_{i,l}]$
      Créer le Terme **t2** ← Coefficient **−1.0** & Variable "**ze__i__j**" $[z^e_{i,j}]$
      Créer le Critère **cr** ← Terme **t1** & Terme **t2**
      Créer la Constrainte **cst** ← Critère **cr** & Opérateur ">=" & Constante **0.0**
    end for
  end for
end for

L'algorithme 3 est un exemple d'algorithme utilisé pour la création de contraintes en JOpt. Les variables sont d'abord récupérées ou créées si elles n'existent pas. Les termes sont ensuite créés en rajoutant des coefficients aux variables, puis le critère est formé par association de termes. La contrainte est finalement crée a partir du critère, d'un opérateur et d'une constante.

Algorithm 4 Forçage des x par OSPF

for $k \in [1, n_A]$ **do**
  Liste **chemins__ospf** ← calculer les chemins OSPF entre N^k_s et N^k_r
  $\mathbf{RN^k}$ ← routeurs utilisés dans **chemins__ospf**
  for $i \notin \mathbf{RN^k}$ **do**
    for $j \notin \mathbf{RN^k}, \exists e_{i,j} \in \mathbf{E}$ **do**
      Récupérer Variable "**x__k__i__j**" $[x^k_{i,j}]$
      Créer Terme **t3** ← Coefficient **1.0** & Variable "**x__k__i__j**" $[x^k_{i,j}]$
      Créer Critère **cr** ← Terme **t3**
      Créer Contrainte **cst** ← Critère **cr** & Operateur "=" & Constante **0.0**
    end for
  end for
end for

L'algorithme 4 montre comment créer des contraintes JOpt qui permettent de forcer les flots non utilisés par le routage OSPF à un débit égal à 0. Pour cela, le critère est construit de la même manière que pour l'algorithme 3, et la contrainte utilise l'opérateur "=" et la constante "0".

Annexe C

DIET et TUNe

Sommaire

C.1 SWDL pour DIET 155
C.2 PDD utilisés pour la destruction des SEDs 156
C.3 Expérience d'équilibrage de charge numéraire pour DIET . 156
C.4 Expérience d'équilibrage de charge contraint pour DIET . 157

C.1 SWDL pour DIET

```
<?xml version='1.0' encoding='ISO-8859-1' ?>

<wrapper name='sed'>

  <method name="start" key="extension.GenericUNIXMethods"
method="start_with_pid_linux" >
    <param value="$dirLocal/$progName $dirLocal/$srname-cfg $arguments $srname"/>
    <param value="LD_LIBRARY_PATH=$dirLocal"/>
    <param value="OMNIORB_CONFIG=$dirLocal/$omni.srname-cfg" />
  </method>

  <method name="configureOmni" key="extension.GenericUNIXMethods"
        method="configure_plain_text">
    <param value="$dirLocal/$omni.srname-cfg"/>
    <param value=" = "/>
    <param value="InitRef:NameService=corbaname::$omni.nodeName:$omni.port"/>
    <param value="DefaultInitRef:corbaloc::$omni.nodeName" />
  </method>
...
```

FIGURE C.1 – SWDL utilisé pour un SED

La figure C.1 montre un exemple de SWDL utilisé pour un SED de l'architecture DIET. La définition des méthodes "start" et "configureOmni" est donnée. Elles utilisent le paquetage "extension.GenericUNIXMethods". La première utilise une méthode java qui permet de lancer un commande UNIX et de récupérer son PID "start_with_pid_linux" et la deuxième "configure_plain_text" qui permet de créer un fichier de configuration en fonction des paramètres d'entrée. Les différents paramètres d'entrée utilisent la notation pointée qui permet à TUNe d'interpréter les expressions au moment de l'exécution et de remplacer celles-ci par les valeurs dynamiques.

C.2 PDD utilisés pour la destruction des SEDs

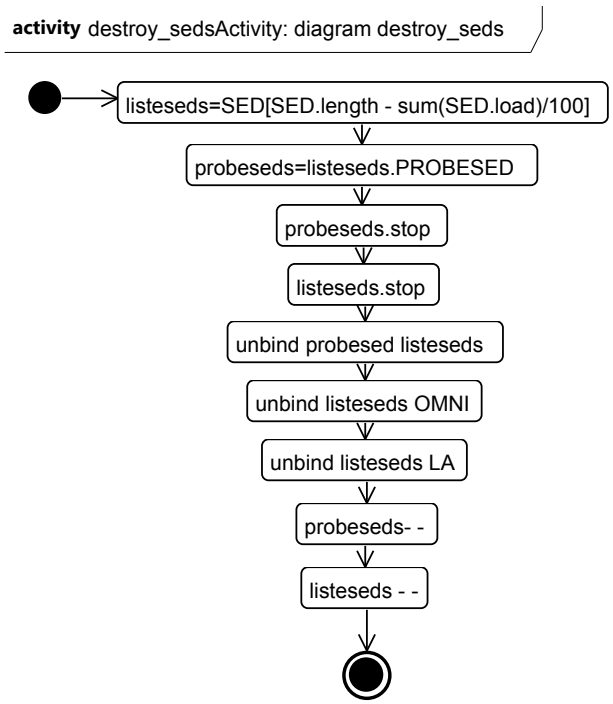


FIGURE C.2 – PDD de destruction des SEDs

La figure C.2 montre un exemple de PDD pour la destruction des SEDs.

C.3 Expérience d'équilibrage de charge numéraire pour DIET

Pour cette expérience, nous avons déployé une architecture DIET avec deux LAs (nous avons rajouté l'attribut *initial=2* pour la classe LA du SDD). Nous avons forcé le placement du premier LA (LA_1) sur une machine puissante (Intel Xeon EM64T 3GHz) et du second LA (LA_2) sur une machine moins puissante (Intel Xeon 5110 1.6 GHz). Pour cela, nous avons créé une nouvelle famille dans le HDD de DIET (figure 3.3) avec des valeurs fixes pour les deux adresses et nous avons changé l'attribut *host-family* de la classe LA du SDD.

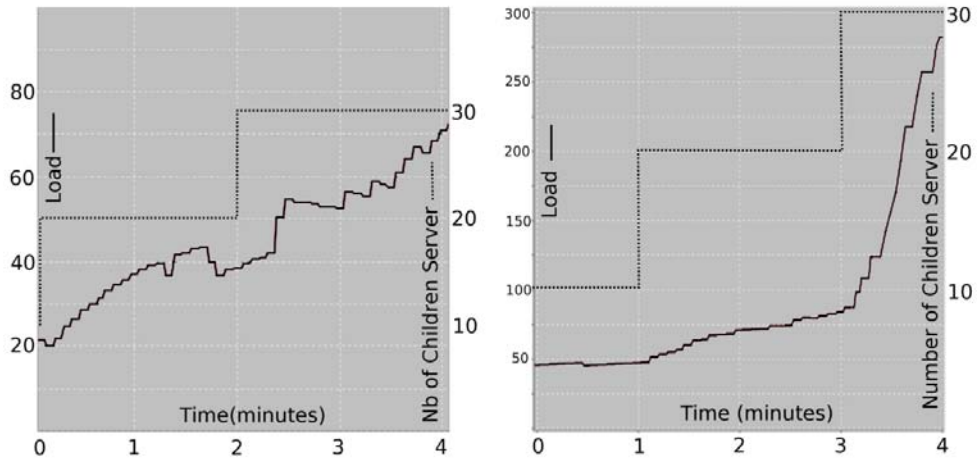


FIGURE C.3 – Equilibrage de charge numéraire (a) gauche : Local Agent 1, machine puissante, (b) droite : Local Agent 2, machine lente.

La première expérience consiste à créer dix nouvelles instances de SEDs chaque minute avec un équilibrage de charge numéraire, c'est à dire que les SEDs sont connectés au LA ayant le plus petit nombre de SEDs à sa charge. Pour cela, nous avons remplacé la deuxième action du PDD de création des instances de SEDs (figure 6.13) par *list_la = min(LA.SED)*.

Les figures C.3 (a) et C.3 (b) démontrent que l'équilibrage de charge numéraire n'est pas toujours une solution optimale. En effet, le LA placé sur la machine la plus lente n'arrive pas à absorber le taux de requête entrant (fixé à 20 requêtes par secondes) quand le nombre d'enfants SEDs atteint la valeur trente. En effet, la charge de la machine dépasse largement les 100% et explose. Au contraire, le LA placé sur la machine plus rapide se charge jusqu'à environ 70%.

Pour éviter ce cas d'explosion du système et garantir de nouveau la stabilité du système, l'idée consiste à contraindre le nombre de SEDs sous un LA en fonction de la charge de celui-ci.

C.4 Expérience d'équilibrage de charge contraint pour DIET

Cette seconde expérience, contrairement à l'expérience d'équilibrage de charge numéraire, contraint la liaison entre les nouveaux SEDs et les LAs en prenant en compte la charge CPU des LAs. Pour cela, c'est la première action du PDD des SEDs *list_la = min(LA.load)* qui se charge de choisir les LAs ayant la charge minimale CPU courante.

Les figures C.4 (a) et C.4 (b) montrent qu'en appliquant ce filtrage sur la charge des CPUs, le LA étant placé sur une machine plus puissante doit gérer plus de SEDs que celui placé sur la machine lente. En effet, les deux LAs atteignent dans les mêmes conditions d'expérimentation une charge moyenne d'environ 80%. En revanche, le premier LA gère quarante SEDs alors que le deuxième gère vingt SEDs. Ceci démontre que dans ce cas, un équilibrage de charge contraint sur la charge CPU des LAs protège d'une explosion

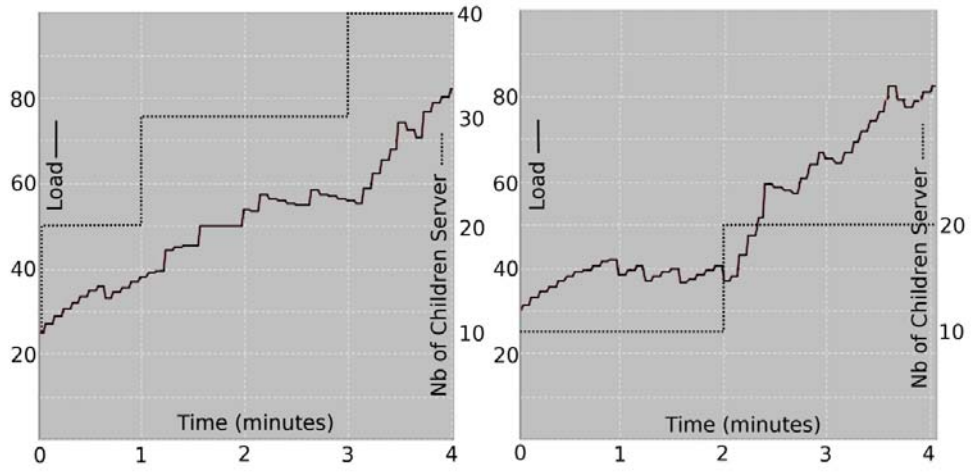


FIGURE C.4 – Equilibrage de charge contraint (a) gauche : Local Agent 1, machine puissante, (b) droite : Local Agent 2, machine lente.

de la charge de certains LAs contrairement à un équilibrage de charge numéraire.

