Coordination Modulaire de Gestionnaires Autonomes par Contrôle Discret

Soguy Mak-Karé Gueye

▶ **To cite this version:**

Soguy Mak-Karé Gueye. Coordination Modulaire de Gestionnaires Autonomes par Contrôle Discret. Calcul parallèle, distribué et partagé [cs.DC]. Universite de Grenoble, 2014. Français. NNT : . tel-01091302

HAL Id: tel-01091302
https://inria.hal.science/tel-01091302

Submitted on 5 Dec 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Soguy Mak-Karé GUEYE

Thèse dirigée par **Éric RUTTEN**
et codirigée par **Noël DE PALMA**

préparée au sein **Laboratoire d'Informatique de Grenoble (LIG) et INRIA Grenoble Rhône-Alpes**
et de **L'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Coordination Modulaire de Gestionnaires Autonomes par Contrôle Discret

Thèse soutenue publiquement le **03/12/2014**,
devant le jury composé de :

Mr. Lionel SEINTURIER
Professeur, Université Lille 1 - UFR IEEA - LIFL & FIL, Rapporteur
Mme. Françoise BAUDE
Professeur, Université de Nice Sophia-Antipolis, INRIA-UNS-CNRS, Rapporteur
Mr. Hervé MARCHAND
Chargé de Recherche, INRIA Rennes-Bretagne Atlantique, Examinateur
Mr. Jean-marc FAURE
Professeur, SupMéca, (LURPA), Examinateur
Mr. Gwenaël DELAVAL
Maître de conférence, Université Joseph Fourier - LIG, Examinateur
Mr. Daniel HAGIMONT
Professeur, INPT/ENSEEIHT, Examinateur
Mr. Éric RUTTEN
Chargé de Recherche, INRIA Grenoble Rhône-Alpes, Directeur de thèse
Mr. Noël DE PALMA
Professeur, Université Joseph Fourier - LIG, Co-Directeur de thèse

Abstract

Computing systems have become more and more distributed and heterogeneous, making their manual administration difficult and error-prone. The Autonomic Computing approach has been proposed to overcome this issue, by automating the administration of computing systems with the help of control loops called autonomic managers. Many research works have investigated the automation of the administration functions of computing systems and today many autonomic managers are available. However the existing autonomic managers are mostly specialized in the management of few administration concerns. This makes necessary the coexistence of multiple autonomic managers for achieving a global system management. The coexistence of several managers make possible to address multiple concerns, yet requires coordination mechanisms to avoid incoherent management decisions. We investigate the use of control techniques for the design of coordination controllers, for which we exercise synchronous programming that provide formal semantics, and discrete controller synthesis to automate the construction of the controller. We follow a component-based approach, and explore modular discrete control allowing to break down the combinatorial complexity inherent to the state-space exploration technique. This improves scalability of the approach and allows constructing a hierarchical control. It also allows re-using complex managers in different contexts without modifying their control specifications. We build a component-based coordination of managers, with introspection, adaptivity and reconfiguration. This thesis details our methodology and presents case-studies. We evaluate and demonstrate the benefits of our approach by coordinating autonomic managers which address the management of availability, and the management of performance and resources optimization.

Keywords. Computer systems, autonomic computing, component-based model, control loops, software reuse, discrete event systems, discrete controller synthesis, synchronous programming

Résumé

Les systèmes informatiques sont devenus de plus en plus distribués et hétérogènes, ce qui rend leur administration manuelle difficile et source d'erreurs. L'administration autonome a été proposée comme solution à ce problème. Elle consiste à automatiser l'administration des systèmes informatiques à l'aide de boucles de contrôle appelées gestionnaires autonomes. De nombreux travaux de recherche se sont intéressés à l'automatisation des fonctions d'administration de systèmes informatiques et aujourd'hui, beaucoup de gestionnaires autonomes sont disponibles. Toutefois, les gestionnaires autonomes existants sont, la plupart, spécialisés dans la gestion de quelques aspects d'administration. Cela rend nécessaire la coexistence de plusieurs gestionnaires autonomes pour atteindre une gestion globale des systèmes. La coexistence de plusieurs gestionnaires permet la gestion de plusieurs aspects, mais nécessite des mécanismes de coordination afin d'éviter des décisions incohérentes. Nous étudions l'utilisation de techniques de contrôle pour la conception de contrôleurs de coordination, nous utilisons la programmation synchrone qui fournit des méthodes formelles, et la synthèse de contrôleur discret pour automatiser la construction de contrôleur. Nous suivons une approche à base de composants, et utilisons le contrôle discret modulaire qui permet de décomposer la complexité combinatoire inhérente à la technique d'exploration d'espace d'états. Cela améliore le passage à l'échelle de notre approche et permet la construction d'un contrôle hiérarchique. Notre approche permet la réutilisation de gestionnaires complexes dans des contextes différents, sans modifier leurs spécifications de contrôle. Nous construisons une coordination de gestionnaires basée sur le modèle à composants offrant introspection, adaptabilité et reconfiguration. Cette thèse présente notre méthodologie et des études de cas. Nous évaluons et démontrons les avantages de notre approche par la coordination de gestionnaires autonomes dédiés à la gestion de la disponibilité, et à la gestion de la performance et l'optimisation de ressources.

Mots-clés. Systèmes informatiques, administration autonome, modèle à base de composants, boucles de contrôle, réutilisation de logiciels, systèmes à événements discrets, synthèse de contrôleur discret, programmation synchrone

Remerciements

Je tiens à adresser mes plus sincères remerciements et toute ma reconnaissance à mes directeurs de thèse, Mr Éric RUTTEN et Mr Noël DE PALMA, pour m'avoir offert la possibilité de faire cette thèse, pour leur soutien constant, leur présence, leur patience et leur confiance.

Je remercie Mr Gwenaël DELAVAL pour sa présence et son aide tout au long de la thèse. Je tiens à remercier Mr Ahmed EL RHEDDANE, Mr Ibrahim SAFIEDDINE, Mr Alain TCHANA, pour leur présence et leur soutien durant toutes ces années. Je tiens également à remercier Mlle Édith GRAC et Mr Frédérico ALVARES pour avoir pris le temps de lire le manuscrit et de suggérer des corrections qui ont sans aucun doute contribué à améliorer la qualité du manuscrit. Je remercie mes collègues et toutes les personnes avec qui j'ai collaboré pendant toutes ces années.

Je remercie également les membres du jury, qui ont accepté de juger mon travail : Mr Lionel SEINTURIER et Mme Françoise BAUDE pour avoir également accepté de rapporter la thèse, Mr Hervé MARCHAND, Mr Jean-marc FAURE, Mr Gwenaël DELAVAL, Mr Daniel HAGIMONT, pour avoir accepté d'examiner en profondeur le travail réalisé, ainsi que pour tous leurs commentaires constructifs.

Je remercie mes amis, principalement Mlle Ndeye Fatou NDIAYE, qui a beaucoup contribué à l'organisation de la soutenance. Enfin, je remercie toutes les personnes qui de près ou de loin m'ont soutenu, encouragé ou assisté durant toutes ces années.

Mes dernières pensées sont réservées à ma très chère famille...

Table des matières

Résumé **i**

Remerciements **iii**

Table des matières **v**

1 Introduction **1**
1.1 Systèmes autonomes 1
1.2 Coordination de gestionnaires autonomes 2
1.3 Approche et contribution 3

2 État de l'art **7**
2.1 Administration autonome 8
2.1.1 Complexité des systèmes informatiques 9
2.1.1.1 Architecture multi-tiers 9
2.1.1.2 Centre de données: ressources à large échelle . 10
2.1.1.3 Limites de l'administration manuelle 10
2.1.2 Gestionnaire autonome 11
2.1.2.1 Définition de gestionnaire autonome 11
2.1.2.2 Implémentation 13
2.1.3 Coordination de gestionnaires autonomes 16
2.1.3.1 Besoin de coordination 16
2.1.3.2 Approches de coordination proposées 17
2.1.4 Synthèse 19
2.2 Modèles réactifs 20
2.2.1 Les langages synchrones 21

2.2.1.1 Heptagon/BZR 22
2.2.1.2 Implémentation des programmes synchrones . 25
2.2.2 Synthèse de contrôleur discret (SCD) 26
2.2.2.1 Synthèse de contrôleur avec Heptagon/BZR . . 27
2.2.2.2 Synthèse modulaire avec Heptagon/BZR . . . 29
2.2.3 Synthèse 31
2.3 Conclusion 32

3 Méthodologie de coordination de gestionnaires autonomes 35
3.1 Spécification de la coordination 36
3.1.1 Modélisation d'un gestionnaire autonome 36
3.1.1.1 Comportement 37
3.1.1.2 Contrôlabilité 38
3.1.2 Modélisation de la coordination 39
3.1.2.1 Modélisation de la coexistence 39
3.1.2.2 Spécification d'une stratégie de coordination . 40
3.1.3 Modélisation modulaire de la coordination 41
3.1.3.1 Contrôle décentralisé 41
3.1.3.2 Spécification modulaire et hiérarchique 42
3.2 Mise en oeuvre de la coordination 44
3.2.1 Le modèle à composants Fractal 44
3.2.1.1 Composant Fractal 44
3.2.1.2 Introspection et reconfiguration 46
3.2.1.3 Fractal ADL 46
3.2.2 Composant de gestionnaire autonome 47
3.2.3 Coordination à base de composants 48
3.2.3.1 Coordination de gestionnaires 48
3.2.3.2 Coordination hiérarchique 50
3.3 Comparaison 52
3.4 Conclusion 53

4 Gestion de la performance et de l'optimisation de ressources d'un système dupliqué 55
4.1 Gestionnaires autonomes non coordonnés 56
4.1.1 Gestionnaire d'auto-dimensionnement: Self-sizing 56

4.1.2 Gestionnaire d'auto-régulation de fréquence CPU: Dvfs 58
4.2 Problèmes d'optimisation de ressources 59
4.3 Conception du contrôleur de coordination 60
4.3.1 Modélisation du contrôle des gestionnaires 60
4.3.1.1 Modélisation du contrôle de self-sizing 61
4.3.1.2 Modélisation de l'état global des Dvfs 63
4.3.2 Spécification de la coordination 64
4.3.2.1 Stratégie de coordination 64
4.3.2.2 Spécification du contrat 64
4.3.2.3 Programme final 64
4.4 Expérimentations 65
4.4.1 Configuration 66
4.4.2 Calibrage des seuils des gestionnaires 66
4.4.2.1 Seuil maximal pour self-sizing et Dvfs 66
4.4.2.2 Seuil minimal pour self-Sizing et Dvfs 67
4.4.3 Évaluation 70
4.4.3.1 Comportement non coordonné 70
4.4.3.2 Comportement coordonné 72
4.5 Conclusion 75

5 Gestion du dimensionnement dynamique et de la réparation d'un système multi-tiers 77
5.1 Gestionnaires autonomes non coordonnés 79
5.1.1 Gestionnaire d'auto-dimensionnement: Self-sizing 79
5.1.2 Gestionnaire d'auto-réparation: Self-repair 80
5.2 Problèmes d'administration d'un système multi-tiers 81
5.3 Conception du contrôleur de coordination 84
5.3.1 Modélisation du contrôle des gestionnaires 84
5.3.1.1 Modélisation du contrôle de self-sizing 84
5.3.1.2 Modélisation du contrôle de self-repair 84
5.3.2 Spécification de la coordination 86
5.3.2.1 Stratégie de coordination 86
5.3.2.2 Spécification du contrat 87
5.3.2.3 Programme final 89
5.4 Expérimentations 90

5.4.1 Configuration 91
5.4.2 Évaluation 91
5.4.2.1 Comportement non coordonné 92
5.4.2.2 Comportement coordonné 95
5.5 Conclusion 98

6 Coordination modulaire pour la gestion d'applications multi-tiers et consolidation 99
6.1 Gestion des ressources d'un centre de données 100
6.1.1 Utilisation des ressources 100
6.1.2 Gestionnaire de consolidation de serveurs 101
6.2 Problèmes 101
6.3 Conception de la coordination modulaire 102
6.3.1 Modélisation des gestionnaires 103
6.3.1.1 Modélisation du gestionnaire self-sizing 103
6.3.1.2 Modélisation du gestionnaire self-repair 104
6.3.1.3 Modélisation du gestionnaire de consolidation 104
6.3.2 Spécification de la coordination 105
6.3.2.1 Stratégie de coordination 105
6.3.2.2 Spécification du contrat 106
6.3.2.3 Synthèse monolithique 106
6.3.2.4 Synthèse modulaire 107
6.3.2.5 Comparaison 111
6.4 Expérimentations 112
6.4.1 Configuration 112
6.4.2 Évaluation 113
6.5 Conclusion 115

7 Exécution distribuée des contrôleurs modulaires 117
7.1 Exécution distribuée de contrôleurs 119
7.1.1 Exécution distribuée synchronisée 119
7.1.1.1 Principe 120
7.1.1.2 Implémentation 120
7.1.2 Exécution distribuée désynchronisée 122
7.1.2.1 Principe 122

7.1.2.2 Implémentation 123
7.2 Exemple: Gestion d'une application multi-tiers 124
7.2.1 Exécution distribuée totalement synchronisée 124
7.2.1.1 Modélisation 124
7.2.1.2 Décomposition 125
7.2.2 Exécution distribuée partiellement synchronisée 126
7.2.2.1 Modélisation 126
7.2.2.2 Décomposition 129
7.2.3 Exécution distribuée désynchronisée 130
7.2.3.1 Modélisation 130
7.2.3.2 Décomposition 131
7.2.4 Comparaison 132
7.3 Expérimentation 133
7.3.1 Configuration 133
7.3.2 Évaluation 133
7.3.2.1 Durée de reconfiguration 133
7.3.2.2 Atteinte des objectifs de contrôle 134
7.4 Conclusion 140

8 Conclusion **143**

Bibliographie **147**

Liste des figures **155**

Liste des tables **159**

1

Introduction

Contents

1.1 Systèmes autonomes 1
1.2 Coordination de gestionnaires autonomes 2
1.3 Approche et contribution 3

1.1 Systèmes autonomes

Aujourd'hui les systèmes informatiques sont présents dans de nombreux secteurs d'activité pour réaliser des traitements complexes, e.g., le commerce en ligne ou les opérations bancaires. Ces systèmes, ayant fait leurs preuves, sont devenus de plus en plus utilisés et de plus en plus complexes, avec de multiples ressources logicielles hétérogènes inter-connectées entre elles. De plus l'environnement d'exécution de ces systèmes a évolué pour répondre à la demande accrue en puissance de calcul. Des équipements de plus en plus sophistiqués sont utilisés. Ces évolutions posent de nouveaux défis, notamment l'administration des systèmes qui devient une tâche de plus en plus complexe.

La complexité inhérente à la taille des systèmes et à leur degré d'hétérogénéité rend difficile leur administration. Cette dernière ne peut plus être assurée efficacement de manière manuelle. En effet, pour gérer les systèmes informatiques, l'intervention humaine implique souvent des coûts élevés, des erreurs

fréquentes, et surtout des temps de réaction lents. Or les enjeux économiques liés au bon fonctionnement des systèmes informatiques ne tolèrent ni des pannes ni de longues périodes d'indisponibilité des systèmes. Il faut donc trouver d'autres moyens pour assurer leur administration.

L'administration autonome [1] a été proposée comme alternative pour faciliter la gestion des systèmes informatiques. Cette approche consiste à concevoir des systèmes autonomes capables de se gérer eux-mêmes. A l'exécution, ces systèmes doivent pouvoir réagir et s'adapter aux changements survenus dans leur environnement d'exécution sans intervention humaine ou peu. Pour ce faire, les systèmes autonomes sont munis d'éléments logiciels dédiés à leur administration. Ce sont eux qui leur permettent de s'auto-administrer et de minimiser l'intervention humaine. Ces éléments, appelés gestionnaires autonomes, implémentent les décisions d'administration. Ils reçoivent, via des capteurs, des données sur l'état courant du système administré et l'occurrence d'événements qui affectent le système. Ils analysent ensuite ces données pour détecter tout écart par rapport au fonctionnement souhaité du système. Lorsqu'écart il y a, ils planifient et exécutent des opérations d'administration pour reconfigurer le système dans un état cohérent et stable.

1.2 Coordination de gestionnaires autonomes

Pour qu'un système soit entièrement autonome, il faut que toutes les fonctions d'administration soient automatisées. Parmi tous les gestionnaires autonomes actuellement disponibles, aucun n'est capable d'implémenter une administration globale. La conception d'un gestionnaire autonome qui implémente toutes les fonctions d'administration peut être complexe. Toutefois, de nombreux gestionnaires autonomes dédiés à différentes fonctions d'administration sont disponibles et réutilisables. De ce fait, ils peuvent être utilisés en parallèle pour l'administration d'un système.

Il peut être avantageux d'utiliser plusieurs gestionnaires pour faciliter une gestion globale d'un système de manière autonome. Cependant une gestion globale et cohérente nécessite la coordination des gestionnaires autonomes,

1. La plupart du temps le terme «**autonomique**» est utilisé. Toutefois, étant donné qu'il n'y a pas de consensus pour l'utilisation de ce terme, dans ce document nous utilisons le terme «**autonome**».

qui sont généralement conçus indépendamment. Individuellement chaque gestionnaire a un comportement cohérent, mais leur coexistence peut amener des incohérences. Chaque gestionnaire assure le respect de ses objectifs d'administration en se basant sur la connaissance qu'il a du système administré, et en appliquant des actions d'administration. Ces actions affectent l'état du système administré, et donc peuvent conduire à une violation des objectifs des autres gestionnaires. Ces derniers peuvent réagir à ces changements en appliquant des actions de correction qui n'aboutissent pas forcément à une stabilité du système administré. De plus, un événement peut altérer l'état du système administré conduisant ainsi à des réactions simultanées de plusieurs gestionnaires.