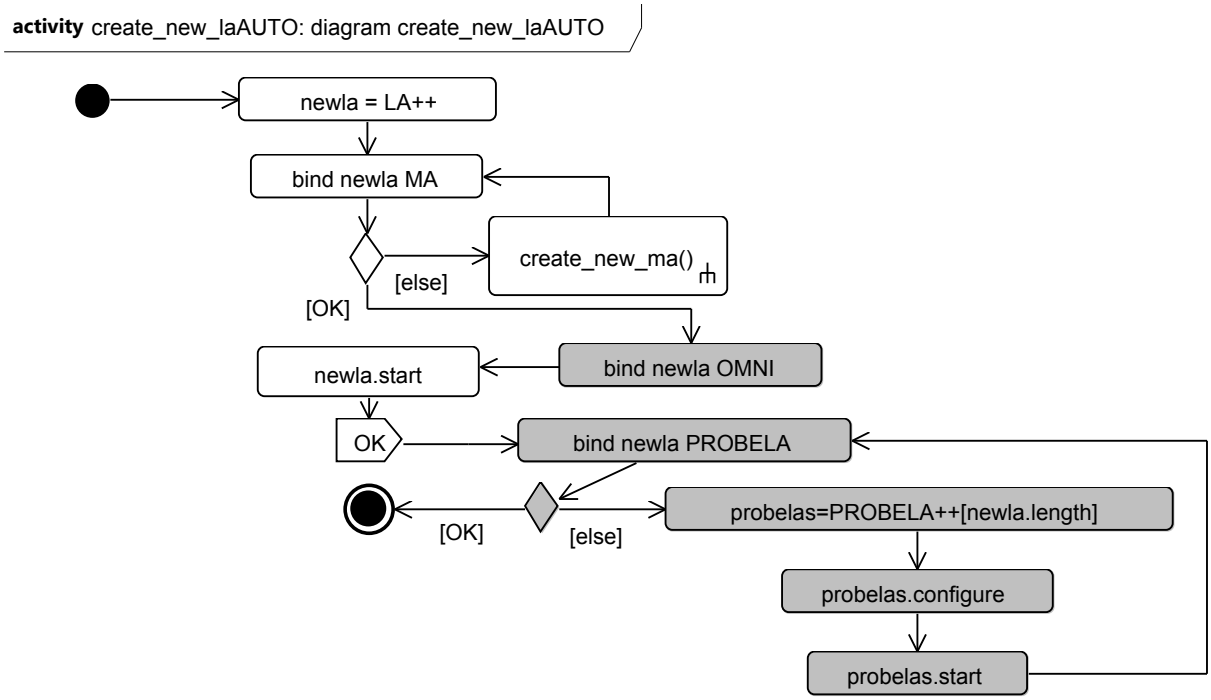


FIGURE C.5 – PDD avec les actions automatiques pour la création d'un nouveau LA

Dans le cas du dépassement de la cardinalité maximale (c'est-à-dire le nombre maximal de SEDs sous un LA : ici cinquante dans le SDD), le PDD de création de nouvelles instances de LA est appelé. De même, si le nombre de LA dépasse la cardinalité maximale du MA (le nombre maximal de LA sous un MA étant de cinq), le PDD de création de nouvelles instances de MA est appelé. Ces différents PDDs sont donnés en figure C.5 et C.6 avec les actions automatisée pour la gestion des cohérences en gris.

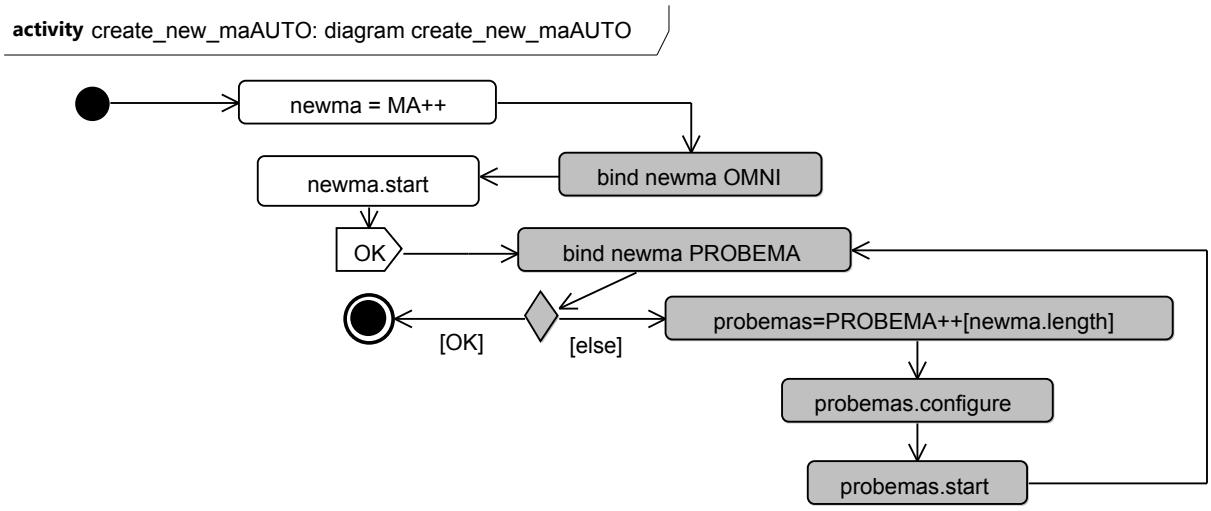


FIGURE C.6 – PDD avec les actions automatiques pour la création d'un nouveau MA

Annexe D

Expériences de mesure de consommation électrique des routeurs de la plateforme Grid'Mip

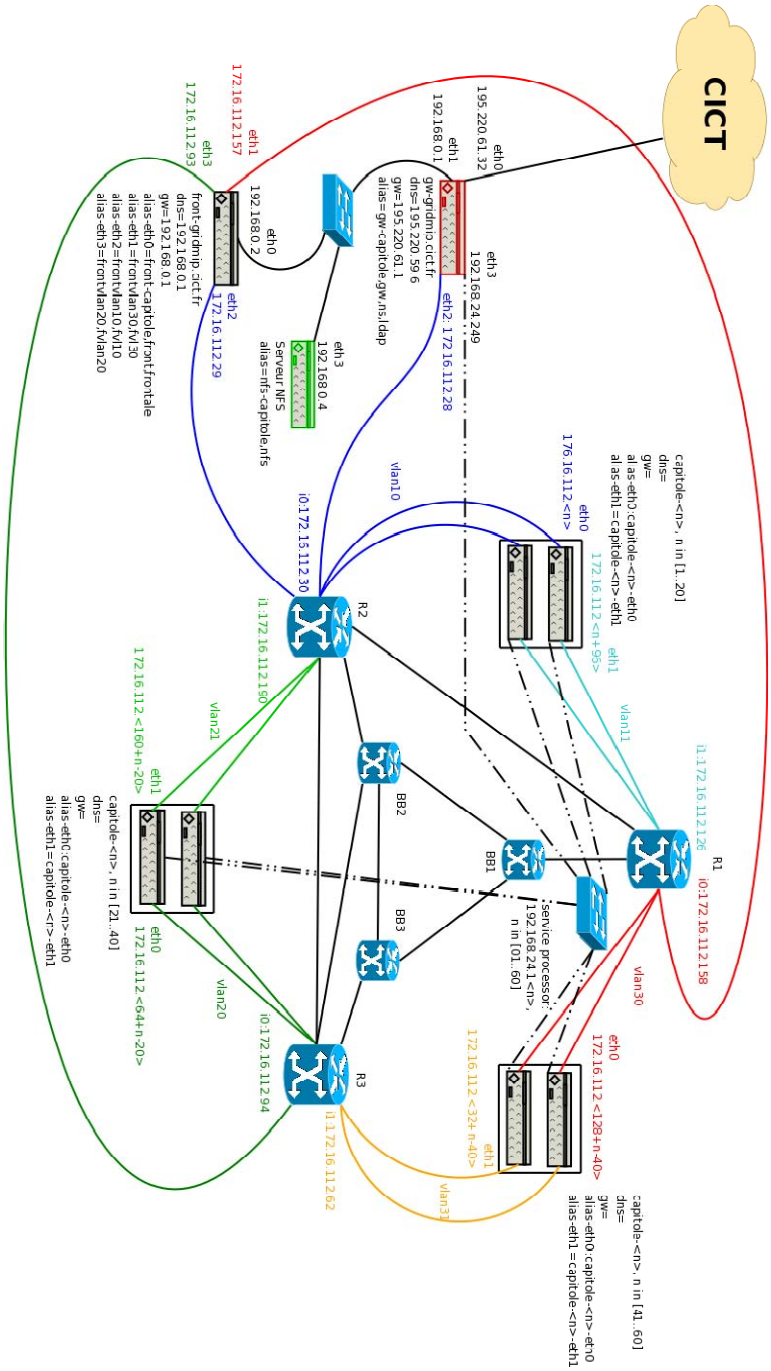
Pour les mesures de consommation électriques de la plateforme Grid'Mip, nous utilisons des wattmètres branchés entre les prises électriques des routeurs et leur module d'alimentation. Nous récupérons les valeurs rafraichies toutes les secondes à l'aide d'une connexion Bluetooth. Les wattmètres diffusent en Bluetooth toutes les secondes la valeur de la consommation mesurée.

La connexion à la plateforme à distance s'effectue avec la commande:

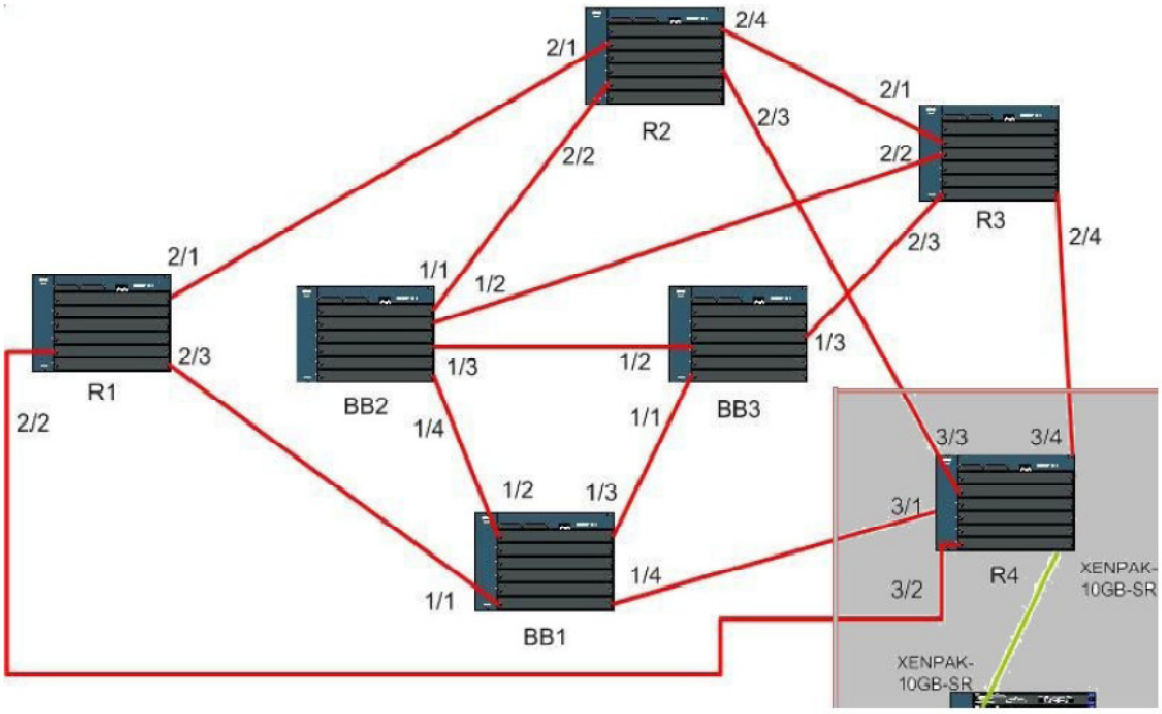
```
ssh -X <utilisateur>@gw-gridmip.cict.fr
```

L'état du cluster est disponible à cette adresse :
https://gw-gridmip.cict.fr/

Voici la topologie globale détaillée de Grid'MIP :



Voici un schéma simplifié de la topologie avec le nom de routeurs et des interfaces réseaux :