Une gestion autonome complète et cohérente requiert donc l'utilisation de plusieurs gestionnaires autonomes coordonnés. La coordination de leurs activités permet d'éviter des décisions conflictuelles ainsi que des actions inutiles et peut-être redondantes.

1.3 Approche et contribution

La coordination de gestionnaires autonomes nécessite une synchronisation, au moins partielle, de leurs activités pour pouvoir autoriser ou inhiber certaines actions en fonction des circonstances. Il s'agit de restreindre le fonctionnement global des gestionnaires dicté par des objectifs globaux. Des approches issues de la théorie du contrôle, comme la synthèse de contrôleur discret (SCD), permettent d'aborder ce type de problème. La synthèse de contrôleur discret est une technique qui permet de construire automatiquement une fonction logique, un contrôleur qui permet de restreindre le fonctionnement d'un système pour respecter les propriétés désirées. Le contrôleur généré restreint le moins possible le fonctionnement du système à contrôler. La synthèse de contrôleur discret repose sur une déclaration des propriétés désirées et sur une modélisation du système à contrôler. La programmation synchrone fournit des langages de haut niveau facilitant la modélisation formelle de systèmes à base d'automates. Elle permet la représentation de systèmes complexes par la composition parallèle et hiérarchique d'automates.

Pour coordonner des gestionnaires autonomes, nous proposons une approche basée sur la synthèse de contrôleur discret et la programmation syn-

chrone. En effet l'implémentation manuelle d'une politique de coordination peut être complexe, coûteuse, et implique des séries de tests et de corrections successives. Avec notre approche, le contrôleur de coordination est construit automatiquement sur la base des objectifs de coordination et d'un modèle du système. Nous adoptons la programmation synchrone pour la modélisation du système à contrôler. Ce dernier correspond à l'ensemble des gestionnaires autonomes à coordonner. La coordination requiert que le comportement des gestionnaires, à l'exécution, puisse être observable et contrôlable. Nous construisons un modèle de chaque gestionnaire dans lequel ces aspects sont décrits. La composition des modèles des gestionnaires représente leur coexistence et décrit l'ensemble des comportements qui peuvent être observés durant leur exécution parallèle. Elle présente les comportements cohérents, et ceux considérés incohérents qui rendent inconsistant l'état du système administré par les gestionnaires.

La mise en oeuvre de la coordination nécessite une implémentation réelle des fonctions permettant d'observer et de contrôler les gestionnaires. Pour cela nous adoptons le modèle à composants qui facilite l'assemblage dynamique d'éléments logiciels sur lesquels des fonctions d'introspection et de reconfiguration peuvent être ajoutées. Nous identifions les contraintes de conception des gestionnaires – comportement observable, et contrôlabilité – pour construire une structure à base de composants où ils sont explicites, et n'impliquant pas la modification de l'implémentation des gestionnaires autonomes pour lesquels ces fonctions de contrôle ne sont pas disponibles.

Contributions

Nos contributions sont les suivantes :

1. Coordination de gestionnaires autonomes basée sur le contrôle discret
   (a) Spécification du contrôle de gestionnaire avec la programmation synchrone
   (b) Respect de la coordination par synthèse de contrôleur discret
   (c) Passage à l'échelle par synthèse modulaire hiérarchique
2. Mise en oeuvre de la coordination basée sur le modèle à composants
   (a) Construction de composants de gestionnaires

i. Implémentation du contrôle de gestionnaire

ii. Réutilisation de gestionnaires existants

(b) Construction de composants composites de coordination

i. Assemblage de composants de gestionnaires

ii. Assemblage de composants composites coordonnés

3. La validation de notre approche à travers des cas d'étude présentés dans les chapitres suivants.

Les contributions de cette thèse s'inscrivent dans le cadre du projet *Ctrl-Green* [2]. Ce projet vise à étudier les moyens matériels et logiciels pour l'optimisation de la consommation énergétique dans les centres de données. La gestion énergétique peut être implémentée à différents niveaux (matériel, système, intergiciel). De multiples boucles de contrôle – gestionnaires autonomes – peuvent donc être implémentées à chaque niveau et elles doivent prendre des décisions globalement cohérentes. De plus le *Green Computing* n'est pas le seul aspect qui doit être géré dans l'administration d'un centre de données. Des politiques pour le passage à l'échelle et/ou la disponibilité des applications hébergées sont également prises en compte. Il est donc nécessaire de gérer les compromis entre performance, disponibilité et énergie. Pour cela la coexistence de nombreux gestionnaires autonomes (avec des objectifs différents, implémentés dans des couches différentes) est nécessaire. Leur coordination est également nécessaire pour obtenir une gestion cohérente.

Organisation du document

Le reste du document est organisé de la manière suivante :

Le **chapitre 2** présente l'état de l'art sur l'administration autonome et les modèles réactifs. D'abord nous présentons l'administration autonome, la nécessité et les solutions proposées. Ensuite nous présentons des techniques issues du contrôle discret pour la conception de systèmes réactifs.

Le **chapitre 3** détaille notre approche de coordination de gestionnaires, basée sur l'utilisation de modèles réactifs et des techniques de contrôle discret pour

2. Ctrl-Green (ANR-11-INFR 012 11) est un projet de recherche financé par l'ANR (Agence Nationale de la Recherche) avec le soutien de MINALOGIC. `http://www.ctrlgreen.org/`

la spécification de la coordination ; et le modèle à composants pour la mise en oeuvre [21, 29]. Nous utilisons le synthèse modulaire pour faciliter le passage à l'échelle et construire un contrôle hiérarchique [23].

Le **chapitre 4** présente une application de notre approche pour la gestion cohérente de l'optimisation des ressources, dans la perspective de l'informatique verte [28]. Les gestionnaires considérés gèrent le dimensionnement dynamique du degré de réplication de serveurs (self-sizing) et l'ajustement dynamique de la fréquence du CPU (DVFS «Dynamic Frequency Voltage Scaling»). Le contrôleur de coordination contrôle les actions d'ajout de serveurs dupliqués en fonction de l'état des processeurs des machines déjà en cours d'utilisation. L'objectif est de n'autoriser un ajout de serveur que lorsque c'est nécessaire, lorsque les serveurs actifs sont réellement saturés.

Le **chapitre 5** présente une autre application de notre approche pour la gestion de la disponibilité et de la performance d'un système multi-tiers basé sur la réplication [30]. Le système exécute l'application Web de référence RUBiS. Les gestionnaires considérés sont self-sizing pour le dimensionnement dynamique du degré de réplication des tiers dupliqués, et self-repair pour la réparation de serveurs en panne. Le contrôleur de coordination conçu orchestre l'exécution des gestionnaires afin d'éviter un sur-dimensionnement.

Le **chapitre 6** présente une application de l'approche modulaire pour la gestion d'un centre de données [23]. Nous considérons que le centre de données héberge un ensemble d'applications de type multi-tiers JEE. Chacune des applications est gérée de manière autonome par deux instances de self-sizing et quatre instances de self-repair. Cet exemple démontre le passage à l'échelle de notre approche par la spécification modulaire du contrôle des gestionnaires des applications et du gestionnaire de consolidation.

Le **chapitre 7** décrit comment le code obtenu d'une spécification modulaire peut être exécuté de manière distribuée. Il détaille les différentes approches d'exécution distribuée et décrit les implémentations possibles pour la mise en oeuvre. Il présente également un exemple d'utilisation de ces différentes approches pour la gestion d'un système multi-tiers.

Le **chapitre 8** conclut la thèse. Il rappelle le contexte, l'approche de coordination et donne des perspectives que nous envisageons.

2

État de l'art

Contents

2.1 Administration autonome 8
2.1.1 Complexité des systèmes informatiques 9
2.1.1.1 Architecture multi-tiers 9
2.1.1.2 Centre de données : ressources à large échelle 10
2.1.1.3 Limites de l'administration manuelle 10
2.1.2 Gestionnaire autonome 11
2.1.2.1 Définition de gestionnaire autonome 11
2.1.2.2 Implémentation 13
2.1.3 Coordination de gestionnaires autonomes 16
2.1.3.1 Besoin de coordination 16
2.1.3.2 Approches de coordination proposées 17
2.1.4 Synthèse 19
2.2 Modèles réactifs 20
2.2.1 Les langages synchrones 21
2.2.1.1 Heptagon/BZR 22
2.2.1.2 Implémentation des programmes synchrones 25
2.2.2 Synthèse de contrôleur discret (SCD) 26
2.2.2.1 Synthèse de contrôleur avec Heptagon/BZR . 27
2.2.2.2 Synthèse modulaire avec Heptagon/BZR . . 29

2.2.3 Synthèse 31
2.3 Conclusion 32

Ce chapitre présente l'administration autonome. Cette approche repose sur l'automatisation des fonctions d'administration. Ce chapitre présente la nécessité de cette approche et les éléments logiciels, gestionnaires autonomes, qui permettent sa mise en oeuvre. Il présente également l'intérêt d'utiliser plusieurs gestionnaires autonomes et la nécessité de les coordonner, et quelques approches de coordination étudiées dans des travaux de recherche.

La seconde partie de ce chapitre présente des méthodes et des techniques, issues de la programmation synchrone et de la synthèse de contrôleur discret, sur lesquelles est basée notre méthodologie de coordination de gestionnaires. La programmation synchrone fournit des langages de programmation de haut niveau permettant la modélisation formelle du comportement d'un système ; et la synthèse de contrôleur discret.

2.1 Administration autonome

L'administration d'un système informatique consiste en un ensemble d'opérations en vue de le mettre en état de marche et de maintenir correct son fonctionnement tout au long de son exécution. La mise en marche du système implique la configuration et la résolution des dépendances de ses différents éléments logiciels déployés sur des machines dédiées. Assurer le bon fonctionnement du système implique une surveillance continue de l'environnement logiciel et matériel afin de détecter des problèmes et d'appliquer des opérations de reconfiguration.

Aujourd'hui les systèmes informatiques sont devenus de plus en plus complexes. Leur administration dépasse les capacités humaines à cause du nombre très important d'éléments logiciels et matériels impliqués. L'administration autonome [39], introduite en 2001 par IBM, a été proposée comme alternative face à cette complexité croissante. Cette approche consiste à automatiser les fonctions d'administration afin de minimiser l'intervention humaine et d'améliorer la réactivité quant à la détection de changements et l'application d'opérations

de correction. Les fonctions d'administration sont implémentées par des éléments logiciels appelés gestionnaires autonomes. A l'exécution, ces derniers vont appliquer les tâches d'administration sur le système administré.

2.1.1 Complexité des systèmes informatiques

L'informatique est un domaine essentiel dans beaucoup de secteurs d'activité. La plupart des activités, souvent complexes, dans ces secteurs est assurée au moyen de systèmes informatiques afin de faciliter leur gestion. Ces systèmes fournissent des services, e.g., le commerce en ligne, les opérations bancaires, ouverts à un nombre important d'utilisateurs. Ils évoluent souvent dans un environnement très dynamique. Les enjeux économiques liés à ces activités font que les systèmes informatiques associés doivent assurer un bon fonctionnement qui requiert une forte fiabilité. Une panne ou une indisponibilité peut causer des pertes financières considérables. Le besoin de stabilité a conduit à la conception de systèmes larges et complexes, avec de nombreux éléments logiciels répartis, déployés suivant une architecture distribuée. Par ailleurs, l'exécution de ces systèmes nécessite de grandes quantités de ressources.

2.1.1.1 Architecture multi-tiers

L'architecture multi-tiers est un exemple d'architecture distribuée de plus en plus utilisée. Elle permet l'interconnexion de serveurs qui rendent des services différents, e.g., la production de page web avec la gestion de base de données. Les serveurs sont groupés en tiers. Chaque tier offre un service à son prédécesseur, et requiert le service fourni par son successeur pour remplir sa part dans la chaîne de traitement des requêtes. Les différents tiers peuvent participer au traitement de chaque requête entrante durant l'exécution du système. Pour des raisons de performance et de disponibilité, chaque tier peut être dupliqué sur plusieurs machines distinctes. Un répartiteur de charge est alors utilisé en frontal à chaque tier basé sur la réplication pour distribuer les requêtes à traiter entre les serveurs dupliqués du tier.

2.1.1.2 Centre de données : ressources à large échelle

La mise en exploitation des systèmes informatiques nécessite aujourd'hui une importante puissance de calcul. Ce besoin a conduit les entreprises à s'orienter vers les centres de données. Un centre de données est constitué d'un ensemble d'équipements matériels sophistiqués et à grande échelle pour fournir une puissance de calcul et une capacité de stockage très élevées. La puissance de calcul et la capacité stockage sont fournies par une plate-forme formée par un nombre important de serveurs physiques puissants inter-connectés via un réseau à haut débit. Cette plate-forme est généralement virtualisée ce qui permet d'exécuter plusieurs applications distinctes simultanément de manière isolée. Un système de virtualisation, e.g., `VMWare`, `Xen`, est installé sur les serveurs physiques pour former la couche virtuelle. La ressource approvisionnée dans ce type de plate-forme est la machine virtuelle. Une machine virtuelle est un logiciel qui reproduit le même comportement qu'un ordinateur réel. Elle contient un système d'exploitation et ne se distingue pas d'un ordinateur réel, vu de l'intérieur. Elle peut héberger des logiciels applicatifs et exécuter la plupart des tâches qu'un ordinateur réel peut exécuter. Le système de virtualisation gère le quota de ressources physiques alloué à chaque machine virtuelle durant son exécution. Le quota de chaque machine virtuelle peut évoluer en fonction de ses besoins.

2.1.1.3 Limites de l'administration manuelle

La complexité croissante des systèmes informatiques a rendu les tâches d'administration difficiles à assurer manuellement. La capacité humaine pour la gestion d'un système informatique – déploiement et reconfiguration – est vite dépassée par la complexité de celui-ci. Le déploiement implique la copie, l'installation de chaque logiciel sur la machine dédiée, leur configuration et leur mise en marche. La reconfiguration implique une surveillance continue de l'ensemble des éléments logiciels et matériels pour la détection de changements dans l'environnement d'exécution – détection de défaillance, charge de travail – auxquels il faut réagir. Ces tâches sont rendues difficiles à effectuer manuellement par le nombre très important d'éléments impliqués.

2.1.2 Gestionnaire autonome

Un système autonome est muni d'éléments logiciels dédiés à son administration. Ces derniers lui permettent ainsi de s'auto-administrer et de minimiser l'intervention humaine. Ces éléments logiciels, responsables des tâches d'administration, sont appelés gestionnaires autonomes.

2.1.2.1 Définition de gestionnaire autonome

Un gestionnaire autonome est un élément logiciel qui implémente une ou plusieurs fonctions d'administration sur des éléments administrés de manière autonome. Parmi les fonctions d'administration figurent l'auto-configuration, l'auto-protection, l'auto-optimisation, l'auto-réparation. Les éléments administrés peuvent être des éléments logiciels et/ou matériels. Le gestionnaire implante une boucle de contrôle au moyen de capteurs et d'actionneurs. Les capteurs permettent de collecter les informations sur les éléments administrés. Les actionneurs, quant à eux, permettent d'appliquer les opérations de reconfiguration. Les capteurs et les actionneurs sont fournis avec les éléments administrés.

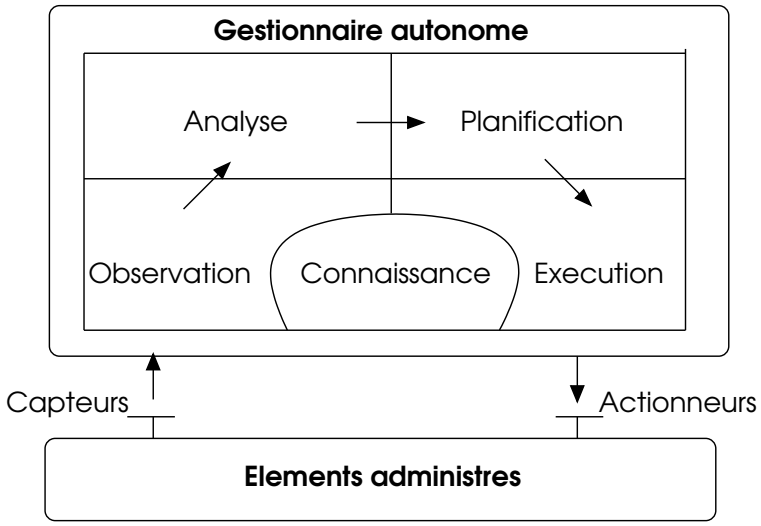

FIGURE 2.1 – Architecture d'un système autonome

L'implémentation d'un gestionnaire autonome est basée sur le modèle de référence, représenté à la figure 2.1, proposé par IBM, `MAPE-K` : Monitor (Obser-

vation), Analyze (Analyse), Plan (Planification), Execute (Exécution) et Knowledge (Connaissance). Ce modèle décrit quatre tâches principales :

1. **Observation.** Cette partie surveille l'exécution des éléments administrés grâce aux mesures provenant des capteurs. Elle permet d'agréger et de filtrer les informations de bas niveau en provenance des capteurs pour générer des données pertinentes destinées à l'analyse (e.g., panne, la surcharge ou la sous-charge d'un serveur).

2. **Analyse.** Elle implante les politiques de décision de la boucle de contrôle. Pour cela, elle analyse les données de surveillance et se base sur le modèle du système administré pour déterminer si des modifications sont nécessaires et va déclencher la phase de planification pour modifier le système.

3. **Planification.** Elle génère un programme de reconfiguration qui contient les tâches de reconfiguration à effectuer ainsi que leurs contraintes d'ordonnancement éventuelles.

4. **Exécution.** C'est le moteur qui exécute les tâches de reconfiguration en fonction des contraintes d'ordonnancement. Le moteur doit aussi assurer la cohérence du système lors d'une reconfiguration. Les tâches de reconfiguration sont exécutées grâce aux actionneurs des éléments administrés.