Les commandes à utiliser pour se connecter aux interfaces d'administration des routeurs, une fois la connexion à la plateforme effectuée sont les suivantes :

```
R1 ssh admin@172.16.112.158
R2 ssh admin@172.16.112.30
R3 ssh admin@172.16.112.94
BB1 ssh admin@192.168.30.25
BB2 ssh admin@192.168.30.41
BB3 ssh admin@192.168.30.57
```

Pour sauver la configuration des routeurs, on utilise un serveur tftp :

172.16.112.157 pour R1 et BB1

172.16.112.29 pour R2 et BB2

172.16.112.93 pour R3 et BB3

Répertoire de sauvegarde des configurations sur le serveur tftp est : /usr/tftpboot/routers/

Pour reloader une configuration, une fois connecté à un routeur, faire :

```
Copy run tftp

Address or name of remote host [] ? 172.16.112.xxx

Destination filename [r1-config] ? /routers/<nom du fichier>
```

Détails des modules pour BB1 :

```
################################################
#         Universite Paul-Sabatier             #
#              Projet GridMip                  #
#               Routeur  BB1                   #
#      Access for authorized users only        #
#         All accesses are logged              #
################################################
admin@192.168.30.25's password:

BB1>show module
Mod Ports Card Type                                Model              Serial No.
--- ----- ---------------------------------------- ------------------ -----------
  1    4  CEF720 4 port 10-Gigabit Ethernet      WS-X6704-10GE      SAL1104F5JC
  5    2  Supervisor Engine 720 (Active)         WS-SUP720-3B       SAL10499X29

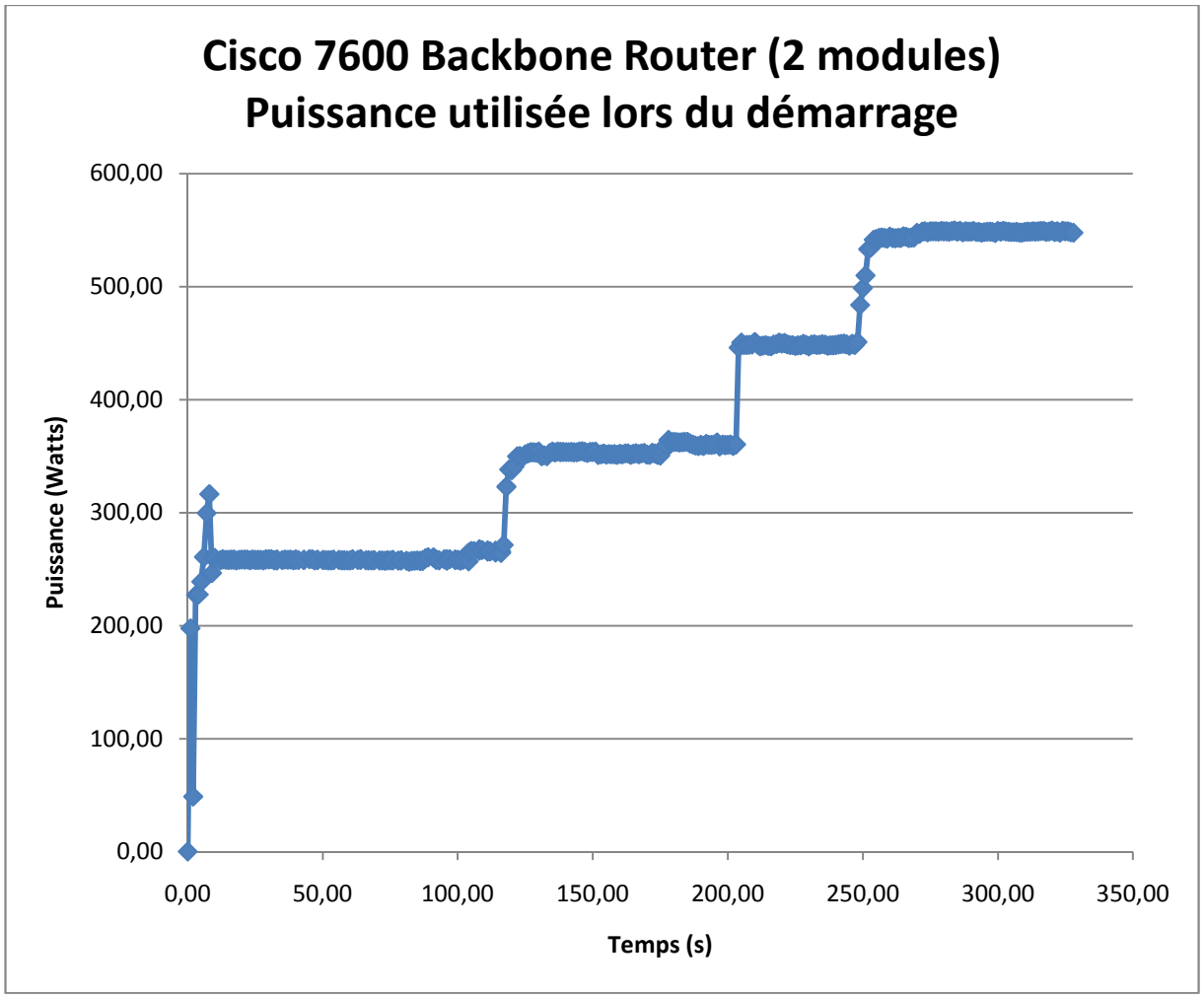
Mod MAC addresses                       Hw    Fw           Sw           Status
--- ----------------------------------- ------ ------------ ------------ -------
  1  001a.a10e.812c to 001a.a10e.812f   2.5   12.2(14r)S5  12.2(33)SRB5 Ok
  5  0017.9444.255c to 0017.9444.255f   5.3   8.4(2)       12.2(33)SRB5 Ok

Mod  Sub-Module                  Model              Serial       Hw     Status
---- --------------------------- ------------------ ----------- ------- -------
  1  Centralized Forwarding Card WS-F6700-CFC       SAL1117MKVC  3.1    Ok
  5  Policy Feature Card 3       WS-F6K-PFC3B       SAL10478LRG  2.3    Ok
  5  MSFC3 Daughterboard         WS-SUP720          SAL10499XC1  2.6    Ok

Mod  Online Diag Status
---- --------------------
  1  Pass
  5  Pass
BB1>
```

Détail de demandes énergétiques théoriques :

```
BB1>show power
system power redundancy mode = redundant
system power redundancy operationally = non-redundant
system power total =     2669.10 Watts (63.55 Amps @ 42V)
system power used =      1039.92 Watts (24.76 Amps @ 42V)
system power available = 1629.18 Watts (38.79 Amps @ 42V)
                      Power-Capacity PS-Fan Output Oper
PS   Type              Watts   A @42V Status Status State
---- ----------------- ------- ------ ------ ------ -----
1    PWR-2700-AC       2669.10 63.55  OK     OK     on
2    none
                      Pwr-Allocated  Oper
Fan  Type              Watts   A @42V State
---- ----------------- ------- ------ -----
1    FAN-MOD-6HS        180.18  4.29  OK
                      Pwr-Requested  Pwr-Allocated  Admin Oper
Slot Card-Type        Watts   A @42V Watts   A @42V State State
---- ----------------- ------- ------ ------- ------ ----- -----
1    WS-X6704-10GE      295.26  7.03   295.26  7.03  on    on
5    WS-SUP720-3B       282.24  6.72   282.24  6.72  on    on
6    (Redundant Sup)      -     -      282.24  6.72  -     -
BB1>
```



Pour ce premier test, nous avons débranché un routeur Cisco de cœur (backbone BB1) de la plateforme Grid'Mip, inséré un Wattmètre (Plogg) sur sa prise et rebranché le routeur dessus. Ce graphique donne la puissance consommée par le routeur au démarrage. L'allumage complet met environ 4min30s. Les paliers correspondent à des consommations différentes lors de l'allumage des modules et de leur initialisation. Ce routeur en comporte deux: un pour la console d'administration et un pour les 4 liaisons 10Gb/s modules. On observe 4 paliers mais on ne peut pas déterminer avec précision à quoi ils correspondent. La consommation du routeur BB1 une fois allumé est en moyenne de 548,58 watts.