Connaissance (modèle du système). A l'exécution, un modèle du système administré est maintenu à jour. Ce modèle est utilisé par le gestionnaire autonome comme base pour les décisions d'administration. Il fournit une connaissance des éléments administrés. L'idéal serait qu'il fournisse également une connaissance sur les gestionnaires autonomes présents dans le système. Il existe différents types de modèles suivant la fonction d'administration implantée par le gestionnaire. On peut citer par exemple des modèles représentant la liste des attaques connues sur le système pour l'auto-protection ou bien des modèles basés sur la structure du système pour l'auto-réparation et l'auto-configuration.

2.1.2.2 Implémentation

L'administration autonome est une approche très prometteuse. Beaucoup de travaux de recherche ont démontré sa faisabilité à travers différentes expérimentations. Aujourd'hui de nombreux gestionnaires autonomes ont été conçus. Ils assurent de manière cohérente les tâches d'administration qu'ils implémentent. Parmi les aspects d'administration considérés figurent l'auto-configuration, l'auto-protection, l'auto-optimisation et l'auto-réparation.

Auto-réparation. Les frameworks proposés dans [18,53] implémentent l'auto-réparation. Ils permettent la restauration des composants logiciels de systèmes, à l'exécution. Dans [18], un modèle architectural du système administré est maintenu à jour. Il est utilisé comme base pour la reconfiguration et la réparation du système. Le modèle expose un ensemble de propriétés sur l'architecture du système, qui permettent de définir des contraintes formelles pour détecter des anomalies, et savoir quand une adaptation est nécessaire. La violation d'une contrainte entraîne la réparation du modèle. Les changements effectués sur le modèle sont ensuite appliqués sur les composants du système réel. La relation entre le modèle et le système réel est assurée au moyen de jauges et de services de traduction. Les jauges permettent l'évaluation des propriétés en fonction des mesures au niveau du système. Les services de traduction, eux, permettent d'interpréter les opérateurs au niveau de l'architecture et les mapper aux opérateurs de réparation sur les composants du système en exécution. La particularité du framework proposé est qu'il permet de choisir le style architectural utilisé pour la définition du modèle du système. L'objectif est de permettre d'utiliser le style qui correspond le mieux à l'architecture du système géré et qui expose les propriétés qui correspondent le mieux aux aspects d'administration. Des méthodes analytiques sont fournies pour la suggestion de stratégies de réparation appropriées. Les stratégies définissent des règles de réparation basées sur des opérateurs de haut niveau permettant l'adaptation du modèle. Le framework proposé dans [53] permet la réparation d'un système logiciel soumis à une haute variabilité. Il permet de surveiller le système pour détecter les fonctions qui ont échoué pour reconfigurer le système dans un état correct. La gestion de la réparation est basée sur un modèle d'objectifs composé d'objectifs et de tâches, structurés sous forme de graphe. Le framework est

constitué de quatre composants : un composant de surveillance, un composant d'analyse, un composant de reconfiguration et un composant d'exécution. Le composant de surveillance collecte, à l'exécution, les données logs du système géré. Ces données sont ensuite utilisées par le composant d'analyse pour identifier en cas d'échec les objectifs et les tâches qui ont échoué et la source de l'échec. Le composant de reconfiguration génère la meilleur configuration à laquelle reconfigurer le système à la prochaine exécution. La nouvelle configuration contient un ensemble de tâches dont la réussite de l'exécution mène à la satisfaction de l'objectif global. La nouvelle configuration est transmise au composant d'exécution. Ce dernier exécute des actions de compensation pour restaurer le système dans un état précédent consistant, puis il reconfigure le système en utilisant la nouvelle configuration calculée.

Auto-optimisation. Les approches proposées dans [1,12,46,49] implémentent l'auto-optimisation. Ils permettent de réagir aux variations de l'environnement de façon à obtenir un fonctionnement optimal du système administré. Le fonctionnement optimal est généralement défini en fonction de critères de performance et/ou de consommation de ressources. Ces approches permettent une allocation dynamique de ressources aux applications administrées hébergées sur une grappe de machines. En effet, une allocation statique peut mener à un gaspillage de ressources. [12, 46, 49] ont proposé une politique d'approvisionnement de ressources basée sur la charge de travail reçue en entrée. Ces approches reposent sur une surveillance de la charge de travail ou de métriques de qualités de service (SLA «Service Level Agreement») pour l'approvisionnement dynamique de ressources dans un environnement non virtualisé ou virtualisé. La ressource approvisionnée peut être du cpu, de la mémoire ou une machine entière. Dans le cas de système basé sur la réplication, l'optimisation est basée sur un mécanisme d'approvisionnement dynamique qui prend en compte la charge de travail courante afin d'ajuster le nombre de serveurs actifs en démarrant de nouveaux serveurs sur des machines disponibles, ou en arrêtant des serveurs et les machines qui les hébergent quand ils ne sont plus nécessaires. [1] propose `ElastMan`, un gestionnaire autonome pour l'élasticité d'applications web dans un environnement de *cloud computing*. Il permet l'approvisionnement et le retrait dynamique des ressources allouées à une ap-

plication afin de garantir les objectifs de niveau de service requis (SLOs «Service Level Objective») à un coût réduit. L'approche combine un contrôle prédictif et un contrôle rétroactif. Le contrôle prédictif permet de détecter et de répondre rapidement à des pics de charge. Le contrôle rétroactif permet de corriger l'écart vis-à-vis de la qualité de service souhaitée.

Auto-protection. Le framework JADE a été proposé dans [26] pour implémenter l'auto-protection d'un système distribué. L'auto-protection est mise en oeuvre en plaçant devant chaque élément logiciel du système un pare-feu. Chaque pare-feu est configuré pour accepter les requêtes autorisées à être traitées par le logiciel associé. Les pare-feux sont également configurés comme détecteurs d'intrusions. En cas d'intrusion, JADE isole les éléments impactés (e.g., les machines "infectées"). Il reconfigure de manière autonome l'architecture du système en remplaçant les éléments impactés et en reconfigurant les pare-feux grâce à un modèle de l'architecture du système maintenu à jour. JADE construit une représentation, à base de composants, du système distribué à administrer. Chaque élément administré du système est encapsulé dans un composant. Le système distribué est administré comme une architecture à composants. JADE est basé sur le modèle à composants Fractal [14] qui fournit des fonctionnalités d'introspection et de reconfiguration dynamique. Chaque composant implémente la même interface d'administration pour l'élément qu'il encapsule. Cela permet de gérer de manière homogène les éléments à administrer en évitant d'utiliser leurs interfaces d'administration complexes et propriétaires.

JADE est un système d'administration autonome. Il permet d'ajouter des comportements autonomes à des systèmes distribués (auto-protection [26], auto-réparation [11] et auto-optimisation [10]). Les gestionnaires autonomes agissent sur les interfaces fournies par les composants qui encapsulent les éléments administrés pour surveiller et appliquer des opérations d'administration sur le système réel à l'exécution. JADE permet d'ajouter plusieurs gestionnaires autonomes sur un même système. Toutefois les gestionnaires s'exécutent de manière indépendante sans aucune coordination. TUNe [13] est une amélioration de JADE. Il permet de masquer la complexité liée à la maîtrise des API de programmation du modèle à composants en fournissant un langage de descrip-

tion de l'architecture du système à administrer. Cependant la coordination des gestionnaires n'est également pas traitée dans TUNe.

Auto-configuration. Cette propriété peut être considérée comme traitée dans tous les travaux qui se sont intéressés à l'auto-administration et qui ont proposé une implémentation. En effet, qu'il s'agisse d'optimisation, de réparation ou de protection, une nouvelle configuration du système administré est générée et appliquée à travers les actions d'administration.

2.1.3 Coordination de gestionnaires autonomes

Aujourd'hui beaucoup de gestionnaires autonomes sont disponibles. Cependant chacun n'implémente qu'une partie des fonctions d'administration. Construire un système entièrement autonome dans lequel toutes les fonctions d'administration sont assurées requiert l'utilisation de plusieurs gestionnaires. Cela permet une gestion complète des différents aspects d'administration. Leur coexistence est rendue nécessaire par la difficulté de concevoir un gestionnaire qui couvre toutes les fonctions d'administration.

2.1.3.1 Besoin de coordination

La coordination des gestionnaires autonomes permet d'éviter des incohérences lors de l'administration d'un système. Individuellement chaque gestionnaire a un comportement cohérent mais leur coexistence peut amener des incohérences. Ils sont conçus indépendamment et n'ont généralement aucune connaissance de leur coexistence. De ce fait ils ne peuvent pas distinguer les changements causés par les phénomènes physiques (auxquels ils doivent réagir) des changements causés par les actions d'administration des uns des autres sur le système administré. Les actions d'un gestionnaire peuvent mener le système dans un état dans lequel ses objectifs peuvent être atteints alors que ceux des autres peuvent ne pas être atteints. Cela peut entraîner des actions de reconfiguration en répétition sans garantir ultimement l'atteinte des objectifs de tous les gestionnaires. Cela peut conduire à une instabilité du système administré.

2.1.3.2 Approches de coordination proposées

La coordination de gestionnaires autonomes a été étudiée dans plusieurs travaux de recherche. Des approches ont été proposées pour implémenter et mettre en oeuvre la coordination. Parmi les solutions proposées pour l'implémentation de la politique de coordination figurent des fonctions d'utilité, des fonctions d'optimisation, des protocoles de consensus et des règles (conditions/priorité).

Coordination basée sur une fonction d'utilité. Les approches proposées par [20, 40] sont basées sur des fonctions d'utilité. La solution de coordination proposée par [40], `vManage`, permet une gestion unifiée de la plate-forme d'exécution physique et de l'environnement virtualisé dans un centre de données. `vManage` comprend un service d'enregistrement, un service de proxy, des coordinateurs et un stabilisateur. Les services d'enregistrement et de proxy permettent la découverte et l'enregistrement des différents capteurs et actionneurs disponibles. Le but est de faciliter l'utilisation de capteurs et d'actionneurs divers, de différents constructeurs et d'unifier les dispositifs de surveillance et de contrôle de la plate-forme et de l'environnement virtualisé. Les politiques de gestion de l'environnement virtualisé basée sur l'état de la plate-forme sont implémentées par un coordinateur connecté au gestionnaire du système de virtualisation. Les politiques de gestion de la plate-forme basée sur l'état de l'environnement virtualisé sont, quant à elles, implémentées par des coordinateurs, et chacun est connecté à un gestionnaire de plate-forme exécuté sur chaque serveur physique supportant l'environnement virtualisé. Les coordinateurs utilisent les services d'enregistrement et de proxy pour accéder aux données de surveillance générées par les capteurs, et pour appliquer les politiques via les actionneurs. La stabilité est assurée par le stabilisateur qui empêche les coordinateurs d'effectuer des actions redondantes et inutiles. L'évaluation de la stabilité est basée sur une fonction de distribution cumulative. Cette dernière permet de calculer la probabilité que les serveurs physiques continuent à fournir suffisamment de ressources aux machines virtuelles qu'ils hébergent dans le futur durant une période de temps déterminée. Le prototype de coordination proposé par [20] permet la gestion de la performance et de la consommation énergétique dans un centre de données basé sur la répartition

de charge. Le prototype est basé sur des agents. Il comprend un agent responsable de la gestion de la performance, un agent responsable de la gestion de la consommation énergétique des serveurs et un agent de coordination. L'agent responsable de la gestion de la performance s'occupe de la répartition de la charge entre les différents serveurs actifs. L'agent responsable de la gestion de la consommation énergétique s'occupe d'ajuster la puissance des serveurs. Les actions de ces deux agents sont coordonnées par l'agent de coordination. La politique de coordination est basée sur un modèle du système et une fonction d'utilité multi-critères sur laquelle est basée le contrôle implémenté par l'agent de coordination.

Coordination basée sur des règles. Parmi les approches basées sur des règles, il y a [45] et [3]. La solution de coordination, `VirtualPower`, proposée par [45] permet la gestion de la consommation énergétique des ressources physiques en prenant en considération les politiques de gestion de ressources intégrées dans les machines virtuelles. Le but est de coupler la gestion de la performance au niveau des machines virtuelles et l'optimisation de l'énergie dans un centre de données. `VirtualPower` assure la satisfaction des politiques de gestion de ressources de machines virtuelles de manière indépendante et isolée via l'hyperviseur. Elle permet également la gestion globale de la puissance de calcul pour supporter l'environnement virtualisé en interprétant l'état des machines virtuelles pour la prise de décision. Le framework `Accord` proposé par [3] est basé sur les modèles à composants pour la construction d'applications autonomes. Les applications autonomes sont formées par la composition de composants autonomes qui intègrent des agents. Les agents implémentent des règles d'administration. Les décisions conflictuelles entre les agents sont résolus grâce à des priorités.

Coordination basée sur une fonction d'optimisation. Les approches proposées dans [51] et [38] sont basées sur des fonctions d'optimisation. [51] propose une extension de l'architecture `GANA` pour assurer la stabilité des boucles de contrôle dans un réseau autonome. Le framework introduit un nouveau module de synchronisation des actions (`ASM`). Ce module est intégré dans certains des éléments de décision (gestionnaire autonomes) qui se chargeront de

la coordination. Chaque module ASM gère une liste d'actions à synchroniser et dont il est responsable. Chaque gestionnaire avant d'exécuter une action consulte le module ASM responsable de l'action pour validation. Il connaît l'impact de chacune des actions sur les métriques de qualité de service. Un ASM sélectionne, parmi l'ensemble des actions à synchroniser, le sous-ensemble d'actions à exécuter qui assure la stabilité locale et globale du réseau. Le choix des actions à exécuter est formulé comme un problème d'optimisation basé sur la programmation linéaire binaire. Des indicateurs de performance sont définis, chacun avec un poids qui indique son importance. Un ASM choisit les actions qui optimise l'ensemble des indicateurs de performance. [38] adresse la stabilité dans un réseau autonome équipé de plusieurs boucles de contrôle. Il identifie trois problèmes qui doivent être considérés pour la stabilité : l'interaction des boucles de contrôle, la résolution de conflits entre les boucles de contrôle et la synchronisation. La théorie des jeux est proposée pour l'étude de la stabilité des comportements autonomes. La conception repose sur l'architecture GANA qui permet une structuration hiérarchique des boucles de contrôle. GANA permet également la résolution de conflits via un module, une fonction de synchronisation des actions [51].

Coordination basée sur un consensus. Parmi les approches basées sur un consensus, nous pouvons citer [2]. Elle propose un framework, LIBERO, qui permet l'implémentation de modèle comportemental de type *Pipeline* et *Farm* avec plusieurs gestionnaires autonomes coordonnés pour la gestion de plusieurs aspects non fonctionnels. La coordination des gestionnaires repose sur un consensus. Un gestionnaire qui planifie l'exécution d'une action demande la validation de l'action par les autres gestionnaires.

2.1.4 Synthèse

Une administration autonome globale requiert plusieurs gestionnaires autonomes pour couvrir tous les objectifs. Toutefois, la coordination des actions exécutées par les gestionnaires est essentielle pour éviter des décisions conflictuelles et garantir la cohérence de l'administration. Des solutions de coordination ont été proposées. Certaines solutions proposent des approches basées sur des fonctions d'optimisation, des fonctions d'utilité, des priorités ou un

consensus. Cependant, dans tous les cas, il y a un besoin de synchronisation et de contrôle des actions des gestionnaires.

La coordination peut être considérée comme un problème de synchronisation et de contrôle logique des actions d'administration. Une méthodologie de conception est d'utiliser les techniques issues des systèmes réactifs et des systèmes à événements discrets de la théorie du contrôle. Cette dernière est la discipline classique pour la conception de contrôleurs automatiques. Elles offrent des garanties sur le comportement du contrôleur vis-à-vis de propriétés désirables.

2.2 Modèles réactifs

Les systèmes réactifs [33] sont des systèmes qui interagissent continuellement avec leur environnement extérieur et au rythme imposé par ce dernier. Ils sont généralement concurrents de nature. Ils évoluent en parallèle avec leur environnement et ils sont souvent constitués de sous-systèmes évoluant en parallèle. Ces systèmes implémentent des fonctions critiques dont la validation de certaines propriétés de fonctionnement est requise avant la mise en exploitation. L'implémentation d'un système réactif avec les langages de bas niveau est souvent source d'erreurs et ne garantit pas la validation de propriétés sur le comportement du système.

Parmi les solutions proposées pour l'étude et la conception de systèmes réactifs figurent *STATECHARTS* [35], les *réseaux de Pétri* et les langages synchrones. *STATECHARTS* est un formalisme basé sur les diagrammes d'états. Il fournit des mécanismes pour représenter le parallélisme, la préemption et la hiérarchie. Les *réseaux de Pétri* permettent la modélisation et l'analyse qualitative de comportements parallèles, de synchronisation et de partage de ressources. Quant aux langages synchrones, ils fournissent des méthodes formelles pour la spécification de systèmes réactifs et disposent d'outils de vérification, de synthèse de contrôleurs discrets [42] et de génération de code exécutable à partir de la spécification. Nous basons notre travail sur ces langages.

2.2.1 Les langages synchrones

Les langages de programmation synchrone [5] sont des langages de haut niveau introduits au début des années 80 pour la conception de systèmes réactifs. Ils permettent une description de haut niveau du comportement d'un système et d'aborder les notions de concurrence et de déterminisme. Ils permettent une spécification formelle et disposent d'outils d'analyse offrant des garanties à la compilation sur le comportement du système à l'exécution.

Les langages synchrones reposent sur une hypothèse appelée l'*hypothèse synchrone*. L'hypothèse synchrone fournit un niveau d'abstraction où les réactions – calculs et/ou communications – du système sont instantanées. Cela permet la spécification du fonctionnement d'un système sans considérer les contraintes liées à l'architecture sur laquelle il est exécuté. L'évolution est basée sur la notion d'instant. Les événements internes et les événements de sortie sont datés précisément en fonction du flux des événements en entrée. Cela facilite au moment de la spécification de raisonner en temps logique sans tenir compte des temps réels des calculs et des communications. Cela facilite également le raisonnement par rapport aux aspects de déterminisme et de concurrence sur le comportement du système souvent décrit comme la composition parallèle de sous-systèmes.