Détails des modules pour R2 :

```
##########################################
#        Universite Paul-Sabatier        #
#             Projet GridMip             #
#               Routeur R2               #
#    Access for authorized users only    #
#        All accesses are logged         #
##########################################
admin@172.16.112.30's password:

R2>show module
Mod Ports Card Type                                Model              Serial No.
--- ----- ---------------------------------------- ------------------ -----------
  1   48  CEF720 48 port 10/100/1000mb Ethernet  WS-X6748-GE-TX     SAL1117MRZ3
  2    4  CEF720 4 port 10-Gigabit Ethernet      WS-X6704-10GE      SAL1103E8U1
  5    2  Supervisor Engine 720 (Active)         WS-SUP720-3B       SAL10499WZY

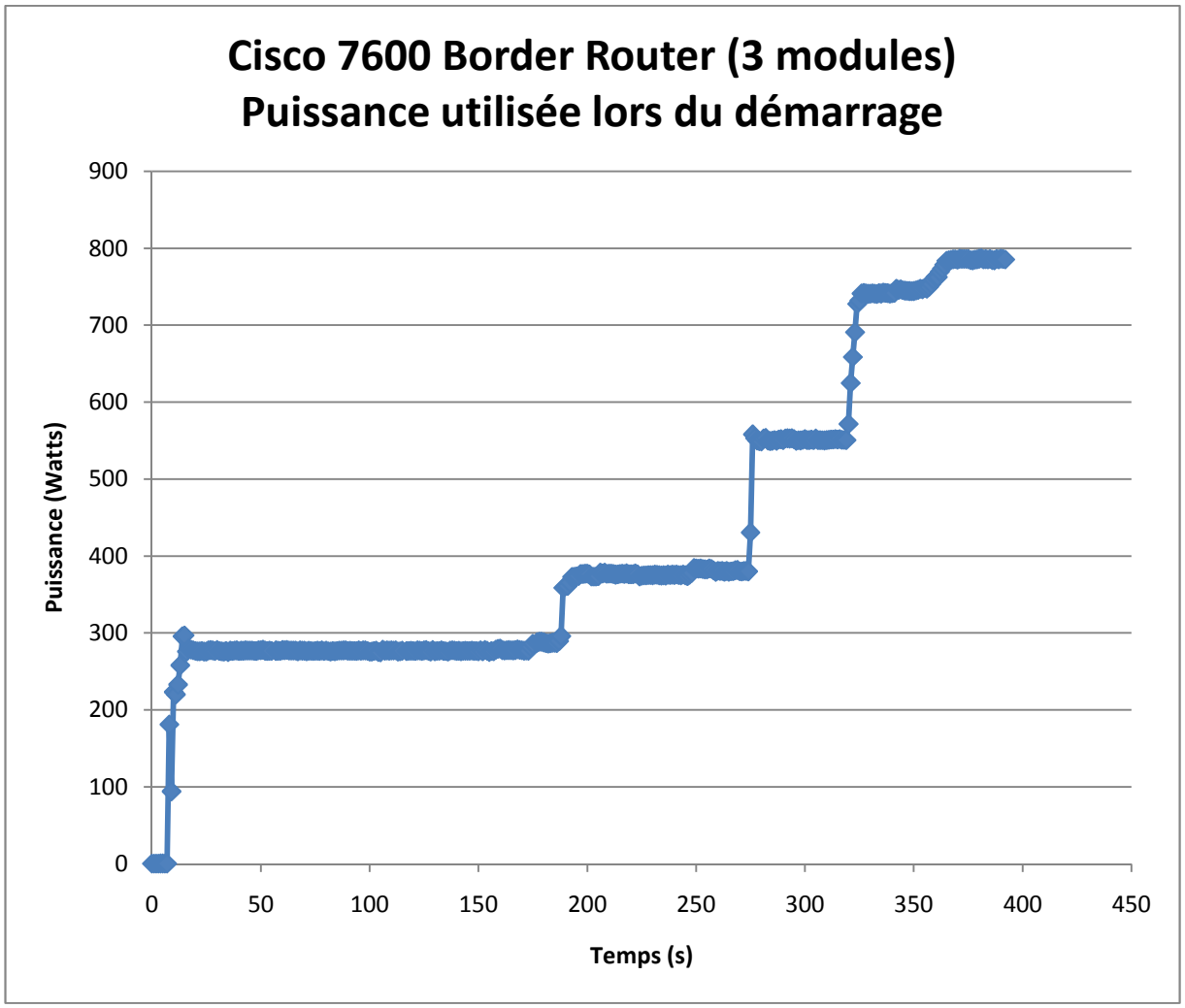
Mod MAC addresses                       Hw     Fw           Sw           Status
--- ----------------------------------- ------ ------------ ------------ -------
  1  001a.a10e.dde4 to 001a.a10e.de13   2.5    12.2(14r)S5  12.2(33)SRB5 Ok
  2  001a.6c97.b01c to 001a.6c97.b01f   2.5    12.2(14r)S5  12.2(33)SRB5 Ok
  5  0017.9568.55f0 to 0017.9568.55f3   5.3    8.4(2)       12.2(33)SRB5 Ok

Mod  Sub-Module                  Model              Serial       Hw     Status
---- --------------------------- ------------------ ----------- ------- -------
  1  Centralized Forwarding Card WS-F6700-CFC       SAL1117MPAL  3.1    Ok
  2  Centralized Forwarding Card WS-F6700-CFC       SAL1117MPEG  3.1    Ok
  5  Policy Feature Card 3       WS-F6K-PFC3B       SAL10489HTK  2.3    Ok
  5  MSFC3 Daughterboard         WS-SUP720          SAL10499X6V  2.6    Ok

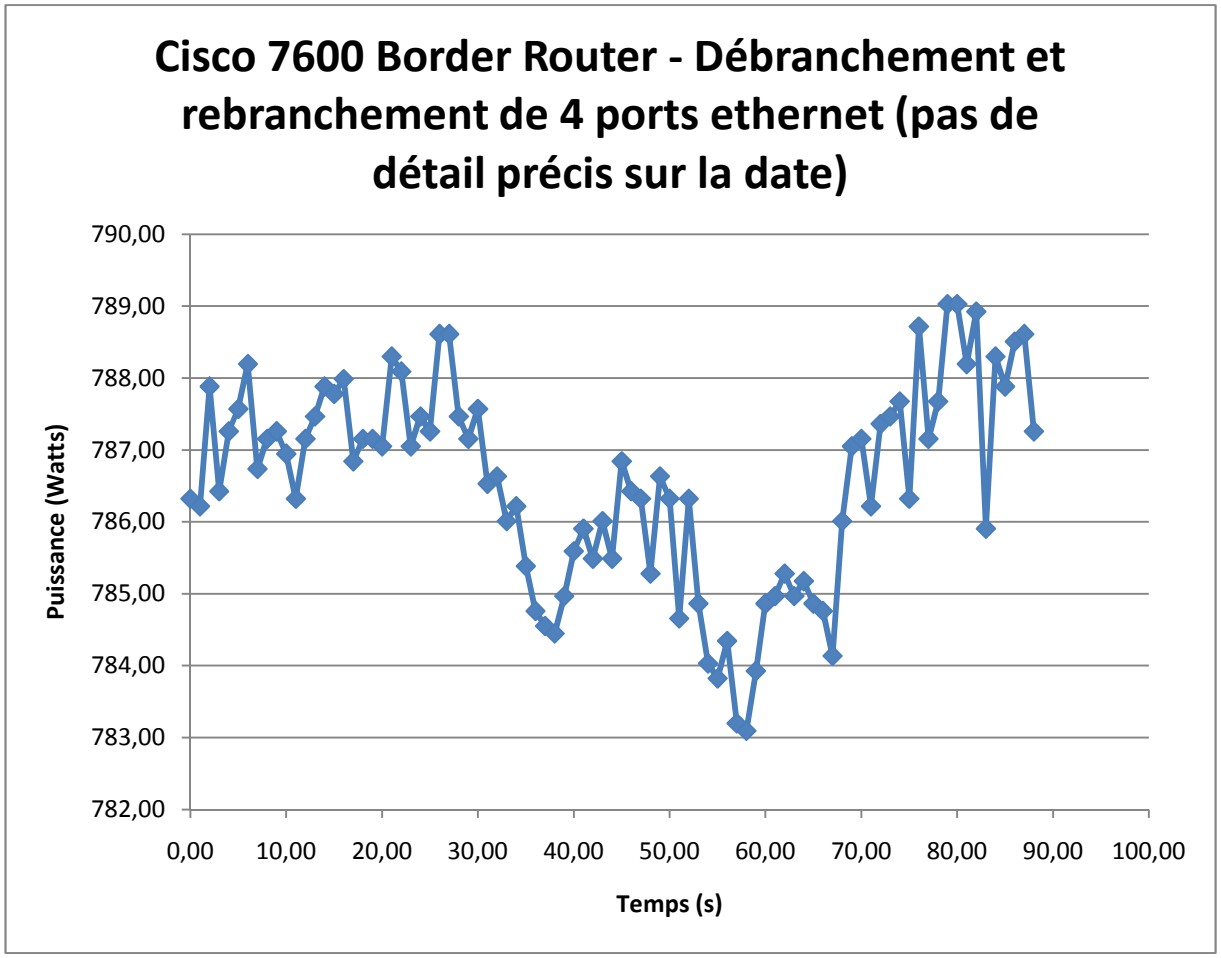
Mod  Online Diag Status
---- -------------------
  1  Pass
  2  Pass
  5  Pass
R2>
```

Détail de demandes énergétiques théoriques :

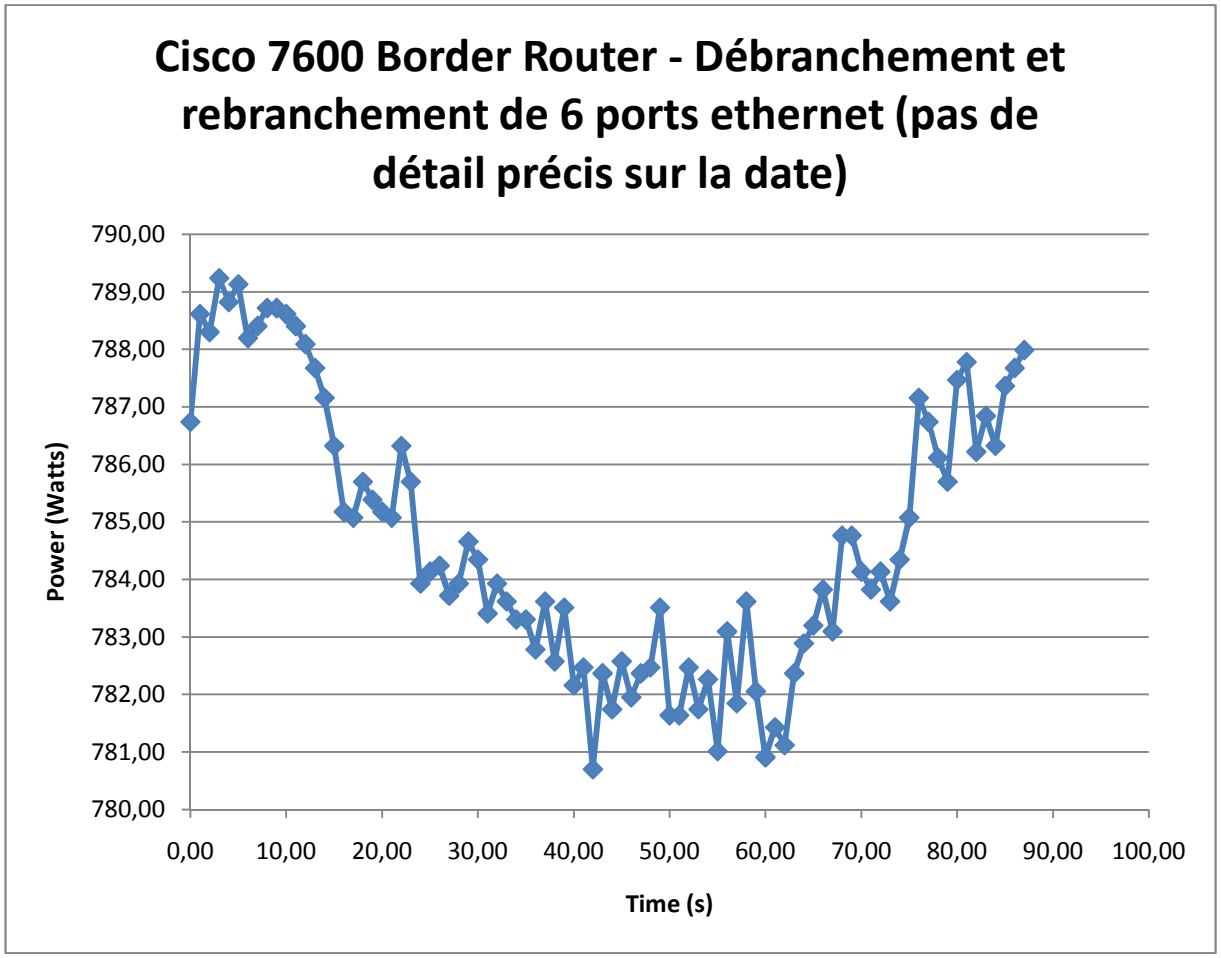
```
R2>show power
system power redundancy mode = redundant
system power redundancy operationally = non-redundant
system power total =     2669.10 Watts (63.55 Amps @ 42V)
system power used =      1365.42 Watts (32.51 Amps @ 42V)
system power available = 1303.68 Watts (31.04 Amps @ 42V)
                       Power-Capacity PS-Fan Output Oper
PS   Type               Watts   A @42V Status Status State
---- ------------------ ------- ------ ------ ------ -----
1    PWR-2700-AC        2669.10 63.55  OK     OK     on
2    none
                       Pwr-Allocated  Oper
Fan  Type               Watts   A @42V State
---- ------------------ ------- ------ -----
1    FAN-MOD-6HS         180.18  4.29  OK
                       Pwr-Requested  Pwr-Allocated  Admin Oper
Slot Card-Type         Watts   A @42V Watts   A @42V State State
---- ------------------ ------- ------ ------- ------ ----- -----
1    WS-X6748-GE-TX      325.50  7.75   325.50  7.75  on    on
2    WS-X6704-10GE       295.26  7.03   295.26  7.03  on    on
5    WS-SUP720-3B        282.24  6.72   282.24  6.72  on    on
6    (Redundant Sup)       -     -      282.24  6.72  -     -
R2>
```