Certains langages synchrones sont basés sur le modèle impératif. *ESTEREL* [7] est un exemple de langage qui adopte ce modèle. D'autres sont basés sur le modèle flot de données par exemple *LUSTRE* [34] et *SIGNAL* [6]. Dans le modèle flot de données toute variable manipulée est un flot, c'est-à-dire une séquence infinie de valeurs d'un même type. A chaque instant, une valeur est associée à chaque variable. Il existe d'autres langages qui permettent de décrire un système à base de modèles mixant des équations de flot de données et des automates (les automates de mode [41]). C'est le cas des langages `Heptagon` [27] et `Heptagon/BZR` [22, 24, 25].

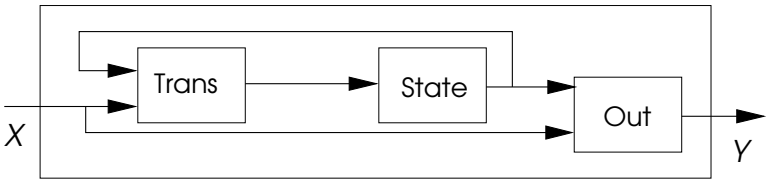

FIGURE 2.2 – Système de transitions

Toutefois, bien que basés sur des modèles différents, les programmes synchrones implantent le même comportement de base illustré par la figure 2.2. A chaque réaction, les valeurs des flux d'entrée X, ainsi que les valeurs courantes de la mémoire, sont utilisées pour calculer l'état suivant (fonction *Trans*), mettre à jour la mémoire (fonction *State*) et calculer les valeurs des flux de sortie Y (fonction *Out*).

2.2.1.1 Heptagon/BZR

`Heptagon/BZR` [1] [22, 24, 25] est un langage de programmation appartenant à la famille des langages synchrones. Il permet de décrire un système à base de modèles mixant des équations de flot de données et des automates [41]. Ce langage permet de décrire un système constitué de sous-systèmes par la composition parallèle et hiérarchique [19] des modèles des différents sous-systèmes. Les modèles évoluent en parallèle de manière synchrone : une réaction globale implique une réaction locale de chacun des sous-modèles.

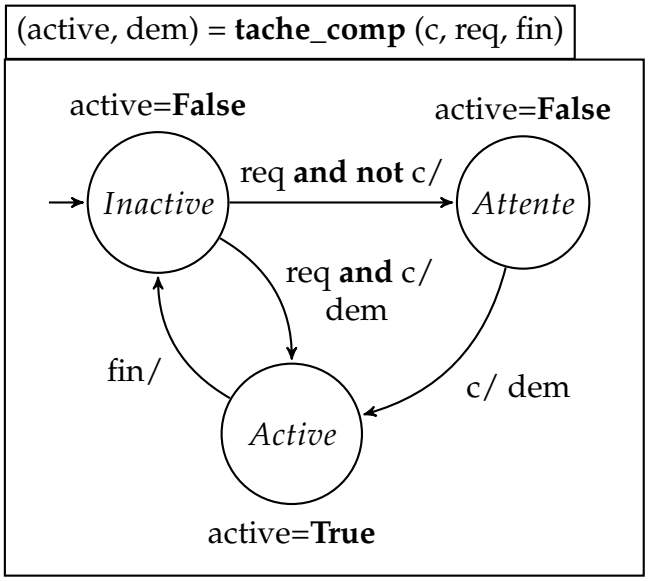

FIGURE 2.3 – Modélisation avec Heptagon/BZR : Tâche différable

La Figure 2.3 présente un exemple simple de modélisation avec le langage `Heptagon/BZR`. Le programme `tache_comp` modélise le comportement contrôlable d'une tâche (e.g., un processus). L'activation de la tâche peut être

1. `http://bzr.inria.fr`

```
node tache_comp(req, c, fin:bool)
                    returns (active, dem:bool)
let automaton
    state Inactive do
              active = false ;
              dem = req and c
         until dem then Active
             | req and not c then Attente
    state Attente do
             active = false ;
             dem = c
          until dem then Active
    state Active do
             active = true ;
             dem = false
         until fin then Inactive
   end
tel
```

FIGURE 2.4 – Modélisation avec `Heptagon/BZR` : programme BZR.

différée. Initialement la tâche n'est pas activée. Cet état est représenté dans le modèle par l'état `Inactive`. Dans cet état, lorsque l'activation de la tâche est demandée (`req` à vrai), si l'activation est autorisée (`c` à vrai) la tâche est activée et passe dans l'état `Active` ; sinon l'activation est retardée en attente de l'autorisation. L'attente de l'autorisation est représentée par l'état `Attente`.

La sortie `dem` indique le déclenchement de l'activation de la tâche dans le système. La sortie `active` indique l'état de la tâche, elle est à vrai lorsque la tâche est en cours d'exécution. La fin de l'activation de la tâche est représentée par l'entrée `fin` à vrai. La figure 2.4 présente le programme `Heptagon/BZR` correspondant à l'automate décrit dans la figure 2.3.

`Heptagon/BZR`, comme tous les autres langages synchrones, permet de modéliser un système par la composition parallèle d'automates, illustrée par la figure 2.5. Chaque automate décrit une partie, un sous-système, du système global. Cela facilite la modélisation de systèmes de grande taille. Les états et les transitions de l'automate produit de la composition est l'ensemble des combinaisons d'états et de transitions possibles des automates composés en parallèle.

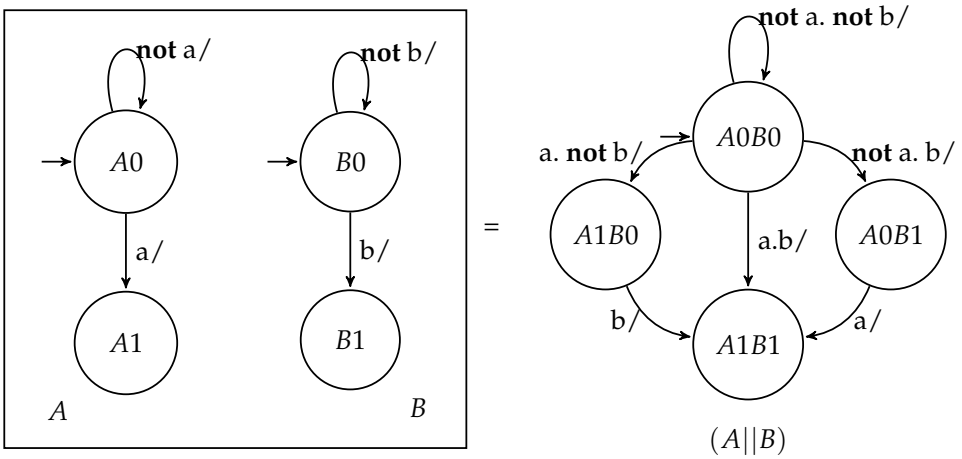

FIGURE 2.5 – Exemple de composition parallèle

Toutefois la composition parallèle ne fait aucune synchronisation entre les automates composés. La composition parallèle est appropriée pour la modélisation du comportement parallèle de systèmes indépendants. Lorsque les systèmes modélisés par les automates doivent communiquer ou se synchroniser les uns avec les autres, la composition parallèle doit être utilisée avec l'encapsulation de certains signaux (entrées/sorties des automates) dédiés.

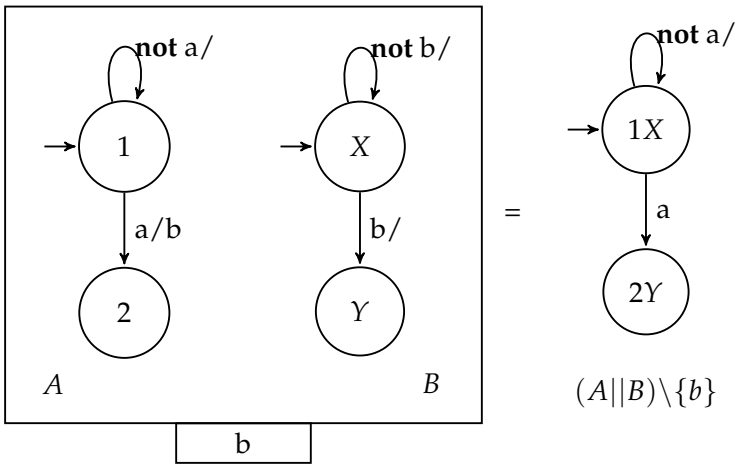

FIGURE 2.6 – Exemple d'encapsulation

L'encapsulation, illustrée à la figure 2.6, est une opération qui permet d'assurer la synchronisation entre deux automates composés par une variable qui est une sortie d'un des automates et une entrée pour l'autre. Dans cet exemple,

la variable encapsulée est la variable `b` qui permet ainsi de synchroniser les automates. Cependant, cette opération peut poser des problèmes de causalité lorsque les automates communiquent de manière bidirectionnelle dans la même réaction.

2.2.1.2 Implémentation des programmes synchrones

L'implémentation des programmes synchrones peut être soit matérielle soit logicielle. Dans le cas d'une implémentation logicielle, le compilateur produit un programme séquentiel exécutable dans un langage de programmation cible (C ou Java). Ce sont en général des langages de programmation impératifs, permettant une intégration aisée du code obtenu dans le système, d'où le choix de langages largement utilisés et indépendants de la plate-forme d'exécution. La génération de code en `Java` [2] produit une ou plusieurs classes Java avec une classe principale. La classe principale, tout comme les autres classes, fournit un *constructor* qui permet de créer une instance deux méthodes : *reset* et *step*. La méthode *reset* ne prend aucun paramètre et permet d'initialiser l'état interne global, e.g., les variables internes et les variables de sortie. La méthode *step* implémente le comportement réactif et permet d'effectuer un pas d'exécution global. Elle met à jour l'état interne et retourne un résultat qui correspond aux sorties. Elle prend comme argument les entrées décrites dans le modèle global.

Le compilateur des langages synchrones produit un code transformationnel ce qui implique qu'il doit être invoqué explicitement. La méthode *step* doit être appelée explicitement avec les bonnes entrées et fréquemment pour obtenir le comportement réactif. Le résultat produit par chacun des appels à la méthode *step* doit également être interprété et exécuté. De ce fait, il est nécessaire d'implémenter une interface [4] facilitant le dialogue avec la méthode *step*. A l'exécution, cette interface collecte les valeurs à passer à la méthode *step*, fait les invocations à la méthode et interprète le résultat retourné en terme d'actions à exécuter. Il existe deux modèles pour la mise en oeuvre du comportement réactif :

- **Modèle général** : Dans ce modèle, le comportement réactif est basé sur l'occurrence des événements. Toute occurrence d'événement conduit à l'invocation de la méthode *step*.

2. Nous utilisons le langage Java dans nos expérimentations

– **Modèle périodique** : Dans ce modèle, le comportement réactif repose sur une réaction par période. L'invocation de la méthode *step* est périodique. Les événements sont collectés et conservés jusqu'à l'appel de la méthode.

Un exemple de branchement d'un programme synchrone est présenté dans [8] pour le contrôle des pilotes de périphériques dans un système embarqué. L'objectif est d'appliquer des politiques de reconfiguration des périphériques permettant une gestion globale de la consommation de l'énergie du système. Ce travail montre un exemple d'intégration d'un programme synchrone dans un système réel. Il montre comment le *step* est utilisé.

2.2.2 Synthèse de contrôleur discret (SCD)

Parmi les méthodes de conception et de validation, la synthèse de contrôleur est l'une des plus séduisantes. Elle permet de raffiner une spécification incomplète de manière à atteindre un certain objectif comme la satisfaction d'une propriété non encore vérifiée par le système initial. La synthèse de contrôleur, issue de la théorie du contrôle, permet d'obtenir une logique de contrôle par construction [44]. Elle est basée sur des méthodes formelles pour la synthèse du contrôleur assurant le respect de propriétés sur un système contrôlé. Elle requiert un modèle du comportement du système à contrôler et une spécification des propriétés exprimées en terme d'objectifs de contrôle, par exemple l'invariance. Le modèle du système décrit de manière formelle tous les comportements possibles, les comportements corrects et incorrects vis-à-vis des propriétés désirées. Il décrit également la contrôlabilité du système. Le système à contrôler, ainsi que les objectifs de contrôle, sont généralement modélisés au moyen de systèmes de transitions étiquetés ou automates [48], et les langages synchrones sont bien adaptés.

Principe. La synthèse de contrôleur construit une logique de contrôle, une contrainte sur les valeurs des entrées contrôlables du système à contrôler, en fonction de son état courant et des valeurs des entrées incontrôlables, de sorte que tous les comportements autorisés satisfassent les propriétés définies comme objectifs de contrôle. La logique de contrôle construite restreint le moins possi-

ble le fonctionnement du système contrôlé.

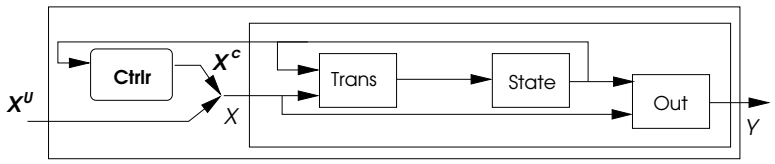

FIGURE 2.7 – Système de transitions contrôlé

La figure 2.7 présente un exemple où le système de transition de la figure 2.2 est le système à contrôler. Le système de transition prend en entrée $X = (X^u \cup X^c)$ à chaque réaction. Les entrées X^u sont incontrôlables alors que les entrées X^c sont contrôlables. Le contrôleur (logique de contrôle) `Ctrlr`, obtenu par synthèse de contrôleur, produit les valeurs à affecter aux variables contrôlables X^c en se basant sur les valeurs des entrées incontrôlables X^u et l'état courant du système afin d'assurer les objectifs de contrôle. Toutefois, il peut arriver qu'il n'existe pas de solution si le système n'est pas assez contrôlable par rapport aux objectifs de contrôle.

Le code exécutable correspondant au modèle contrôlé décrit à la figure 2.7 constitue un contrôleur réel. Il permet à l'exécution de contrôler le système modélisé lorsqu'ils sont couplés.

2.2.2.1 Synthèse de contrôleur avec Heptagon/BZR

`Heptagon/BZR` intègre un outil de synthèse de contrôleur discret `SIGALI` [3] [44] dans sa compilation. Il permet une utilisation facile de la synthèse de contrôleur en introduisant la notion de contrat dans la modélisation de système. Le contrat est décrit de manière déclarative [24]. Il est constitué de trois parties : **`assume`**, **`enforce`** et **`with`**.

Le contrat contient les propriétés que le fonctionnement du système doit respecter. Ces propriétés sont déclarées comme objectifs de contrôle dans la

3. `http://www.irisa.fr/vertecs/Logiciels/sigali.html`

partie **enforce**. Lorsque le modèle qui décrit le fonctionnement du système à contrôler ne garantit pas le respect des propriétés, Heptagon/BZR génère une logique de contrôle qui permet d'assurer le respect des propriétés lorsque des entrées contrôlables sont définies dans le modèle. Les variables contrôlables dans le modèle du système sont déclarées comme variables locales contrôlables dans la partie **with** du contrat. La logique de contrôle qui assure le respect des propriétés détermine les valeurs à assigner à ces variables contrôlables de sorte à restreindre le fonctionnement aux comportements qui satisfont les propriétés. Les propriétés pertinentes concernant l'environnement d'exécution sont déclarées dans la partie **assume** du contrat. Cette information est prise en compte lors de la synthèse de la logique de contrôle.

$(\texttt{active}_1, \texttt{dem}_1, \texttt{active}_2, \texttt{dem}_2) = \texttt{deuxtaches}(\texttt{req}_1, \texttt{fin}_1, \texttt{req}_2, \texttt{fin}_2)$

assume true

enforce not $(\texttt{active}_1$ **and** $\texttt{active}_2)$

with c_1, c_2

$(\texttt{active}_1, \texttt{dem}_1) = \texttt{tache_comp}_1(\texttt{c}_1, \texttt{req}_1, \texttt{fin}_1)$;

$(\texttt{active}_2, \texttt{dem}_2) = \texttt{tache_comp}_2(\texttt{c}_2, \texttt{req}_2, \texttt{fin}_2)$

FIGURE 2.8 – Heptagon/BZR contrat : exclusion mutuelle

La figure 2.8 présente un exemple de programme auquel est associé un contrat. Ce programme modélise le contrôle de deux tâches. L'exécution de chacune des tâches peut être retardée. Les deux tâches sont modélisées par deux instances du programme à la figure 2.3. Le contrat associé à ce programme consiste à n'autoriser l'activation d'une tâche que lorsque l'autre tâche n'est pas en cours d'exécution. Les deux tâches ne doivent pas être actives en même temps. De ce fait, l'objectif est de contrôler le démarrage de l'exécution des tâches de sorte que les deux tâches ne soient pas actives en même temps. Cela est exprimée par la propriété «**not** $(active_1$ **and** $active_2)$», avec $active_1$ à vrai (**true**) lorsque la tâche n1 est active et $active_2$ à vrai (**true**) lorsque la tâche n2 est active. Cette propriété étant l'objectif de contrôle à garantir est déclarée dans la partie **enforce** du contrat. Les variables c_1 et c_2, déclarées comme étant contrôlables dans la partie **with**, vont être utilisées par la logique de contrôle synthétisée pour empêcher l'activation de l'exécution d'une des tâches lorsque l'autre est en cours d'exécution.

A la compilation Heptagon/BZR invoque SIGALI pour la synthèse de la logique de contrôle. Une fois la logique de contrôle générée, Heptagon/BZR l'intègre dans le modèle et produit un programme exécutable. Ce dernier constitue un contrôleur qui permet de contrôler l'exécution des deux tâches modélisées pour respecter la propriété. Heptagon/BZR permet de produire une seule solution de manière déterministe pour un problème de synthèse de contrôleur discret. Le compilateur de Heptagon/BZR favorise, pour les variables contrôlables, la valeur vrai (**true**) à faux (**false**) et en prenant en compte l'ordre de déclaration des variables.

2.2.2.2 Synthèse modulaire avec Heptagon/BZR

La synthèse de contrôleur est de complexité exponentielle. Elle est basée sur l'exploration de l'espace d'états [16, 44] pour construire la logique de contrôle. Cela limite son passage à l'échelle concernant son application sur des modèles de systèmes larges. La synthèse modulaire permet d'adresser des systèmes larges et complexes. Elle permet une spécification décentralisée du contrôle d'un système. Un système est généralement modélisé par la composition de sous-modèles qui peuvent également être la composition de sous-modèles. La synthèse modulaire exploite cette structure du système [43] pour réduire la complexité en offrant la possibilité de définir dans chacun des sous-modèles les objectifs de contrôle à assurer sur les comportements qu'il encode. Chaque sous-modèle intègre une logique de contrôle qui assure le respect de propriétés vis-à-vis de son contexte local, et le respect de propriétés globales vis-à-vis de sa composition avec d'autres sous-modèles.

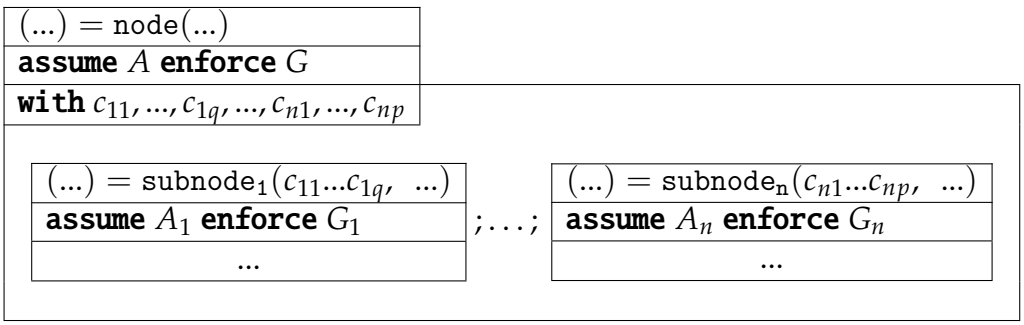

FIGURE 2.9 – Synthèse modulaire avec Heptagon/BZR.

La figure 2.9 présente un exemple graphique de la spécification modulaire du contrôle en `Heptagon/BZR`. Chaque sous-modèle, $\texttt{subnode}_i$, inclut un contrat qui contient des objectifs de contrôle G_i à assurer localement. Le modèle global `node` inclut un contrat qui contient les objectifs de contrôle global G à assurer sur l'ensemble. Le contrat de `node` utilise les entrées contrôlables c_{ij} des sous-modèles $\texttt{subnode}_i$ comme variables contrôlables pour assurer le respect des objectifs globaux.

Principe. La synthèse modulaire dans `Heptagon/BZR` est basée sur l'utilisation des contrats des sous-modèles comme abstraction de leurs comportements. [24] et [21] fournissent une description formelle détaillée de cette approche. Néanmoins nous décrivons le principe à travers l'exemple présenté à la figure 2.9.

La synthèse modulaire permet de construire indépendamment la logique de contrôle à intégrer dans chaque sous-modèle $\texttt{subnode}_i$. La synthèse ne dépend que des entrées et du comportement qu'il encode (section 2.2.2) en plus des objectifs de contrôle. Lorsqu'une partie des objectifs de contrôle G_i définis dans le contrat d'un sous-modèle $\texttt{subnode}_i$ concerne l'application du contrôle reçu via ses entrées contrôlables c_{ij}, alors la logique de contrôle construite pour ce modèle assure le respect du contrôle externe.

Lors de la réutilisation d'un sous-modèle dans un contexte global, comme `node`, son contrat offre une garantie de l'application du contrôle externe du comportement qu'il encode. Cela permet d'utiliser le contrat comme abstraction du comportement. De ce fait pour la construction la logique de contrôle pour le modèle global `node`, on peut supposer, en plus de A, le respect du contrôle appliqué sur chaque sous-modèle $\texttt{subnode}_i$ via ses entrées contrôlables c_{ij} pourvu que la supposition A_i définie dans son contrat soit respectée : $\bigwedge_{\mathtt{i}=1}^{\mathtt{n}}(\mathtt{A_i} \implies \mathtt{G_i})$. Par conséquent il n'est plus nécessaire de parcourir les comportements encodés dans les sous-modèles pour la synthèse de la logique de contrôle pour le modèle `node`. Cela présente comme avantage la diminution de la taille de l'espace d'états à explorer pour le modèle global. Le problème de synthèse consiste alors à construire une logique de contrôle qui assure le respect de G et aussi le respect de $\bigwedge_{i=1}^{n} A_i$.

Nous détaillons dans le chapitre suivant comment les objectifs associés aux

entrées de contrôle sont définis dans le contrat d'un modèle dans le cadre de ce travail de thèse.

Réduction de la complexité. La synthèse modulaire permet d'appliquer la synthèse sur des modèles simples avec peu d'états à explorer et de les composer de manière modulaire et hiérarchique. Elle offre également la possibilité d'appliquer la synthèse sur les compositions de modèles en considérant le contrat des sous-modèles comme abstraction des comportements qu'ils encodent. Cela diminue le nombre d'états à explorer par synthèse et par conséquent réduit la complexité.

La synthèse est appliquée sur chacun des modèles de manière indépendante que ce soit un modèle simple ou une composition. La synthèse modulaire sur une composition requiert uniquement le contrat défini dans les sous-modèles. Ainsi l'application de la synthèse modulaire sur un modèle global ne nécessite pas l'application de la synthèse modulaire sur les sous-modèles qui le constituent. Une fois qu'une logique de contrôle est générée pour un modèle, il n'est plus nécessaire, lors de sa réutilisation, d'appliquer à nouveau la synthèse sur ce modèle à moins que ce dernier ait subi des modifications.

Cependant l'abstraction des détails internes des sous-modèles diminue les solutions possibles qui peuvent être construites pour un problème donné, comparé à l'approche monolithique. De plus, comme pour la synthèse monolithique, il peut arriver qu'il n'existe pas de solution également dans le cas de la synthèse modulaire.

Réutilisation de code exécutable. La compilation des modèles est également effectuée de manière modulaire. La logique de contrôle qui satisfait les objectifs de contrôle d'un modèle est construite et intégrée dans le modèle ; et le code exécutable correspondant à l'ensemble est généré. Le code reste inchangé et réutilisable tel quel dès lors que la spécification ne change pas. La recompilation n'est nécessaire que si la spécification change.

2.2.3 Synthèse

La synthèse de contrôleur discret (SCD) permet de construire un contrôleur qui, mis en parallèle avec le système qu'on veut contrôler, le restreint aux

comportements qui satisfont les propriétés désirées. Son application requiert une modélisation du système à contrôler et une spécification des propriétés à garantir qui sont exprimées sous forme d'objectifs de contrôle. Généralement le système est décrit par un système de transition ou un automate avec des langages de haut niveau comme les langages synchrones. Ces langages permettent une description formelle du comportement d'un système sous forme d'automate (aspects fonctionnels et/ou non fonctionnels). Ils permettent de modéliser un système par la composition parallèle et hiérarchique de modèles simples. Ils fournissent des compilateurs puissants qui permettent la génération automatique de code exécutable à partir de la spécification du système. Cela permet de réduire l'écart entre la spécification d'un système et son implémentation.

La synthèse modulaire permet une spécification décentralisée en offrant la possibilité d'appliquer la synthèse sur des modèles simples avec peu d'états à explorer et de les composer de manière modulaire et hiérarchique. Elle offre également la possibilité d'appliquer la synthèse sur les compositions de modèles en considérant le contrat des sous-modèles comme abstraction des comportements qu'ils encodent. Cela permet de diminuer le nombre d'états à explorer par synthèse et par conséquent réduit la complexité.

2.3 Conclusion

L'automatisation des fonctions d'administration des systèmes informatiques est un sujet qui suscite encore beaucoup d'intérêt en recherche. Les travaux déjà effectués démontrent sa faisabilité à travers différentes expérimentations. Aujourd'hui de nombreux gestionnaires autonomes sont implémentés et assurent de façon cohérente les fonctions d'administration ; mais aucun n'assure une administration complète. Cela rend leur coexistence nécessaire pour une administration globale. Toutefois la coordination de leur coexistence est importante pour assurer une administration cohérente. Beaucoup de travaux de recherche se sont intéressés à la coordination de gestionnaires autonomes. Différentes approches ont été proposées et évaluées à travers des expérimentations. Cependant nous remarquons que toutes ces approches de coordination requièrent la

synchronisation des gestionnaires et le contrôle de leurs actions, et ces aspects ont été largement étudiés en théorie du contrôle discret.

La théorie du contrôle et les outils qui en résultent ont récemment commencé à être utilisés pour les systèmes informatiques. La plupart des cas d'utilisation reposent sur des modèles continus ; généralement pour traiter des aspects quantitatifs [36, 37, 47]. Des utilisations plus récentes reposent sur des modèles de la famille des systèmes à Événements Discrets [16] sur lesquels des propriétés logiques sont étudiées. Ils utilisent les notions de contrôle supervisé [48], généralement pour garantir des propriétés logiques ou à des fins de synchronisation [52]. Le contrôle discret est basé sur des modèles sous la forme de systèmes de transitions, comme les réseaux de Petri ou automates. Il fournit des langages de haut niveau, comme les langages synchrones, pour la spécification formelle de système ; et des outils de vérification et de synthèse de contrôleur.

Dans ce travail de thèse, nous nous intéressons à l'application des techniques issues de la théorie du contrôle pour la coordination de gestionnaires autonomes. Nous utilisons la synthèse de contrôleur discret pour la construction automatique de contrôleur de coordination. La construction du contrôleur est basée sur le modèle de la coexistence des gestionnaires et une spécification de contrôle. Nous utilisons la programmation synchrone pour la modélisation des gestionnaires et de leur coexistence (coordonnée par SCD) et pour l'implémentation en Java du contrôleur de coordination.

3

Méthodologie de coordination de gestionnaires autonomes

Contents

3.1 Spécification de la coordination 36
3.1.1 Modélisation d'un gestionnaire autonome 36
3.1.1.1 Comportement 37
3.1.1.2 Contrôlabilité 38
3.1.2 Modélisation de la coordination 39
3.1.2.1 Modélisation de la coexistence 39
3.1.2.2 Spécification d'une stratégie de coordination 40
3.1.3 Modélisation modulaire de la coordination 41
3.1.3.1 Contrôle décentralisé 41
3.1.3.2 Spécification modulaire et hiérarchique 42
3.2 Mise en oeuvre de la coordination 44
3.2.1 Le modèle à composants Fractal 44
3.2.1.1 Composant Fractal 44
3.2.1.2 Introspection et reconfiguration 46
3.2.1.3 Fractal ADL 46
3.2.2 Composant de gestionnaire autonome 47
3.2.3 Coordination à base de composants 48
3.2.3.1 Coordination de gestionnaires 48
3.2.3.2 Coordination hiérarchique 50

3.3 Comparaison **52**
3.4 Conclusion **53**

Ce chapitre détaille notre méthodologie pour la coordination de gestionnaires autonomes. Notre approche est basée sur l'utilisation de méthodes et de techniques issues du contrôle discret. Nous utilisons la technique de synthèse de contrôleur discret. Cette dernière permet de construire automatiquement un contrôleur qui est capable de restreindre le comportement d'un système pour garantir le respect de spécifications logiques de fonctionnement. La synthèse de contrôleur discret est basée sur un modèle du système à contrôler et une spécification des objectifs de contrôle désirés. Pour la modélisation, nous utilisons la programmation synchrone qui fournit des langages de haut niveau facilitant la spécification formelle de système. Elle fournit également des outils de vérification et de génération de code exécutable.

Dans ce travail le système à contrôler est un système d'administration constitué de plusieurs gestionnaires autonomes indépendants. Le contrôle consiste à restreindre le comportement des gestionnaires afin de garantir la cohérence des actions d'administration appliquées sur le système qu'ils administrent. Pour la mise en oeuvre du contrôle des gestionnaires, nous adoptons les modèles à composants. Nous utilisons le modèle à composants Fractal, plus précisément l'implémentation Java, Julia [15].

3.1 Spécification de la coordination

La coordination repose sur la connaissance du comportement des gestionnaires, de leurs fonctionnalités contrôlables, et des événements pertinents auxquels ils réagissent. Le comportement des gestionnaires doit être observable et contrôlable à l'exécution pour permettre de contrôler leurs actions.

3.1.1 Modélisation d'un gestionnaire autonome

Nous modélisons le comportement observable et contrôlable de chaque gestionnaire autonome, avec un niveau d'abstraction qui présente les états et transitions pertinents pour le problème de contrôle – la coordination. Le

modèle d'un gestionnaire expose des variables contrôlables. Ces dernières correspondent aux points de choix sur le comportement du gestionnaire ; c'est à travers eux qu'un contrôle peut être appliqué pour faire respecter une politique de coordination. Pour modéliser le comportement d'un gestionnaire, nous utilisons la programmation synchrone. Nous modélisons le comportement de chaque gestionnaire indépendamment des autres.

3.1.1.1 Comportement

Le comportement d'un gestionnaire est modélisé à base d'automates. Chaque état du modèle décrit une situation dans laquelle le gestionnaire est dans un mode d'exécution défini, e.g., exécute des actions de reconfiguration ou surveille l'état du système administré. Les transitions correspondent, quant à elles, aux changements d'états du gestionnaire suite à l'occurrence d'événements auxquels il réagit. Chaque transition est étiquetée avec les événements en entrée qui l'activent et les actions en sortie produites par le gestionnaire. Dans les modèles à base d'automates, les réactions sont considérées instantanées. Cependant, dans un gestionnaire, une réaction implique l'exécution d'une (ou plusieurs) action non instantanée. De ce fait, nous ne considérons pas toujours l'exécution d'une action comme instantanée. Elle l'est ou pas selon sa pertinence dans la description du comportement du gestionnaire. Lorsqu'elle n'est pas considérée instantanée, elle sera alors représentée par un état distinct dans le modèle du gestionnaire.

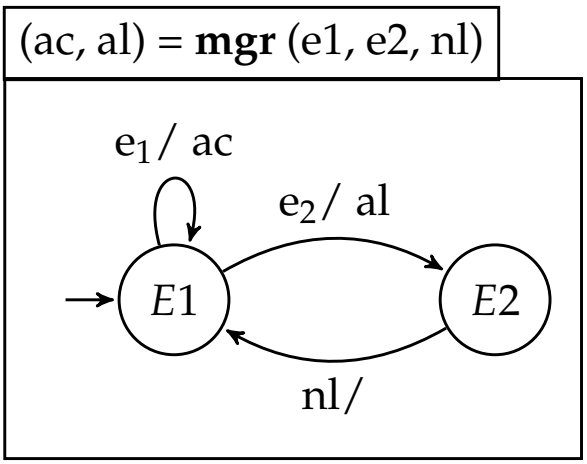

FIGURE 3.1 – Comportement d'un gestionnaire

La figure 3.1 présente un exemple simple de modèle du comportement d'un gestionnaire. Le gestionnaire a deux états représentés par $E1$ et $E2$. Initialement il est dans l'état $E1$. Dans cet état, le gestionnaire réagit en présence de l'événement $e1$ en produisant l'action courte ac en réponse. En présence de l'événement $e2$, il réagit en produisant l'action al, qui est une action longue, et se met dans l'état $E2$. L'état $E2$ représente l'exécution de l'action al qui n'est pas considérée instantanée contrairement à l'exécution de l'action ac. A la fin de l'exécution de l'action al, représentée par la présence de l'événement nl de notification de fin d'exécution de l'action al, le gestionnaire retourne dans l'état $E1$.

3.1.1.2 Contrôlabilité

Nous définissons la contrôlabilité d'un gestionnaire comme la capacité à autoriser ou inhiber ponctuellement certaines de ses actions suite à l'occurrence d'événements auxquels il réagit. Dans le modèle d'un gestionnaire, la contrôlabilité est représentée par des entrées de contrôle associées aux transitions. Ainsi pour décrire une transition contrôlable, nous lui ajoutons une entrée de contrôle. Selon la valeur affectée à cette entrée, la transition sera autorisée ou non en présence de l'événement qui l'active. Le contrôle d'un gestionnaire nécessite également que son état soit observable. Cette propriété est représentée, dans le modèle, par des variables d'état qui indiquent l'état courant du gestionnaire.

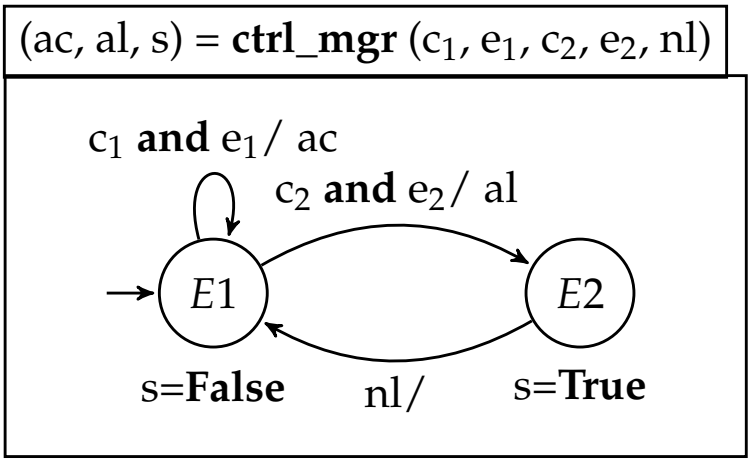

FIGURE 3.2 – Gestionnaire contrôlable

La figure 3.2 présente le modèle qui décrit la contrôlabilité du comportement

du gestionnaire présenté dans la figure 3.1. Les entrées $c1$ et $c2$ représentent les entrées contrôlables du gestionnaire. Elles permettent, respectivement, le contrôle des actions *ac* et *al* du gestionnaire. L'état courant du gestionnaire est indiqué par la variable d'état *s*. Cette dernière est à vrai lorsque l'action *al* est en cours d'exécution.