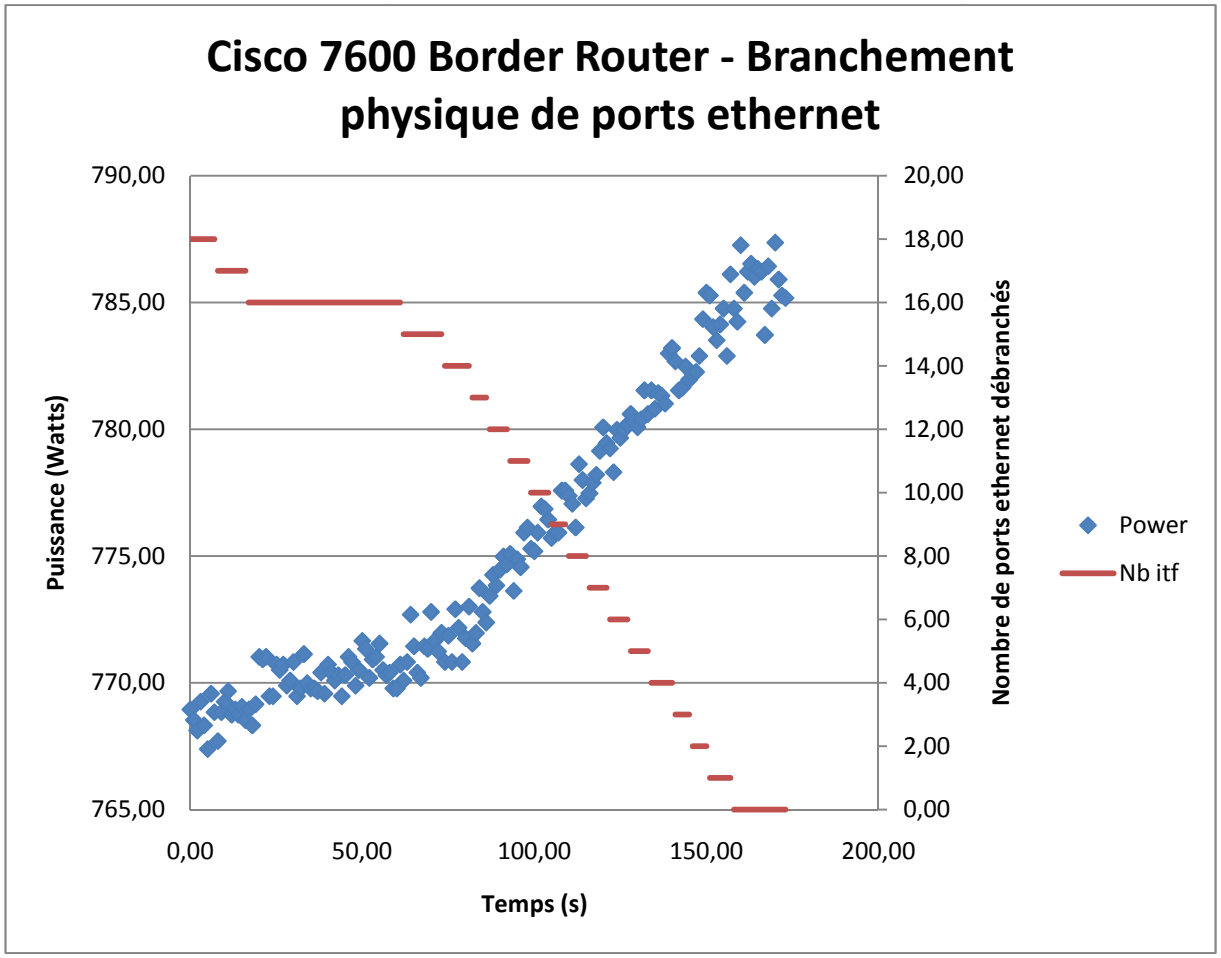
Pour ce test, nous avons mesuré de la même manière que pour le premier test la consommation d'un routeur Cisco dit « de bordure » (Rx). Ce graphique donne la puissance consommée par le routeur R2 au démarrage. Le premier palier est légèrement supérieur à celui du routeur de cœur (backbone) à environ 280 Watts au lieu de 250 Watts. Ce routeur comporte un module supplémentaire avec 48 ports ethernet. On voit une légère montée dans ce dernier palier qui pourrait correspondre à l'allumage séquentiel de tous les ports ethernet de 356s à 365s. Le routeur R2 consomme en moyenne 785,64 watts une fois allumé.



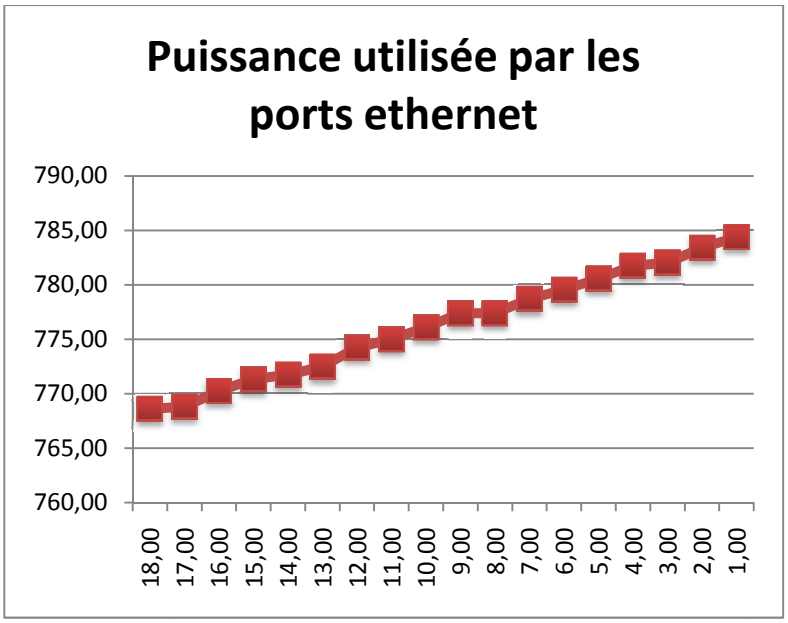
Pour ce test, on débranche 4 câbles ethernet du côté des ordinateurs reliés au routeur R2, les câbles restent branchés sur le module ethernet du routeur, mais ne sont plus actifs électriquement parlant, puis on les rebranche. Ce premier test confirme que la consommation baisse légèrement.

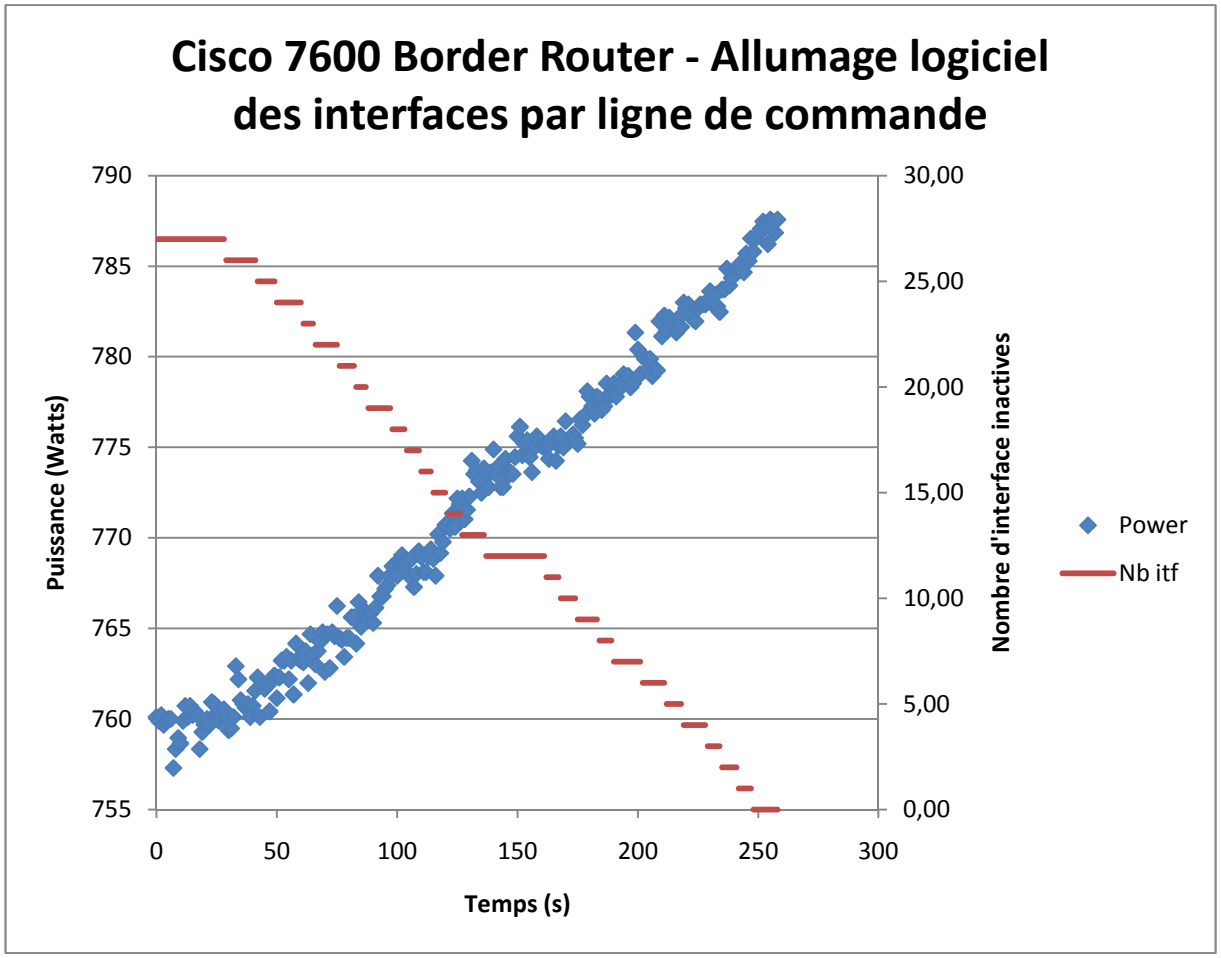


Même test que précédemment, mais cette fois en débranchant 6 câbles ethernet utilisés et actifs directement du module ethernet du routeur R2, puis en les rebranchant. De même, cela fait légèrement baisser la consommation.



Pour ce test, on branche physiquement 18 ordinateurs supplémentaires sur le module ethernet du routeur R2. L'axe vertical de droite représente le nombre d'interfaces qui ne sont pas encore branchées. La consommation mesurée est en moyenne de 0.94 watt et est linéaire par rapport au nombre de ports allumés comme le montre le graphique suivant (moyennes des paliers du graphique ci-dessus reportées sur un axe) :





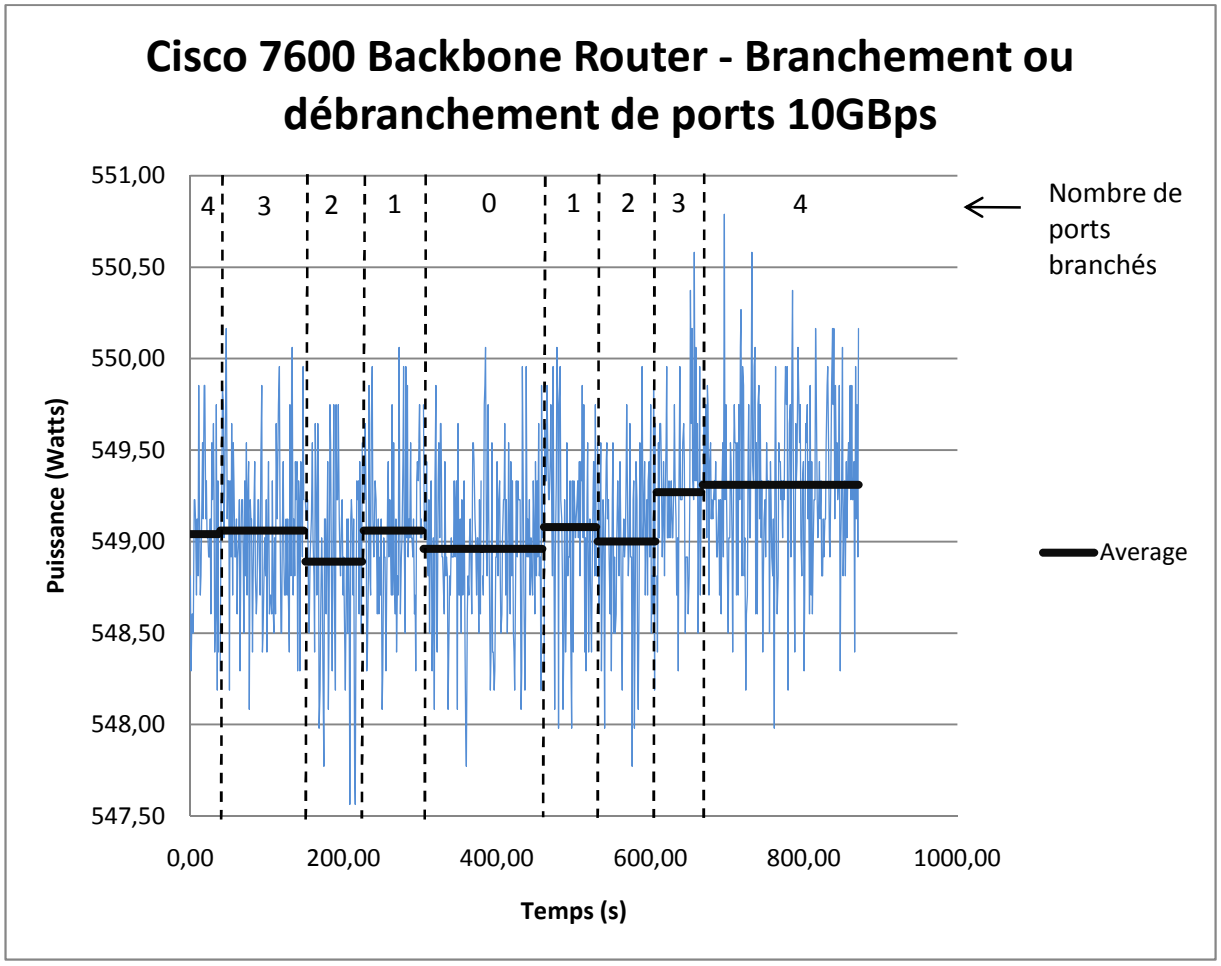
Pour ce test, on allume logiciellement 28 ports du module ethernet du routeur R2. De même que précédemment, la consommation est en moyenne de 0,94 watt par port allumé logiciellement (même méthode de calcul que précédemment : moyenne par port puis moyenne globale).