3.1.2 Modélisation de la coordination

La composition des modèles des gestionnaires autonomes reflète leur coexistence non coordonnée. Pour modéliser la coordination de la coexistence des gestionnaires, nous associons un contrat à la composition des modèles des gestionnaires à coordonner. Ce contrat décrit les objectifs de contrôle – la politique de coordination – à atteindre sur la composition. Grâce à ce contrat, la synthèse de contrôleur pourra être appliquée pour construire automatiquement une logique de contrôle qui est capable d'agir sur les points de choix des modèles des gestionnaires pour respecter la politique de coordination. La composition des modèles des gestionnaires autonomes couplée avec la logique de contrôle, ensemble elles vérifient la politique de coordination. Elles modélisent la coordination des gestionnaires.

3.1.2.1 Modélisation de la coexistence

Les modèles des gestionnaires constituent les briques du modèle de leur coexistence. La composition des modèles des gestionnaires décrit l'ensemble des comportements possibles lors de la coexistence des gestionnaires. Elle comprend les comportements cohérents ainsi que les comportements incohérents qui peuvent conduire le système qu'ils administrent dans un état inconsistant.

$$(s_1, s_2, ac_1, ac_2, ...) = \textbf{comp_mgrs}\ (c_{11}, e_{11}, c_{21}, e_{21}, ...)$$

$$(ac_1, al_1, s_1) = \textbf{ctrl_mgr}_1\ (c_{11}, e_{11}, c_{12}, e_{12}, nl_1)$$
$$(ac_2, al_2, s_2) = \textbf{ctrl_mgr}_2\ (c_{21}, e_{21}, c_{22}, e_{22}, nl_2)$$

FIGURE 3.3 – Modèle de la coexistence de gestionnaires

La Figure 3.3 présente un exemple de modèle de la coexistence de deux gestionnaires. Nous composons deux instances du modèle présenté à la figure 3.2. Cette composition exhibe l'ensemble des états observables s, l'ensemble des événements e auxquels réagissent les gestionnaires, l'ensemble des actions a, mais également l'ensemble des entrées de contrôle c disponibles pour le contrôle des gestionnaires. Toutefois, aucune stratégie de coordination n'est encore définie dans ce modèle global. Tous les comportements sont possibles.

3.1.2.2 Spécification d'une stratégie de coordination

La stratégie de coordination est exprimée sous forme de contrat associé au modèle global qui décrit la coexistence de gestionnaires autonomes à coordonner. Le contrat contient un ensemble d'objectifs de contrôle. Le respect de ces objectifs de contrôle est assuré en agissant sur les entrées de contrôle disponibles sur les modèles des gestionnaires. Ces entrées correspondent aux points de choix qui permettent de contrôler les actions des gestionnaires. Elles sont déclarées comme variables locales contrôlables dans le contrat.

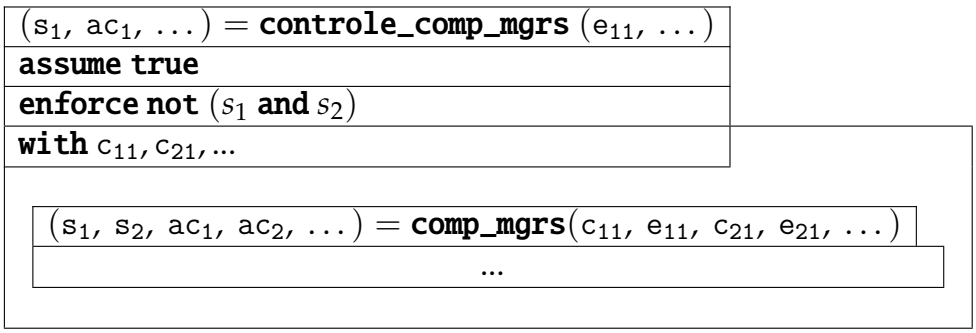

FIGURE 3.4 – Spécification de stratégie de coordination

La figure 3.4 présente un exemple de spécification de contrat pour le modèle de la figure 3.3. Dans cet exemple, l'objectif de contrôle défini dans le contrat est : « **not** (s$_1$ **and** s$_2$) ». Il empêche le gestionnaire mrg$_2$ d'exécuter l'action ac_2 lorsque le gestionnaire mgr$_1$ est dans l'état s_1 à vrai (**true**) et réciproquement. Les variables contrôlables c_{ij} sont déclarées comme points de contrôle sur lesquels agir pour garantir le contrat.

Lorsque le modèle est compilé, la logique de contrôle qui garantit le respect

de l'objectif déclaré est automatiquement construite si cela est possible. Une fois construite, la logique de contrôle est intégrée dans le modèle. La compilation produit un programme exécutable, par exemple en Java. Ce programme correspond au couplage de la logique de contrôle avec le modèle non coordonné. Il constitue un contrôleur qui permet de respecter l'objectif de contrôle à l'exécution réelle.

3.1.3 Modélisation modulaire de la coordination

Le passage à l'échelle des techniques de SCD est limité par la taille du modèle du système à contrôler et par la spécification des objectifs de contrôle. La SCD est basée sur l'exploration de l'espace d'états pour construire le contrôleur. Pour faciliter le passage à l'échelle de notre approche, nous utilisons la synthèse modulaire. Pour un système large, cela permet de casser la complexité de la synthèse de contrôleur. Nous utilisons cette technique pour coordonner les gestionnaires autonomes par petit nombre, et construire un contrôle hiérarchique.

3.1.3.1 Contrôle décentralisé

L'application monolithique de la synthèse de contrôleur sur un modèle de grande taille pourrait ne pas aboutir à un résultat. Comme le montre la figure 3.5, cette approche consiste à centraliser le contrôle en définissant un unique contrat qui contient l'ensemble des objectifs de contrôle à assurer. Ce contrat est associé au modèle global du système qui peut être de grande taille.

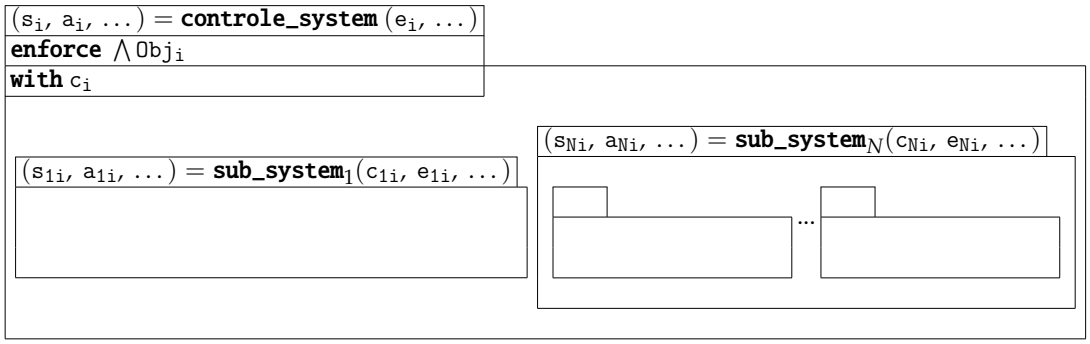

FIGURE 3.5 – Spécification monolithique du contrôle

Un système est généralement constitué de plusieurs sous-ensembles (e.g, plusieurs gestionnaires autonomes), de ce fait il est modélisé par la composition modulaire et hiérarchique des modèles de ses sous-ensembles. L'approche modulaire de la synthèse de contrôleur tire profit de cette structure pour permettre de décentraliser la spécification du contrôle.

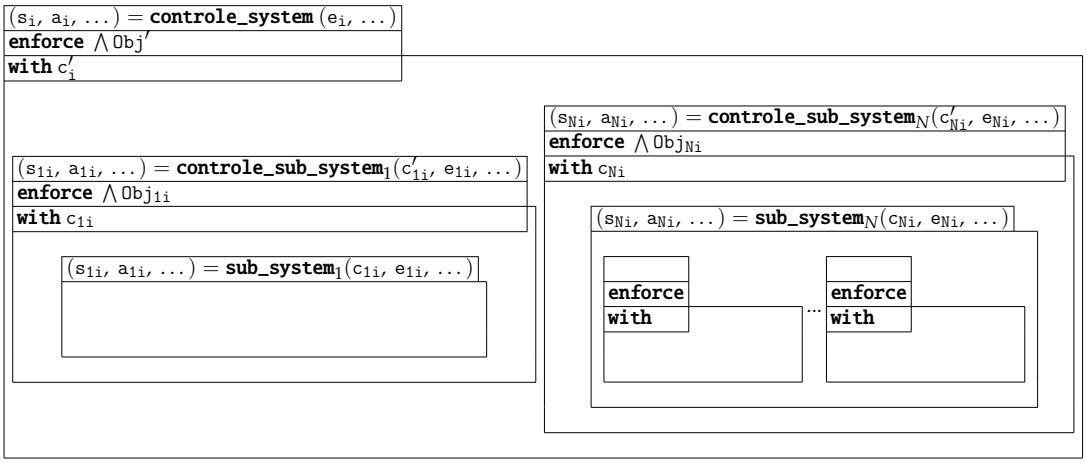

FIGURE 3.6 – Spécification modulaire du contrôle

Au lieu de garantir l'ensemble des objectifs par un unique contrôleur, un ensemble de contrôleurs est construit. Chaque contrôleur assure une partie de l'ensemble des objectifs. Chacun de ces contrôleurs est construit indépendamment des autres et intégré dans le modèle global. Comme le montre la figure 3.6, la spécification du contrôle est faite de manière modulaire et hiérarchique. Chaque objectif de contrôle qui concerne exclusivement un sous-ensemble est associé au modèle du sous-ensemble. Un contrôleur local est construit pour chaque modèle qui a un contrat.

3.1.3.2 Spécification modulaire et hiérarchique

Pour pouvoir appliquer la synthèse modulaire, il est nécessaire que les modèles avec un contrat puissent être réutilisables tels quels. Ces modèles doivent également permettre d'étendre les objectifs de contrôle définis dans leur contrat. De plus, tout cela doit être possible sans changer l'implémentation des contrôleurs obtenus de ces modèles. Pour cela, un modèle avec un contrat doit exposer des entrées qui permettent de recevoir des ordres de contrôle

supplémentaire. En plus des objectifs de contrôle local, le contrat défini dans le modèle doit également contenir des objectifs de contrôle garantissant le respect des ordres extérieurs reçus.

Extension de la contrôlabilité. Dans l'approche monolithique de la synthèse de contrôleur, les entrées contrôlables du modèle du système à contrôler sont déclarées comme variables contrôlables dans le contrat. Elles ne sont pas visibles à l'extérieur du modèle contrôlé du système. Pour réutiliser un modèle contrôlé, il est nécessaire que ces entrées contrôlables soient accessibles pour appliquer un contrôle supplémentaire non défini dans le contrat de ce dernier.

L'extension de la contrôlabilité d'un modèle de coordination de gestionnaires autonomes consiste à exposer des entrées de contrôle supplémentaires c'_i, et à ajouter des objectifs de contrôle Obj'_i associés à ces entrées dans le contrat. Ces entrées permettent ultérieurement à un contrôleur de niveau supérieur de transmettre des ordres de contrôle au contrôleur obtenu. Chaque entrée de contrôle c'_i correspond au contrôle d'une action a_i des gestionnaires coordonnés. Les objectifs de contrôle Obj'_i, autres que ceux définis pour la stratégie de coordination locale, garantissent l'application du contrôle reçu via ces entrées de contrôle. Ces objectifs supplémentaires doivent explicitement figurer dans le contrat. L'objectif de contrôle Obj'_i qui permet de garantir l'inhibition d'une action a_i lorsqu'elle est sollicitée via c'_i est formulé comme suit : $(\neg c'_i \Rightarrow \neg a_i)$. Selon la nature de l'action (courte/instantanée ou longue), l'objectif se traduit différemment. Pour les actions courtes, il est traduit directement par : $Obj'_i = (c'_i \ \texttt{or not} \ a_i)$. Pour une action longue, l'objectif est traduit d'une manière différente : $Obj'_i = \text{LongActions}(c'_i, a_i, s_i)$.

$$\text{LongActions}(c'_i, a_i, s_i) \stackrel{\text{def}}{=} (\text{c'}_i \ \texttt{or not} \ a_i) \ \texttt{and} \left(\left(\texttt{not} \ (\texttt{false fby} \ s_i) \ \texttt{and not} \ a_i \right) \Rightarrow \texttt{not} \ s_i \right)$$

Dans cette expression nous ajoutons le fait que l'exécution d'une action longue ne peut être empêchée que si elle n'est pas déclenchée à l'instant précédent. En effet durant l'exécution d'une action longue, tout ordre d'inhibition de cette action ne peut concerner l'exécution en cours. L'exécution d'une action

longue a_i ne peut être empêchée par c'_i que si elle n'est pas déclenchée à l'instant précédent : (**not** (**false fby** s_i))[1]. De ce fait si l'action a_i n'est pas activée ni à l'instant précédent (**not** (**false fby** s_i)) ni à l'instant courant (**not** a_i) alors l'action ne sera pas exécutée à cet instant (**not** s_i).

Ces deux expressions sont définies de façon générique et peuvent être réutilisées comme des patterns sans avoir à en redéfinir d'autres.

3.2 Mise en oeuvre de la coordination

Pour la mise en oeuvre de la coordination de gestionnaires autonomes, nous utilisons le modèle à composants. Les modèles à composants fournissent un ensemble de fonctionnalités qui permettent la construction de systèmes complexes avec des capacités d'introspection et de reconfiguration dynamique. Notre approche est basée sur le modèle à composants Fractal.

3.2.1 Le modèle à composants Fractal

Le modèle à composants Fractal [14] a été défini par France Telecom R&D et l'INRIA en 2004. L'objectif de ce modèle est de permettre la construction, le déploiement et l'administration (e.g. observation, contrôle, reconfiguration dynamique) de systèmes logiciels complexes. Il est associé à un langage de description d'architecture, Fractal ADL. Ce dernier est basé sur une syntaxe extensible, et permet de construire des assemblages de composants Fractal. Fractal a été implémenté dans différents langages de programmation comme Java, C, C++.

3.2.1.1 Composant Fractal

L'unité de structuration dans le modèle Fractal est le composant. Un composant est une entité d'exécution qui expose un ensemble d'interfaces. Une interface est un point d'accès au composant. Il existe deux catégories d'interfaces : les interfaces serveurs et les interfaces clients. Les interfaces serveurs correspondent aux services fournis par le composant. Elles permettent de faire

1. **fby** est un opérateur Heptagon qui introduit un délai avec une valeur initiale : v **fby** x représente la valeur précédente de x, initialisé avec v au premier instant.

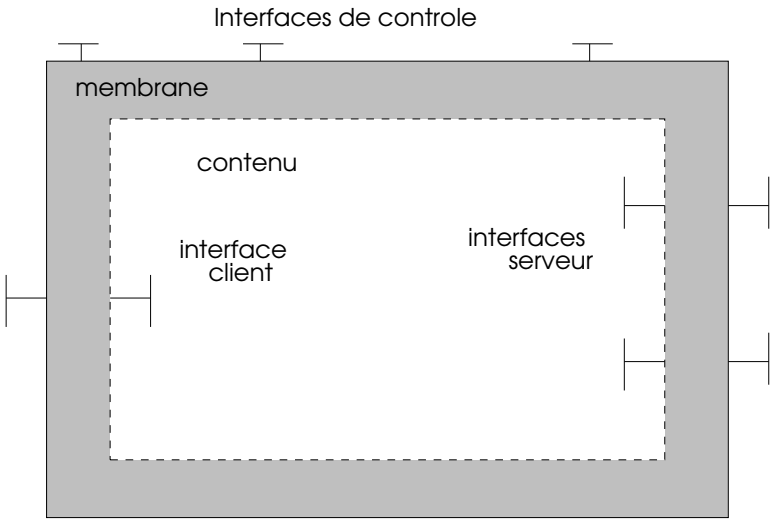

FIGURE 3.7 – Composant Fractal

des appels de méthodes du composant. Les interfaces clients correspondent aux services requis par le composant. Elles permettent au composant d'émettre des appels de méthodes. Comme le montre la figure 3.7, un composant expose également des interfaces non fonctionnelles, contrôleurs, qui permettent à l'exécution, son administration et celle des services qu'il contient (démarrage, arrêt, configuration, etc.). L'ensemble des interfaces fonctionnelles et non fonctionnelles constitue la membrane du composant.

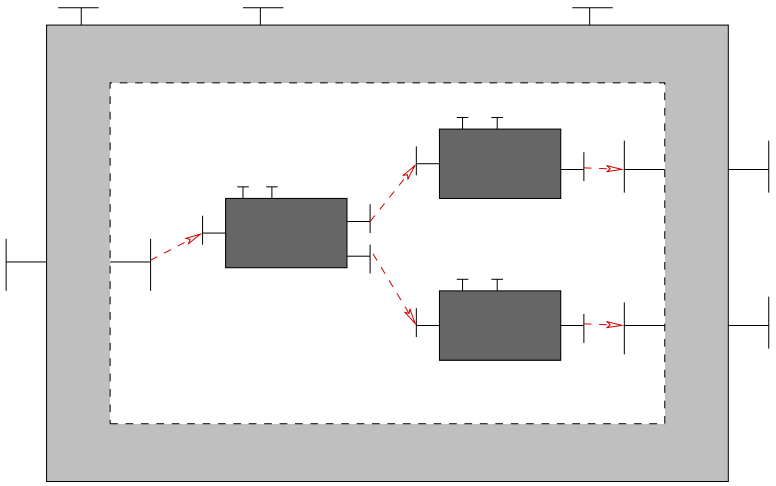

FIGURE 3.8 – Composant composite

Ce modèle distingue généralement deux types de composants : les composants primitifs qui encapsulent généralement les services et les composants composites. Comme le montre la figure 3.8, un composant composite contient d'autres composants primitifs et/ou composites.

3.2.1.2 Introspection et reconfiguration

Les capacités réflexives d'un composant sont assurées par des contrôleurs. Ces contrôleurs implémentent des fonctions permettant l'introspection, la reconfiguration, et l'interception de flux (entrée/sortie). Le modèle à composants Fractal fournit quelques contrôleurs par défaut. Parmi ces contrôleurs, figurent :

- Un **contôleur d'attributs** qui fournit les méthodes permettant l'accès et le contrôle des attributs d'un composant.
- Un **contrôleur de liaisons** qui fournit des méthodes pour contrôler les liaisons (`bind,unbind`) du composant avec d'autres composants, et de consulter, modifier l'état des liaisons.
- Un **contrôleur de cycle de vie** qui fournit des méthodes pour contrôler les principales phases comportementales du composant (ex. démarrage (`start`)/arrêt (`stop`)).
- Un **contrôleur de contenu** qui fournit des méthodes permettant de consulter le contenu d'un composant composite et d'ajouter/retirer des sous-composants.

Le modèle est cependant extensible et ne contraint pas la nature des contrôleurs contenus dans les composants. Il est possible d'adapter le contrôle associé aux composants. Le modèle permet de modifier ou de développer de nouveaux contrôleurs en fonction des besoins.

3.2.1.3 Fractal ADL

Le modèle à composants Fractal fournit un langage de description d'architecture appelé Fractal ADL, basé sur la syntaxe XML. Fractal ADL permet la description de la structure d'application construite à partir de composants

Fractal. Il fournit des constructions de base pour énumérer des composants, des interfaces, des liaisons et laisse aux développeurs la possibilité d'étendre le langage pour intégrer d'autres informations spéficiques à leur cas d'utilisation. Il permet la description d'une architecture sur plusieurs fichiers distincts.

3.2.2 Composant de gestionnaire autonome

Nous utilisons le modèle à composants pour implémenter les aspects non-fonctionnels nécessaires pour la gestion des gestionnaires. Cela permet d'ajouter des fonctions de surveillance et de contrôle pour les gestionnaires autonomes déjà conçus et qui ne fournissent pas explicitement ces fonctions. Cela permet également de séparer l'implémentation d'un gestionnaire (les fonctions d'administration) et l'implémentation du contrôle de ce dernier. Ainsi la spécification et l'implémentation du contrôle du gestionnaire peuvent être modifiées sans impacter l'implémentation des fonctions d'administration. Chaque gestionnaire est encapsulé dans un composant Fractal comme le montre la figure 3.9.

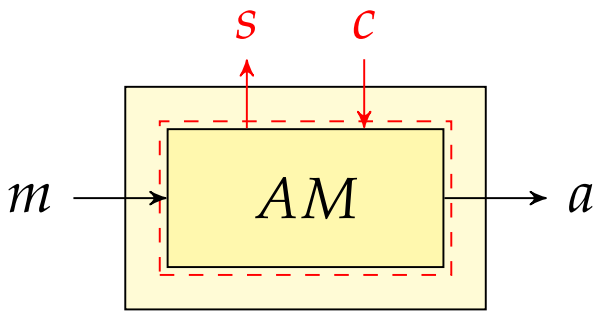

FIGURE 3.9 – Composant de gestionnaire contrôlable

Implémentation de l'interface de coordination. Nous implémentons une interface de contrôle qui exhibe les fonctions de contrôle disponibles sur les gestionnaires lorsque ces fonctions sont explicitement définies. Cette interface de contrôle expose les informations s sur l'état du gestionnaire par rapport au contrôle appliqué sur son comportement. Elle fournit également des informations sur les événements m auxquels le gestionnaire réagit. Elle permet également d'appliquer les opérations de coordination concernant le gestionnaire à travers ses entrées c de contrôle qui permettent d'autoriser ou d'inhiber

les actions a que le gestionnaire peut exécuter. Pour les gestionnaires n'ayant pas de fonctions de contrôle, nous utilisons les contrôleurs par défaut dans Fractal pour implémenter leur contrôle.

3.2.3 Coordination à base de composants

3.2.3.1 Coordination de gestionnaires

Une fois les composants de gestionnaires construits, ces derniers sont assemblés dans un composant composite, comme le montre l'exemple dans la figure 3.10. Le composant composite coordonne leur exécution grâce au contrôleur de coordination obtenu par synthèse. Ce dernier agit sur les interfaces de coordination disponibles sur les composants de gestionnaires.

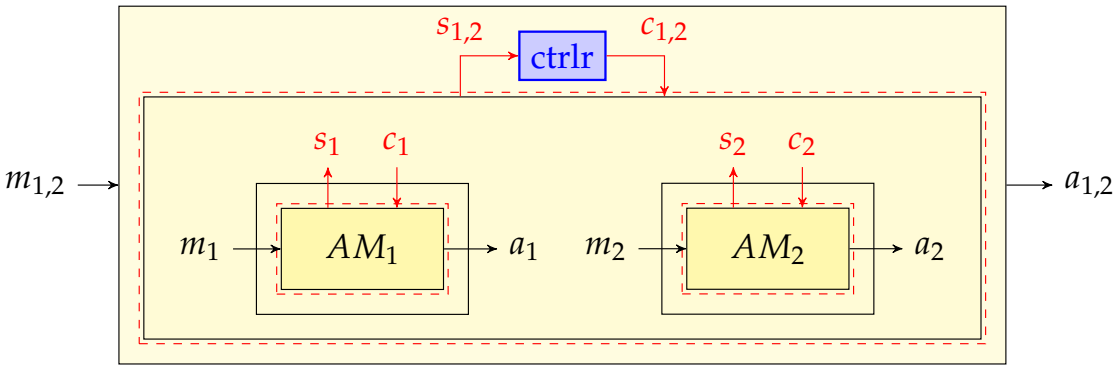

FIGURE 3.10 – Composants de gestionnaires coordonnés

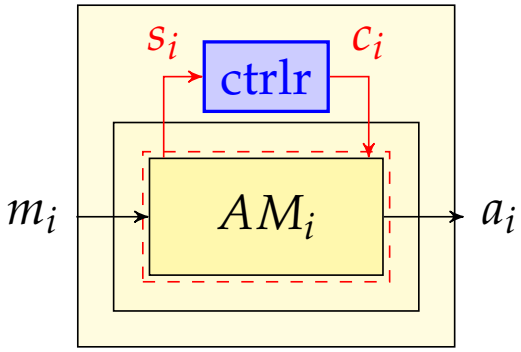

FIGURE 3.11 – Composant composite

De manière générale, comme le montre la figure 3.11, les composants de gestionnaires à coordonner sont encapsulés dans un composant composite. Le contrôleur de coordination, obtenu par programmation synchrone et synthèse de contrôleur discret, est alors intégré dans le composant composite. Il est connecté aux interfaces de coordination des composants de gestionnaires. A l'exécution, il agit sur ces interfaces de coordination pour le respect de la stratégie de coordination.

Coordination. Le contrôleur de coordination présenté à la figure 3.11 correspond au modèle de la coexistence coordonnée des gestionnaires autonomes contenus dans le composant composite. Il est constitué de l'ensemble des modèles du comportement et de la contrôlabilité des gestionnaires couplé avec la logique de contrôle pour la coordination. A l'exécution, le modèle d'un gestionnaire reflète son état courant et facilite l'application dynamique de restrictions imposées par la logique de contrôle sur son comportement. Les sorties du modèle d'un gestionnaire exhibent ses actions d'administration autorisées ou inhibées. Ces sorties doivent être appliquées sur le composant de gestionnaire correspondant afin de garantir la cohérence entre le modèle et l'état du gestionnaire. Si un gestionnaire fournit des fonctions de contrôle explicites, le contrôle exhibé par son modèle est directement appliqué via ces fonctions ; sinon le contrôle par défaut fourni par Fractal est utilisé.

Contrôle par défaut. Pour les gestionnaires qui ne disposent pas de fonctions de contrôle explicites, Fractal définit des contrôleurs par défaut qui permettent la gestion dynamique d'un composant et de ses interactions. Ces contrôleurs fournissent des actions d'administration :

1. **Arrêt et démarrage de composant** : Ces actions sont disponibles avec le contrôleur de cycle de vie. Cette option permet de suspendre entièrement toutes les fonctions d'administration d'un gestionnaire. L'arrêt du composant rend le gestionnaire inaccessible. Il ne reçoit aucun flux en entrée.
2. **Association et dissociation d'interfaces fonctionnelles** : Ces actions sont disponibles avec le contrôleur de liaisons. Cette option permet de suspendre certaines fonctions d'administration d'un gestionnaire. La dissoci-

ation d'une liaison permet de désactiver un lien de communication établi entre le gestionnaire et un autre service (e.g., un capteur). Cependant les autres liens continuent à fonctionner.

3. **Interception des flux d'entrée et de sortie** : Dans l'implémentation Julia, ces actions sont implémentées par des objets Java appelés *Interceptors*. Ils permettent d'intercepter le flux en entrée et/ou en sortie. Ils permettent également d'informer les contrôleurs auxquels ils sont associés avant et/ou après chaque appel de méthodes.

   Dans notre travail, nous utilisons les intercepteurs pour capturer les événements destinés aux gestionnaires et qui sont les entrées du contrôleur de coordination (les entrées de la méthode `step`). Les intercepteurs servent également à filtrer les événements à passer à un gestionnaire. Cela évite de suspendre entièrement (`stop`) ou partiellement (`unbind`) les composants de gestionnaires. Le filtragre des événements est basé sur les sorties du contrôleur de coordination. Ces sorties décrivent l'état dans lequel chaque gestionnaire doit être, et également les actions autorisées à être exécutées.

3.2.3.2 Coordination hiérarchique

Avec la coordination modulaire, les contrôleurs construits garantissent l'application de la stratégie de coordination qu'ils doivent assurer, mais également l'application d'ordres de coordination dont la stratégie est définie ailleurs aux niveaux supérieurs. Cela permet leur réutilisation dans différents contextes plus globaux.

Contrôle d'un composant composite. Comme le montre la figure 3.12, un contrôleur généré par synthèse modulaire fournit des entrées c' pour contrôler les actions des gestionnaires et des sorties s' pour informer de l'état des gestionnaires. Le composant composite associé à ce contrôleur et qui encapsule les composants de gestionnaires peut être réutilisé dans un contexte dans lequel les gestionnaires qu'il encapsule constituent un sous-ensemble des gestionnaires à coordonner. Dans ce cas, l'interface de contrôle fournie par le contrôleur de coordination associé au composite permet l'application d'autres objectifs de contrôle sur ce sous-ensemble. Cela permet la mise en oeuvre de la coordination

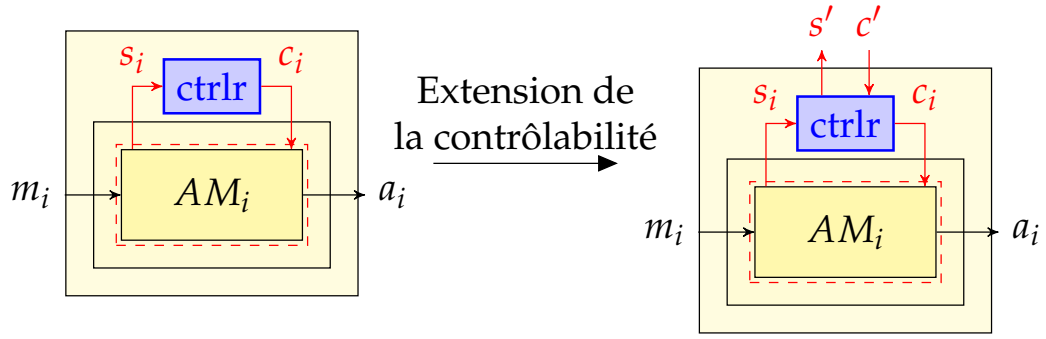

FIGURE 3.12 – Extension de la contrôlabilité

par l'assemblage de composants simples et/ou de composants composites de manière hiérarchique sans aucun changement de leur contenu.

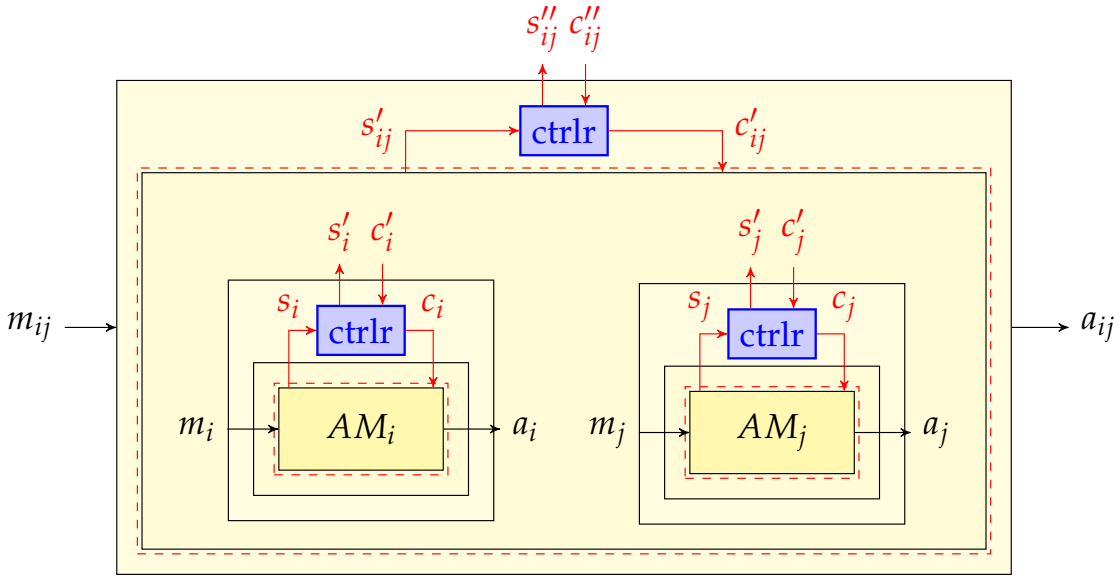

FIGURE 3.13 – Coordination hiérarchique

Contrôle hiérarchique. La figure 3.13 montre un exemple de coordination hiérarchique. Des composants composites qui encapsulent des composants de gestionnaires autonomes coordonnés sont à leur tour encapsulés dans un composant composite. Ce dernier est équipé d'un contrôleur de plus haut niveau qui applique une stratégie de coordination plus globale. Cette stratégie concerne les gestionnaires contenus dans les composants composites internes.

Le contrôleur de haut niveau agit sur les contrôleurs de bas niveau présents dans les composants composites internes pour respecter ses objectifs de contrôle. Les composants composites internes sont réutilisés sans aucune modification.

3.3 Comparaison

Notre approche de coordination est basée sur des aspects qualitatifs. Elle consiste généralement à garantir des propriétés logiques comme l'invariance ou l'exclusion mutuelle, contrairement aux approches basées sur des aspects quantitatifs [20, 38, 40, 51]. Ces dernières garantissent l'optimisation d'indicateurs de performance à travers des fonctions d'utilité/d'optimisation multi-critères. Chaque indicateur de performance a un poids qui indique son importance, et le choix des poids est très important pour la sélection des actions à exécuter. Avec notre approche, l'autorisation ou l'inhibition d'actions dépend de l'état du système et des actions d'administration en cours d'exécution. Nous étudions les relations entre les événements auxquels réagissent les gestionnaires et l'impact des actions de chaque gestionnaire sur les objectifs d'administration. Cette étude permet d'identifier les situations qui peuvent conduire à des décisions conflictuelles ou redondantes. Puis, pour éviter ces décisions, nous définissons les propriétés de coordination, les objectifs de contrôle. Notre approche ressemble aux approches basées sur des règles (priorités/condition-action) [3, 45]. Toutefois avec notre approche, la fonction de contrôle est automatiquement construite et restreint le moins possible le comportement des gestionnaires.

Notre approche repose sur un contrôle externe des gestionnaires, contrairement aux approches basées sur un consensus [2]. En effet, la mise en oeuvre d'un consensus nécessite plusieurs participants. Dans le cas où ces participants sont les gestionnaires, ces derniers vont implémenter les fonctions de décision pour le consensus, en plus des fonctions d'administration. Cela rend difficile la réutilisation des gestionnaires. De plus, la moindre modification du protocole implique une modification de l'implémentation des gestionnaires. Des modifications du protocole peuvent être nécessaires lorsque certains aspects doivent être pris en compte ou bien quand d'autres gestionnaires doivent être intégrés. Notre approche, quant à elle, consiste à identifier les aspects observables et contrôlables des gestionnaires. Ces aspects sont ensuite définis et exposés pour

permettre la réutilisation des gestionnaires dans différents contextes sans aucune modification. L'application des stratégies de coordination repose sur les points de contrôle des gestionnaires. De plus, les fonctions de contrôle d'un gestionnaire sont séparées de l'implémentation de ses fonctions d'administration. Cela permet la réutilisation des gestionnaires existants. Chaque gestionnaire est encapsulé dans un composant qui fournit les fonctions qui permettent son contrôle. De ce fait la spécification du contrôle d'un gestionnaire peut être modifiée sans modifier l'implémentation de ses fonctions d'administration.