Commandes utilisées:

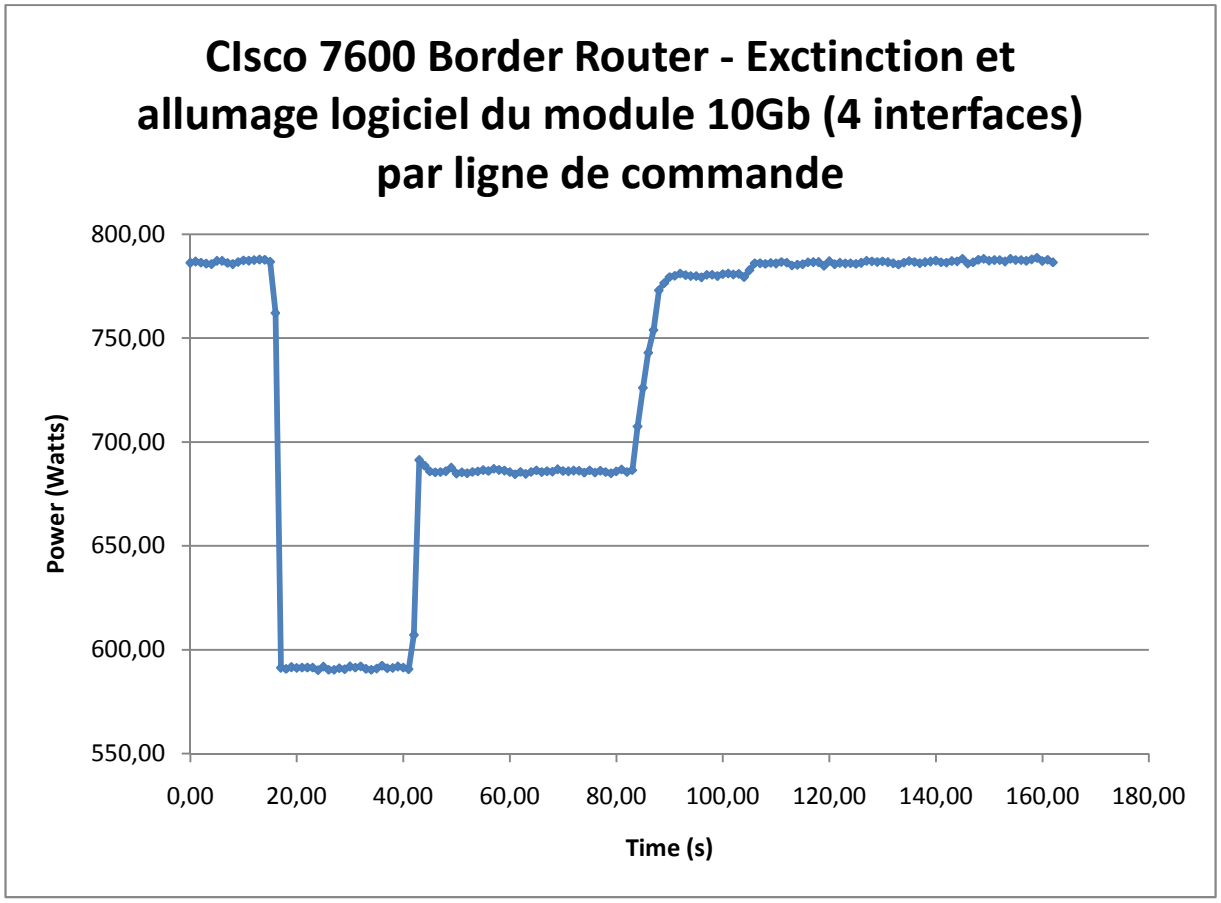
```
R2>enable
Password:
R2# configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
R2(config)#interface GigabitEthernet 1/48
R2(config)#no shutdown
```

Il suffit de remplacer 1/48 par les bons numéros d'interfaces (exemple avec une interface « down »):

```
R2#show interfaces GigabitEthernet ?
  <1-6>  GigabitEthernet interface number
R2#show interfaces GigabitEthernet 1?
/
R2#show interfaces GigabitEthernet 1/?
  <1-48>  GigabitEthernet interface number
R2#show interfaces GigabitEthernet 1/48?
.  <1-48>
R2#show interfaces GigabitEthernet 1/48.?
  <0-4294967295>  GigabitEthernet interface number
R2#show interfaces GigabitEthernet 1/48.0
GigabitEthernet1/48 is administratively down, line protocol is down (disabled)
  Hardware is C6k 1000Mb 802.3, address is 001a.3029.de00 (bia 001a.3029.de00)
  MTU 1500 bytes, BW 1000000 Kbit, DLY 10 usec,
     reliability 255/255, txload 1/255, rxload 1/255
```



Pour ce test, on débranche physiquement et un par un les 4 ports 10Gb/s du routeur BB1, puis on les rebranche. On remarque que, contrairement aux ports ethernet 1Gb/s, la consommation reste la même à 0,25 watts près.



Pour ce test, on arrête logiciellement avec la commande no power le module 4*10Gb/s de R2 ayant les 4 ports allumés. On le rallume avec la commande power 20 secondes plus tard. Cela fait gagner 195,57 watts (différence des moyennes de consommation quand le module est allumé et quand il est éteint). Le module se rallume en deux phases, une première de 40s à 80s qui fait consommer environ 100 Watts de plus et qui dure environ 40 secondes, et une deuxième qui fait consommer encore 100 Watts de plus. On remarque une petite remontée de quelques Watts à 100s, c'est-à-dire à 20 secondes dans la deuxième phase.

Commandes utilisées :

```
R2>enable
Password:
R2#configure terminal
R2(config)#no power enable module <numero module>
```

Sur d'autres version IOS Cisco, la commande peut varier, par exemple:

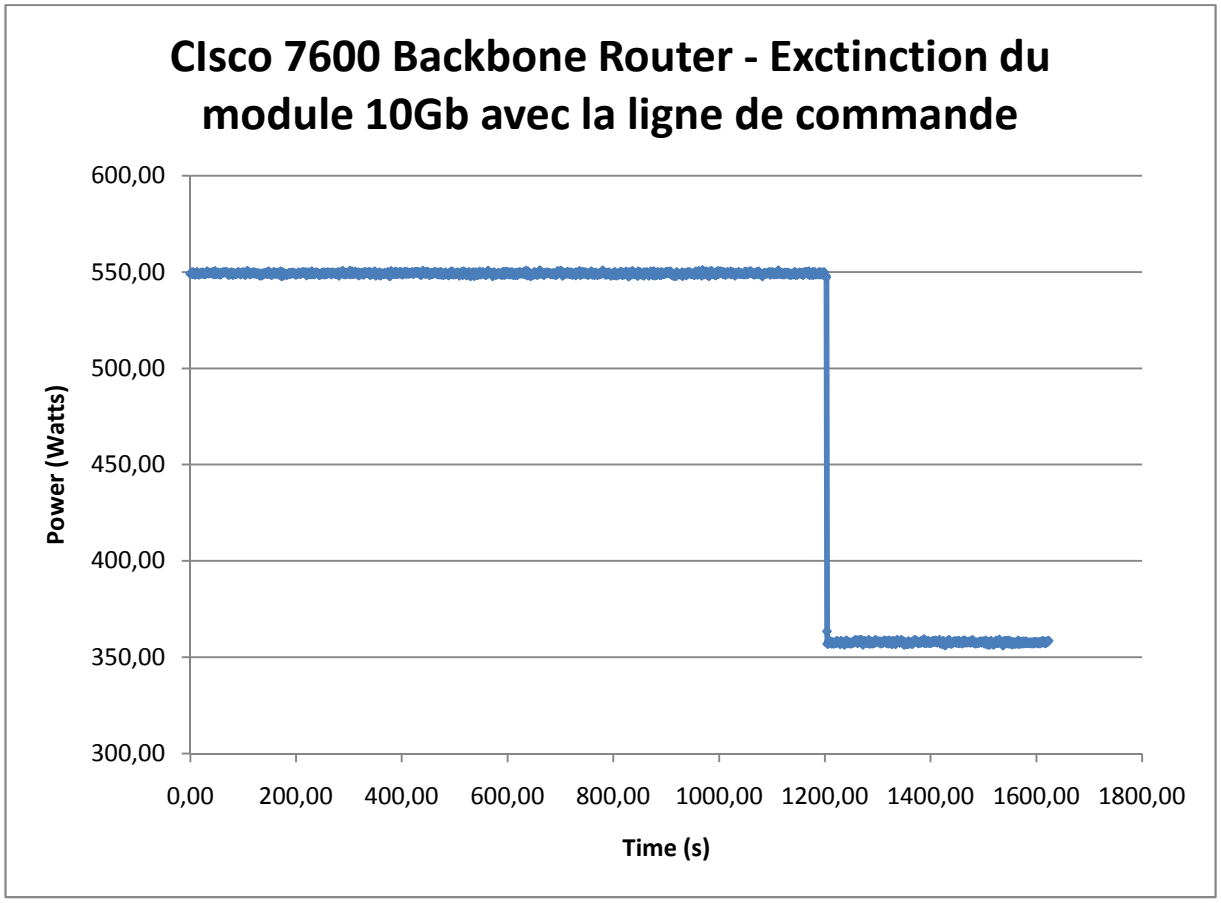
```
#hw-module module <numero> shutdown
```

Parfois on a le choix qu'entre:

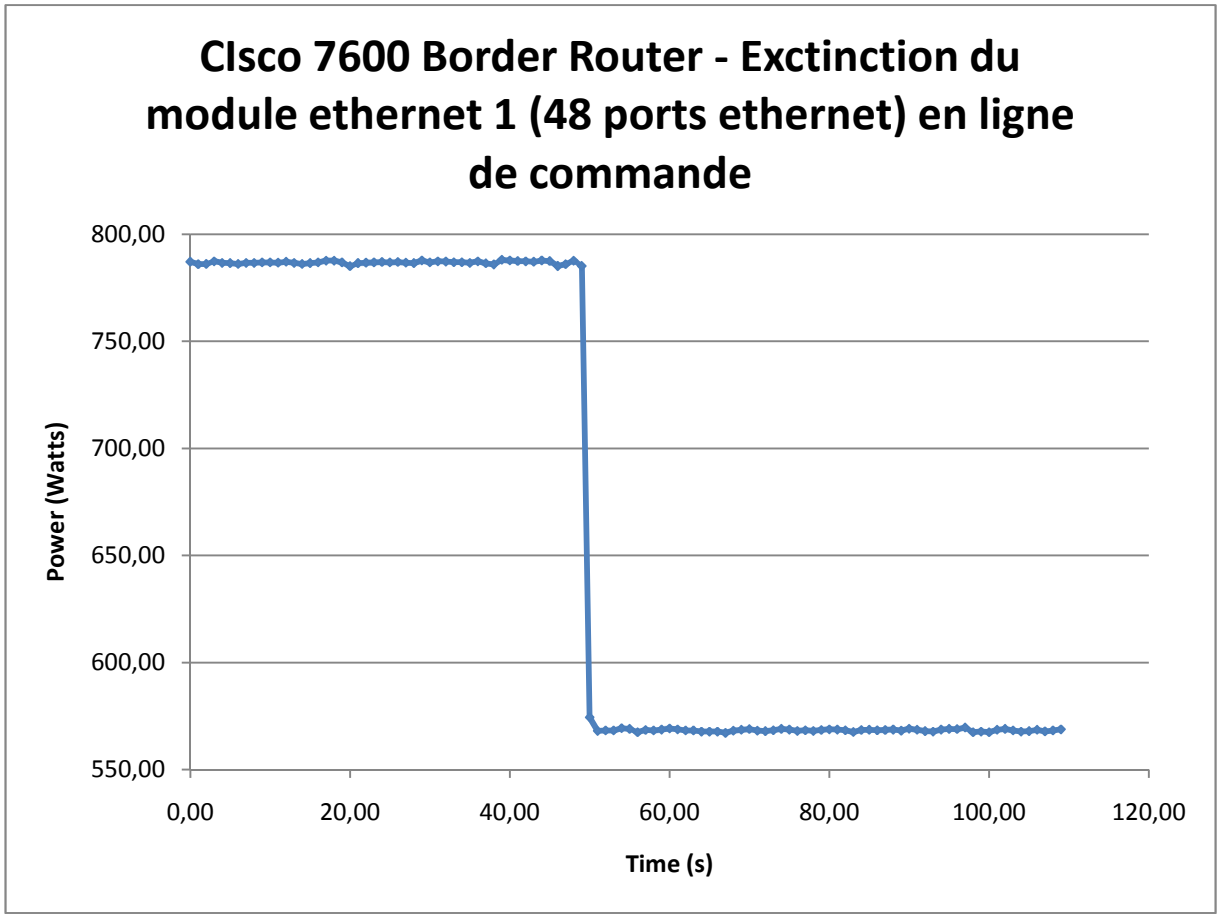
```
hw-module module <numero> boot
hw-module module <numero> reset
hw-module module <numero> simulate
```

Dans ce cas, il faut utiliser la commande no power. Pour rallumer, on utilise la commande :

```
R2(config)#power enable module <numero module>
```



Pour ce test, on arrête logiciellement avec la commande no power le module 4*10Gb/s de BB1 ayant les 4 ports allumés. C'est la même expérience que précédemment mais cette fois-ci on ne peut pas le rallumer car sur les routeurs de bordure on perd l'accès en coupant le module 10Gbps. Le module consomme 549,33 – 357,74 = 191,59 Watts en moyenne.



Pour ce test, on arrête logiciellement avec les mêmes commandes que précédemment (no power) le module 48 ports ethernet de R2 ayant tous les ports allumés. Cela fait gagner 218,36 watts. Par contre, on ne peut plus le rallumer car sur les routeurs de bordure on perd l'accès en coupant le module ethernet. Il faudrait passer par le module d'administration dédié.

Commandes utilisées :

```
R2>enable
Password:
R2#configure terminal
R2(config)#no power enable module <numero module>
```

Voici un résumé des mesures de consommation électrique faites sur Grid'Mip :

Module 10Gbps routeur de coeur : 191,59 watts

Module 10Gbps routeur de bordure : 195,57 watts

Ports 10Gbps : pas d'influence

Module ethernet 1Gbps routeur de bordure avec tous les ports allumés : 218,36 Watts

Ports 1Gbps : 0,94Watts chacun

Résumé des calculs :

Calcul du module ethernet 1Gbps routeur de bordure avec tous les ports éteints (mesure impossible car perte de contrôle du routeur) : 218,36 – 48*0,94 = 173,24 Watts

Détails des commandes show module:

```
BB1>show module 1
Mod Ports Card Type                                Model              Serial No.
--- ----- ---------------------------------------- ------------------ -----------
  1    4  CEF720 4 port 10-Gigabit Ethernet      WS-X6704-10GE      SAL1104F5JC

Mod MAC addresses                          Hw    Fw           Sw           Status
--- ---------------------------------- ------ ------------ ------------ -------
  1  001a.a10e.812c to 001a.a10e.812f   2.5   12.2(14r)S5  12.2(33)SRB5 Ok

Mod  Sub-Module                  Model              Serial       Hw     Status
---- --------------------------- ------------------ ------------ ------- -------
  1  Centralized Forwarding Card WS-F6700-CFC       SAL1117MKVC  3.1    Ok

Mod  Online Diag Status
---- -------------------
  1  Pass
BB1>


R2>show module 1
Mod Ports Card Type                                Model              Serial No.
--- ----- ---------------------------------------- ------------------ -----------
  1   48  CEF720 48 port 10/100/1000mb Ethernet  WS-X6748-GE-TX     SAL1117MRZ3

Mod MAC addresses                          Hw    Fw           Sw           Status
--- ---------------------------------- ------ ------------ ------------ -------
  1  001a.a10e.dde4 to 001a.a10e.de13   2.5   12.2(14r)S5  12.2(33)SRB5 Ok

Mod  Sub-Module                  Model              Serial       Hw     Status
---- --------------------------- ------------------ ------------ ------- -------
  1  Centralized Forwarding Card WS-F6700-CFC       SAL1117MPAL  3.1    Ok

Mod  Online Diag Status
---- -------------------
  1  Pass
R2>
```

Bibliographie

[1] Paul Horn. Autonomic computing : Ibm's perspective on the state of information technology. Technical report, IBM white papers, 2001.

[2] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, et al. Recovery-oriented computing (ROC) : motivation, definition, techniques, and case studies. Technical report, Computer Science Division, University of California at Berkeley, 2002.

[3] M. Wooldridge and N. R Jennings. Intelligent agents : Theory and practice. *The knowledge engineering review*, 10(02) :115–152, 2009.

[4] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, page 41–50, 2003.

[5] D. F Bantz, C. Bisdikian, D. Challener, J. P Karidis, S. Mastrianni, A. Mohindra, D. G Shea, and M. Vanover. Autonomic personal computing. *IBM Systems Journal*, 42(1) :165–176, 2003.

[6] T.W. Martin and K.C. Chang. A distributed data fusion approach for mobile ad hoc networks. In *8th international conference on information fusion*, Philadelphia, PA, USA, 2005.

[7] David S. Wile and Alexander Egyed. An externalized infrastructure for Self-Healing systems. In *Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, 2004.

[8] S. J Russell and P. Norvig. *Artificial intelligence : a modern approach*. Prentice hall, 2009.

[9] P. Pazandak and D. Wells. Distributed runtime software instrumentation. *First international workshop on unanticipated software evolution*, 2002.

[10] R. Brightwell, D. Doerfler, and K. D Underwood. A comparison of 4x infiniband and quadrics elan-4 technologies. In *IEEE International Conference on Cluster Computing*, page 193–204, 2004.

[11] W. Gropp, E. Lusk, and A. Skjellum. Using MPI : portable parallel programming with the message passing interface. *Massachusetts Institute of Technology Press*, 1999.

[12] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, et al. Grid'5000 : a large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, page 99–106, Seattle, Washington, USA, 2005.

[13] J. P LE GUIGNER. RENATER : réseau national de la technologie, de l'enseignement et de la recherche. *Bulletin des bibliothèques de France*, 39(1) :39–44, 1994.

[14] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Cluster Computing and the Grid, 2005. CCGrid 2005.*, 2005.

[15] MESCAL Project (CNRS, INPG, INRIA, UJF, LIG). Outil de déploiement kadeploy, développé par l'IMAG de grenoble. http ://kadeploy.imag.fr/, 2009.

[16] Brian Hayes. Cloud computing. *Communications of the ACM Journal*, 51(7) :9–11, 2008.

[17] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms : Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6) :599–616, 2009.

[18] I. F Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks : a survey. *Computer networks*, 38(4) :393–422, 2002.

[19] R. B Smith. SPOTWorld and the sun SPOT. In *Proceedings of the 6th international conference on Information processing in sensor networks*, page 566, 2007.

[20] G. Kaiser, G. Valetto, P. Gross, G. Kc, and J. Parekh. An approach to autonomizing legacy systems. COLUMBIA UNIV NEW YORK, 2005.

[21] E. Caron and F. Desprez. DIET : a scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3) :335, 2006.

[22] P. B Johns. Use of the transmission-line modelling (tlm) method to solve non-linear lumped networks. *Radio and Electronic Engineer*, 50(1/2), 1980.

[23] E. Perret, H. Aubert, and H. Legay. Scale-changing technique for the electromagnetic modeling of MEMS-controlled planar phase shifters. *Microwave Theory and Techniques, IEEE Transactions on*, 54(9) :3594–3601, 2006.

[24] A. G Ganek and T. A Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1) :5–18, 2003.

[25] S. MacKay. The state of the art in concurrent, distributed configuration management. *Software Configuration Management*, pages 180–193, 1995.

[26] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker. Towards distributed configuration. *KI 2001 : Advances in Artificial Intelligence*, page 198–212, 2001.

[27] Alan Lippman. Video coding for multiple target audiences. In *Visual Communications and Image Processing '99*, San Jose, CA, USA, 1999.

[28] M. Coutinho, R. Neches, A. Bugacov, K. T Yao, V. Kumar, I. Y Ko, R. Eleish, and S. Abhinkar. GeoWorlds : a geographically based information system for situation understanding and management. *TeleGeo*, 99 :6–7, 2001.

[29] G. M Lohman and S. S Lightstone. SMART : making DB2 (More) autonomic. In *Proceedings of the 28th international conference on Very Large Data Bases*, page 879, 2002.

[30] R. Gupta, S. Talwar, and D. P Agrawal. Jini home networking : a step toward pervasive computing. *Computer*, 35(8) :34–40, 2002.

[31] S. Agrawal, N. Bruno, S. Chaudhuri, and V. Narasayya. AutoAdmin : Self-Tuning database systems technology. *IEEE Data Engineering Bulletin*, 29(3) :7–15, 2006.

[32] L. Gürgen, J. Nyström-Persson, A. Cherbal, C. Labbé, C. Roncancio, and S. Honiden. Plug&manage heterogeneous sensing devices. In *Proceedings of the Sixth International Workshop on Data Management for Sensor Networks*, page 3, 2009.

[33] R. S Hall and H. Cervantes. An OSGi implementation and experience report. In *IEEE Consumer Communications and Networking Conference*, page 5, 2004.

[34] J. A. McCann and J. S. Crane. Kendra : Internet distribution & delivery system. *Proceedings of SCS Euromedia*, page 134–140, 1998.

[35] Chamara Gunaratne, Ken Christensen, and Bruce Nordman. Managing energy consumption costs in desktop PCs and LAN switches with proxying, split TCP connections, and scaling of link speed. *International Journal of Network Management*, 15(5) :297–310, 2005.

[36] C. Panarello, A. Lombardo, G. Schembra, L. Chiaraviglio, and M. Mellia. Energy saving and network performance : a trade-off approach. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, page 41–50, 2010.

[37] L. Chiaraviglio and I. Matta. GreenCoop : cooperative green routing with energy-efficient servers. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, page 191–194, 2010.

[38] J. Menon, D. A Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM storage Tank-A heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2) :250–267, 2003.

[39] C. Gui and P. Mohapatra. SHORT : self-healing and optimizing routing techniques for mobile ad hoc networks. In *4th ACM international symposium on Mobile ad hoc networking & computing*, page 290. ACM, 2003.

[40] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano-SLA based management of a computing utility. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, volume 5, Seattle , WA , USA, 2001.

[41] A. Montresor. Anthill : a framework for the design and analysis of peer-to-peer systems. In *4th European Research Seminar on Advances in Distributed Systems*, 2001.

[42] A. Montresor, H. Meling, and O. Babaoglu. Messor : Load-balancing through a swarm of autonomous agents. *Agents and Peer-to-Peer Computing*, page 125–137, 2003.

[43] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, et al. Oceanstore : An architecture for global-scale persistent storage. *ACM SIGARCH Computer Architecture News*, 28(5) :190–201, 2000.

[44] M. Pipattanasomporn, H. Feroze, and S. Rahman. Multi-agent systems in a distributed smart grid : Design and implementation. In *Proc. Proc. IEEE PES 2009 Power Systems Conference and Exposition (PSCE'09)*, 2009.

[45] S. W Cheng, D. Garlan, B. Schmerl, J. Sousa, B. Spitznagel, P. Steenkiste, and N. Hu. Software architecture-based adaptation for pervasive systems. *Trends in Network and Pervasive Computing—ARCS 2002*, page 217–233, 2002.

[46] P. Pazandak and D. Wells. ProbeMeister : distributed runtime software instrumentation. In *First international workshop on unanticipated software evolution*, Malaga, Spain, 2002.

[47] S. Agarwala, Y. Chen, D. Milojicic, and K. Schwan. QMON : QoS-and utility-aware monitoring in enterprise systems. In *IEEE International Conference on Autonomic Computing, 2006. ICAC'06*, page 124–133, 2006.

[48] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon : An information flow based approach to message brokering. *IBM TJ Watson Research Center Reports*, 1998.

[49] C. Poellabauer, K. Schwan, S. Agarwala, A. Gavrilovska, G. Eisenhauer, S. Pande, C. Pu, and M. Wolf. Service morphing : Integrated system-and application-level service adaptation in autonomic systems. *Active Middleware Services*, 2003.

[50] R. Van Renesse, K. P Birman, and W. Vogels. Astrolabe : A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems (TOCS)*, 21(2) :164–206, 2003.

[51] B. Jacob, R. Lanyon-Hogg, D.K. Nadgir, and A.F. Yassin. *A practical guide to the IBM autonomic computing toolkit*. Redbooks IBM, IBM, international technical support organization edition, 2004.

[52] J. P Bigus, D. A Schlosnagle, J. R Pilgrim, W. N. Mills, and Y. Diao. ABLE : a toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3) :350–371, 2002.

[53] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. AutoMate : enabling autonomic applications on the grid. *Cluster Computing*, 9(2) :161–174, 2006.

[54] M. L Massie, B. N Chun, and D. E Culler. The ganglia distributed monitoring system : design, implementation, and experience. *Parallel Computing*, 30(7) :817–840, 2004.

[55] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. Autonomia : an autonomic computing environment. In *Proc. of the 2003 IEEE International Performance, Computing, and Communication Conference*, page 61–68, 2003.

[56] Areski Flissi, J Dubus, Nicolas Dolet, and Philippe Merle. Deploying on the grid with DeployWare. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 177–184, Lyon, France, 2008.

[57] S. Sicard, F. Boyer, and N. De Palma. Using components for architecture-based management : the self-repair case. In *Proceedings of the 30th international conference on Software engineering*, page 101–110, 2008.

[58] M. Fowler and K. Scott. *UML distilled : a brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[59] M. Coppola, N. Tonellotto, M. Danelutto, C. Zoccolo, S. Lacour, C. Pérez, and T. Priol. Towards a common deployment model for grid systems. *Integrated Research in GRID Computing*, page 15–30, 2007.

[60] Boris Daix. *Abstraction des systèmes informatiques à haute performance pour l'automatisation du déploiement d'applications dynamiques.* PhD thesis, Université de Rennes 1, 2009.

[61] I. Foster. Globus toolkit version 4 : Software for service-oriented systems. *Journal of Computer Science and Technology*, 21(4) :513–520, 2006.

[62] L.. Broto, P. Stolf, J. P. Bahsoun, D. Hagimont, and N. de Palma. Spécification de politiques d'administration autonome avec tune. In *RenPar'18 / SympA'2008 / CFSE'6*, 2008.

[63] L. Broto, D. Hagimont, P. Stolf, N. de Palma, and S. Temate. Autonomic management policy specification in tune. In *ACM Symposium on Applied Computing*, pages 1658–1663, Fortaleza, Ceara, Brazil, 2008.

[64] Broto Laurent. *Support langage et système pour l'administration autonome.* PhD thesis, Université de Toulouse, Toulouse, France, September 2008.

[65] B. Dobing and J. Parsons. How UML is used. *Communications of the ACM*, 49(5) :113, 2006.

[66] B. Combemale, L. Broto, X. Crégut, D. Hagimont, and M. Dayde. Autonomic management policy specification : From UML to DSML. In *Conference on Parallel and Distributed Processing Techniques and Applications*, 2008.

[67] M. Dumas and A. ter Hofstede. UML activity diagrams as a workflow specification language. *UML 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, page 76–90, 2001.

[68] I. Poernomo. The meta-object facility typed. In *Proceedings of the 2006 ACM symposium on Applied computing*, page 1849, 2006.

[69] R. S Scowen. Extended BNF-a generic base standard. In *Software Engineering Standards Symposium*, volume 3, page 6–2, 1993.

[70] J. E Robbins and D. F Redmiles. Cognitive support, UML adherence, and XMI interchange in Argo/UML. *Information and Software Technology*, 42(2) :79–89, 2000.

[71] J. Reinders. *Intel threading building blocks : outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc., 2007.

[72] L. Dagum and R. Menon. OpenMP : an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1) :46–55, 2002.

[73] D. P Anderson. BOINC : a system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, page 4–10, 2004.

[74] D. P Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@ home : an experiment in public-resource computing. *Communications of the ACM*, 45(11) :56–61, 2002.

[75] A. D Birrell and B. J Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1) :39–59, 1984.

[76] J. M Geib and P. MERLE. CORBA : des concepts à la pratique. *Techniques de l'ingénieur. Informatique*, (H2758) :H2758, 2000.

[77] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web : an introduction to SOAP, WSDL, and UDDI. *IEEE Internet computing*, 6(2) :86–93, 2002.

[78] K. Scribner, K. Scribner, and M. C Stiver. *Understanding Soap : Simple Object Access Protocol*. Sams Indianapolis Editors, 2000.

[79] C. Ouyang, E. Verbeek, W. M.P Van Der Aalst, S. Breutel, M. Dumas, and A. H.M Ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3) :162–198, 2007.

[80] I. Fikouras and E. Freiter. Service discovery and orchestration for distributed service repositories. *Service-Oriented Computing-ICSOC 2003*, page 59–74, 2003.

[81] F. Khalil, B. Miegemolle, T. Monteil, H. Aubert, F. Coccetti, and R. Plana. Simulation of micro Electro-Mechanical systems (MEMS) on grid. In *VECPAR'08 - 8th international meeting high performance computing for computational science*, Toulouse, France, 2008.

[82] B. Claudel, G. Huard, and O. Richard. TakTuk, adaptive deployment of remote executions. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, page 91–100, 2009.

[83] J. Aagedal and E. Ecklund. Modelling QoS : towards a UML profile. *UML 2002—The Unified Modeling Language*, page 65–75, 2002.

[84] F. Chang and V. Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *High Performance Data Computing*, page 11, 2000.

[85] D. Wu, Y. T Hou, W. Zhu, Y. Q Zhang, and J. M Peha. Streaming video over the internet : Approaches and directions. *IEEE Transactions on circuits and systems for video technology*, 11(3) :282–300, 2001.

[86] J. Skene, D. D Lamanna, and W. Emmerich. Precise service level agreements. In *26th International Conference on Software Engineering (ICSE'04)*, Edinburgh, Scotland, United Kingdom, 2004.

[87] C. Aurrecoechea, A. T Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia systems*, 6(3) :138–151, 1998.

[88] D. Grossman et al. New terminology and clarifications for diffserv. *RFC 3260*, 2002.

[89] T. Davis, W. Tarreau, C. Gavrilov, C. N Tindel, J. Girouard, and J. Vosburgh. Linux ethernet bonding driver HOWTO. *Linux Channel Bonding project Web site http ://sourceforge. net/projects/bonding/*, 2007.

[90] R. W Callon. Use of OSI IS-IS for routing in TCP/IP and dual environments. *Network*, 1990.

[91] J. Moy. RFC2328 : OSPF version 2. *RFC Editor United States*, 1998.

[92] A. Zinin. *Cisco IP routing : packet forwarding and intra-domain routing protocols*. Addison-Wesley, 2002.

[93] C. Hedrick et al. Routing information protocol. Technical report, Citeseer, 1988.

[94] R. Albrightson, J. J. Garcia-Luna-Aceves, and J. Boyle. EIGRP-a fast routing protocol based on distance vectors. In *Proc. Networld/Interop*, volume 94, 1994.

[95] X. Lin and N. B Shroff. Utility maximization for communication networks with multipath routing. *IEEE Transactions on Automatic Control*, 51(5), 2006.

[96] JOpt, a simplified java wrapper for linear and mixed integer programming. http ://www.eecs.harvard.edu/econcs/jopt/, 2005.

[97] IBM - mathematical programs - IBM ILOG CPLEX optimizer - software. http ://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, 2009.

[98] LP_SOLVE : linear programming code. http ://www.cs.sunysb.edu/~algorith/implement/lpsolve/implement.shtml, 2008.

[99] GLPK - GNU project - free software foundation (FSF). http ://www.gnu.org/software/glpk/, 2008.

[100] R. Jain. Quality of experience. *IEEE Multimedia*, 11(1) :96–95, 2004.

[101] D. M Chess, C. C Palmer, and S. R White. Security in an autonomic computing environment. *IBM Systems Journal*, 42(1) :107–118, 2010.

[102] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4) :287–317, 1983.

[103] M. Little. Transactions and web services. *Communications of the ACM*, 46(10) :49–54, 2003.

[104] S. Lacour, C. Pérez, and T. Priol. A network topology description model for grid application deployment. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, page 61–68, 2004.

[105] L. Valcarenghi, L. Foschini, F. Paolucci, P. Castoldi, and F. Cugini. Topology discovery services for monitoring the global grid. *Communications Magazine, IEEE*, 44(3) :110–117, 2006.

[106] S. Frolund and J. Koistinen. Quality-of-service specification in distributed object systems. *Distributed Systems Engineering*, 5 :179, 1998.

[107] M. A Touré. *Administration d'applications réparties à grande échelle*. PhD thesis, Université de Toulouse, June 2010.

[108] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM Journal*, 53(4), 2010.

[109] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in java. *Software : Practice and Experience*, 36(11-12) :1257–1284, 2006.

[110] C. Taton, S. Bouchenak, N. D. Palma, D. Hagimont, S. Krakowiak, and J. Arnaud. Administration autonome de services internet : Expérience avec l'auto-optimisation. In *5ème Conférence Française sur les Systèmes d'Exploitation (CFSE 2006), Le Canet en Roussillon, France*, 2006.