3.4 Conclusion

Nous avons vu dans ce chapitre comment concevoir la coordination de gestionnaires autonomes avec les techniques de contrôle discret. Nous décrivons le comportement observable et contrôlable des gestionnaires que nous composons pour modéliser leur coexistence non coordonnée. La synthèse de contrôleur est appliquée sur le modèle de la coexistence pour construire une logique de contrôle qui assure la stratégie de coordination sur le modèle via les points de contrôle définis. Pour la mise en oeuvre de la coordination, nous utilisons le modèle à composants Fractal. Chaque gestionnaire est encapsulé dans un composant qui fournit les fonctions de contrôle. Les composants de gestionnaires sont assemblés dans un composant composite qui assure leur coordination grâce au contrôleur de coordination généré par synthèse de contrôleur discret.

4

Gestion de la performance et de l'optimisation de ressources d'un système dupliqué

Contents

4.1 Gestionnaires autonomes non coordonnés 56

4.1.1 Gestionnaire d'auto-dimensionnement : Self-sizing . . 56

4.1.2 Gestionnaire d'auto-régulation de fréquence CPU : Dvfs 58

4.2 Problèmes d'optimisation de ressources 59

4.3 Conception du contrôleur de coordination 60

4.3.1 Modélisation du contrôle des gestionnaires 60

4.3.1.1 Modélisation du contrôle de self-sizing 61

4.3.1.2 Modélisation de l'état global des Dvfs 63

4.3.2 Spécification de la coordination 64

4.3.2.1 Stratégie de coordination 64

4.3.2.2 Spécification du contrat 64

4.3.2.3 Programme final 64

4.4 Expérimentations 65

4.4.1 Configuration 66

4.4.2 Calibrage des seuils des gestionnaires 66

4.4.2.1 Seuil maximal pour self-sizing et Dvfs 66

4.4.2.2 Seuil minimal pour self-Sizing et Dvfs 67

4.4.3 Évaluation 70

4.4.3.1 Comportement non coordonné 70
4.4.3.2 Comportement coordonné 72
4.5 Conclusion 75

Dans ce chapitre, nous nous intéressons à l'administration d'un système basé sur la réplication. Il s'agit d'un système distribué constitué de serveurs dupliqués hébergés par des machines distinctes. Les requêtes entrantes sont réparties entre les serveurs par un équilibreur de charge. Nous considérons la coordination de gestionnaires dédiés à l'optimisation de ressources : self-sizing et Dvfs. Le gestionnaire self-sizing est utilisé pour minimiser le nombre de serveurs actifs et un Dvfs est installé sur chaque machine qui exécute un serveur pour ajuster la fréquence à laquelle s'exécutent ses processeurs. Nous appliquons notre approche pour coordonner les gestionnaires afin d'optimiser de manière efficace les ressources utilisées par le système.

Nous présentons dans ce chapitre un exemple d'application simple de notre approche. Toutefois cet exemple permet de démontrer sa faisabilité.

4.1 Gestionnaires autonomes non coordonnés

Les gestionnaires ci-dessous assurent la gestion de la performance et l'optimisation des ressources de calcul. Ils sont conçus indépendamment.

4.1.1 Gestionnaire d'auto-dimensionnement : Self-sizing

Ce gestionnaire autonome est dédié au dimensionnement dynamique d'un système en fonction de la charge de travail de ce dernier. Il peut être appliqué sur des systèmes dont la structure est basée sur le canevas d'équilibrage de charge. Dans ce canevas, le modèle de communication est synchrone (Client/Serveur), les serveurs sont clonés statiquement lors du démarrage du système et un aiguilleur est placé en frontal des serveurs. Le rôle de cet aiguilleur est de répartir la charge entre tous les serveurs. Une requête peut donc être traitée indifféremment par n'importe lequel des serveurs. Lorsqu'un serveur reçoit une requête, il l'exécute, il met en cohérence son état avec les autres serveurs si besoin, puis il retourne le résultat de la requête au client. On considère que ce

canevas s'exécute sur une grappe de machines. L'aiguilleur, ainsi que chaque serveur, s'exécutent sur une machine différente.

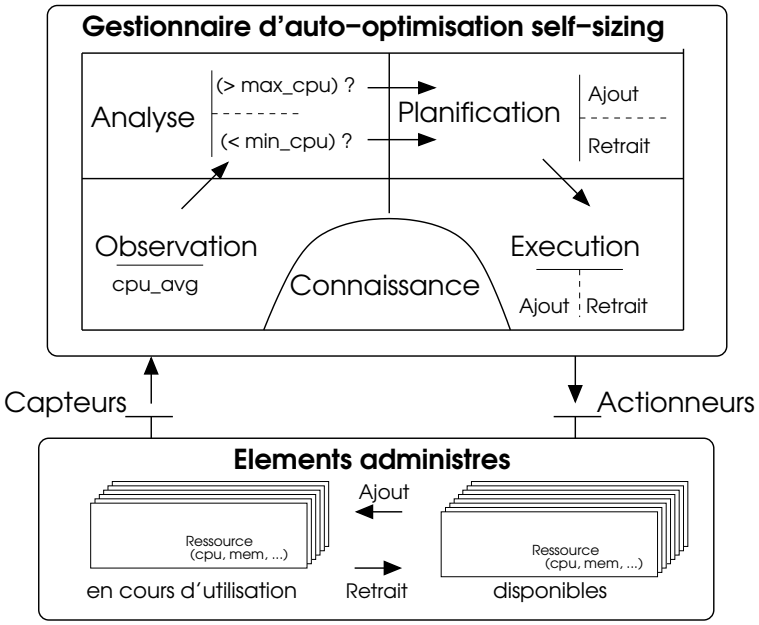

FIGURE 4.1 – Gestionnaire d'auto-dimensionnement : self-sizing

La figure 4.1 décrit le fonctionnement du gestionnaire self-sizing. Ce dernier permet de dimensionner dynamiquement le degré de duplication des serveurs qui constituent le système administré. Le dimensionnement est effectué en fonction de la charge de travail soumise au système. En cas de surcharge, le système est approvisionné en ressource – ajout de serveur – alors qu'en cas de sous-charge, les ressources du système sont optimisées – retrait de serveur. Pour réaliser ces opérations, le gestionnaire utilise la connaissance qu'il a de la structure du système. Il connaît les machines en cours d'utilisation sur lesquelles sont exécutés les serveurs, ainsi que les machines disponibles.

L'état du système est surveillé via des sondes (capteurs sur la figure 4.1). Celles-ci récupèrent périodiquement la charge CPU de chaque machine qui exécute un serveur dupliqué. Le gestionnaire calcule une moyenne glissante EWMA (exponentiellement pondérée) des charges CPU. Cette moyenne est utilisée pour évaluer le niveau d'utilisation des machines. Le niveau acceptable, le niveau pour lequel le gestionnaire estime le redimensionnement du système non nécessaire, est borné par un seuil minimal et un seuil maximal. Le gestionnaire considérera que les machines sont saturées lorsque la moyenne est

supérieure au seuil maximal. A l'opposé, il considérera que les machines sont sous utilisées lorsque la moyenne est inférieure au seuil minimal.

Lorsqu'une surcharge du système est détectée, le gestionnaire démarre un nouveau serveur dupliqué sur une machine disponible, et met à jour l'état de ce dernier en fonction des autres serveurs. Puis il intègre ce nouveau serveur dans la liste des serveurs dupliqués au niveau de l'aiguilleur de charge. Dans le cas d'une sous-charge, il sélectionne un serveur à arrêter, le déconnecte de l'aiguilleur de charge, l'arrête, et le désinstalle de la machine. Puis il remet la machine dans la liste des machines disponibles. Les systèmes administrés fournissent les actionneurs permettant d'appliquer les actions d'administration.

4.1.2 Gestionnaire d'auto-régulation de fréquence CPU : Dvfs

La plupart des micro-processeurs récents offrent la possibilité d'ajuster leur fréquence d'exécution. Cette fonctionnalité permet de réduire leur puissance d'exécution lorsque la charge de travail est faible pour optimiser la consommation énergétique. L'utilisation de cette fonctionnalité dans l'administration d'un système impliquant un nombre important de machines peut contribuer à réduire l'énergie consommée sans altérer la performance du système.

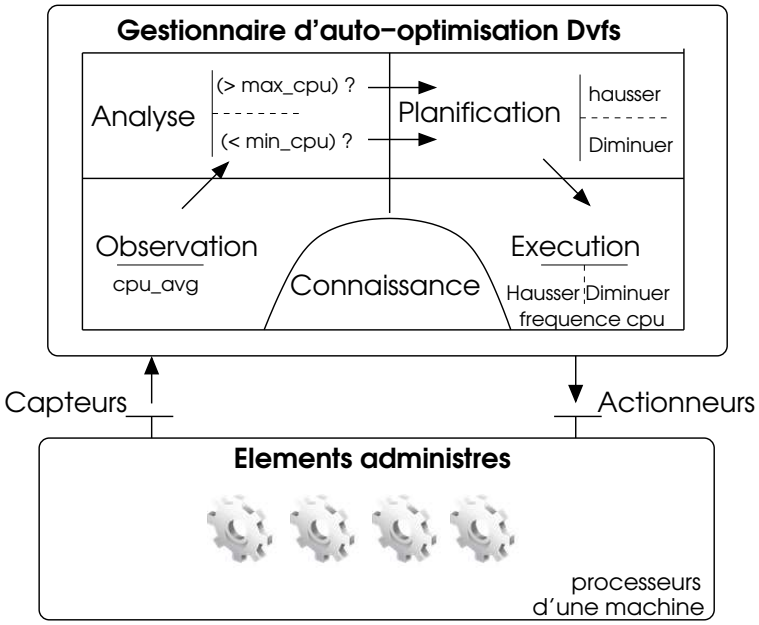

FIGURE 4.2 – Gestionnaire d'auto-régulation : Dvfs

Le gestionnaire Dvfs, représenté dans la figure 4.2, exploite de cette fonctionnalité. Il est dédié à l'ajustement de la fréquence des processeurs d'une machine. Il adapte dynamiquement la fréquence des processeurs en fonction de la charge de travail à traiter. Il diminue la fréquence CPU d'une machine lorsque celle-ci est sous-utilisée ; il augmente la fréquence lorsqu'elle est surchargée. Le gestionnaire connaît les différents niveaux de fréquences des processeurs de la machine administrée. Il connaît également l'état courant de la machine.

Une sonde récupère périodiquement la charge CPU de la machine. Une moyenne exponentiellement pondérée (EWMA) de ces valeurs est calculée et utilisée pour évaluer le niveau d'utilisation des processeurs de la machine administrée. Les niveaux de charge considérés acceptables sont délimités par un seuil minimal et un seuil maximal. Lorsque la moyenne est dans cet intervalle, aucune action n'est exécutée. Mais, lorsqu'elle est au-dessus du seuil maximal, les processeurs sont considérés surchargés et le gestionnaire réagit en augmentant leur fréquence s'ils ne sont pas en fréquence maximale. Lorsque la moyenne est en-dessous du seuil minimal, les processeurs sont considérés sous-chargés et le gestionnaire réagit en diminuant leur fréquence s'ils ne sont pas en fréquence minimale. L'exécution de ces opérations est effectuée par des actionneurs fournis par la machine administrée.

4.2 Problèmes d'optimisation de ressources

L'utilisation des gestionnaires self-sizing et Dvfs peut permettre une meilleure optimisation de l'utilisation des ressources allouées à un système basé sur la réplication de service. Le gestionnaire self-sizing peut être utilisé pour minimiser le nombre de serveurs actifs. Un Dvfs peut être utilisé sur chaque machine active pour minimiser la fréquence d'exécution de son (ses) processeur(s). Cependant les deux types de gestionnaires se basent sur la charge CPU et leurs actions affectent la charge CPU. De ce fait leur exécution non coordonnée pour l'administration du même système peut avoir des effets indésirables.

En fréquence maximale un processeur peut effectuer beaucoup plus de calculs par unité de temps qu'en fréquence inférieure. Ainsi une charge de calcul qui sature un processeur en une fréquence quelconque pourrait ne pas saturer le processeur lorsque ce dernier est à une fréquence supérieure. Les

machines équipées d'un Dvfs n'ont pas toujours leur(s) processeur(s) à la puissance maximale. Lorsque self-sizing détecte une surcharge (évaluée via la charge CPU) des machines qui exécutent les serveurs actifs, il n'a aucune connaissance de la fréquence d'exécution des processeurs de ces machines. Si les processeurs des machines ne sont pas en fréquence maximale, la hausse de la fréquence CPU des machines par les Dvfs pourrait permettre aux machines de supporter la charge. Dans ce cas, un ajout de serveur devient inutile. De plus lorsque les machines sont à une fréquence CPU autre que leur fréquence maximale, l'occurrence d'une surcharge détectée simultanément par self-sizing et les Dvfs entraîne l'ajout d'un nouveau serveur mais également la hausse de la fréquence CPU des machines. Ces opérations simultanées peuvent entraîner la baisse de la charge CPU des machines jusqu'en dessous du seuil minimal acceptable ce qui entraîne l'exécution d'opérations de retrait et/ou de baisse de fréquence. Une situation similaire peut également arriver à l'occurrence d'une sous-charge. L'exécution non coordonnée des gestionnaires peut entraîner une oscillation de la charge CPU des machines entre les seuils maximal et minimal entraînant des réactions répétitives. Cela peut conduire à une instabilité du système administré.

4.3 Conception du contrôleur de coordination

Cette section présente la conception du contrôleur de coordination du gestionnaire self-sizing et des Dvfs. Nous présentons les modèles des gestionnaires. Dans cet exemple, seul le gestionnaire self-sizing est contrôlable. Pour les Dvfs nous ne modélisons que leurs états d'exécution globaux, nécessaires pour autoriser ou inhiber les actions d'ajout de nouveau serveur.

4.3.1 Modélisation du contrôle des gestionnaires

Les modèles qui décrivent les gestionnaires sont constitués d'un ou de plusieurs automates. Le modèle du gestionnaire self-sizing est constitué d'automates qui décrivent son comportement et le contrôle des actions d'administration qu'il peut exécuter. Pour les gestionnaires Dvfs, nous ne modélisons que leurs états d'exécution globaux. Un seul automate est utilisé pour représenter

les états globaux des gestionnaires Dvfs actifs.

4.3.1.1 Modélisation du contrôle de self-sizing

Le modèle du comportement contrôlable du gestionnaire self-sizing, représenté dans la figure 4.3, est constitué de trois automates. L'automate au centre représente le comportement de self-sizing et les deux autres modélisent le contrôle des actions d'ajout et de retrait de serveurs.

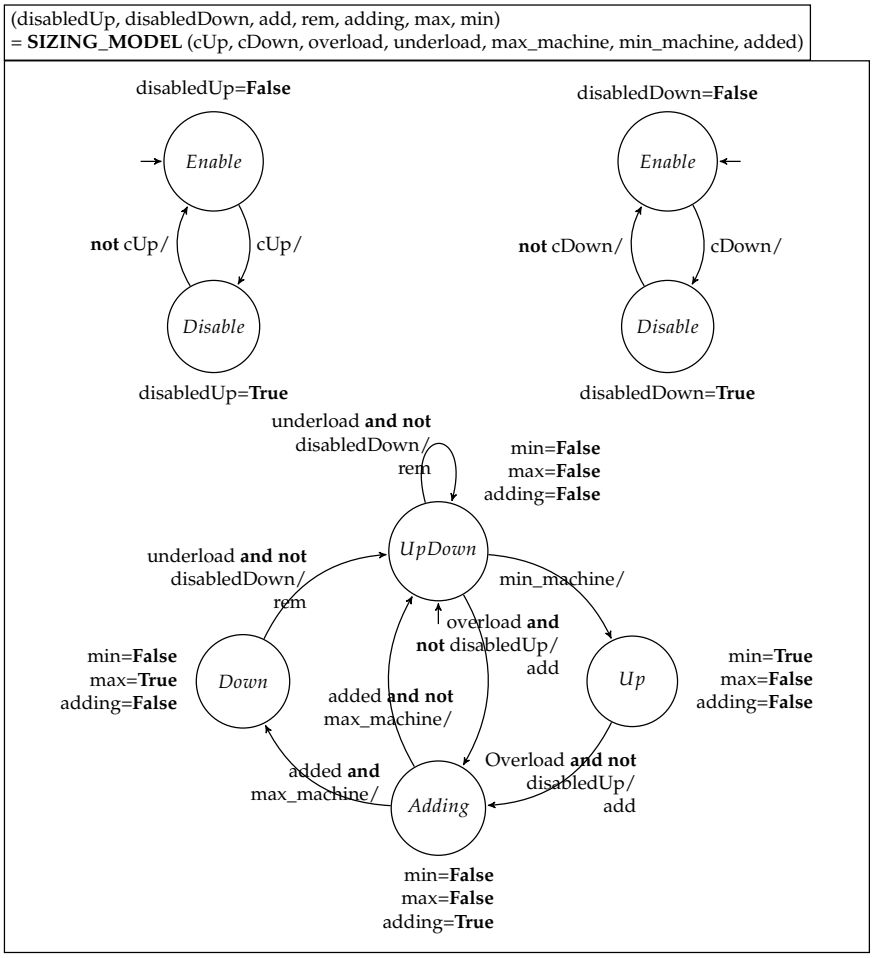

FIGURE 4.3 – Modèle de contrôle de self-sizing

L'automate à droite représente le contrôle des actions de retrait de serveurs. Il est constitué de deux états : `Enable` et `Disable`. L'état `Enable`, état initial, indique que les actions de retrait sont autorisées et l'état `Disable` indique que les actions de retrait sont inhibées. Le passage de l'état `Enable` à l'état

Disable et réciproquement est contrôlé via l'entrée cDown. Lorsqu'elle est à **true** l'automate se met dans l'état Disable et lorsqu'elle est à **false** dans l'état Enable. L'état courant du contrôle des actions de retrait de serveur est indiqué par la sortie disabledDown qui est une variable d'état. Elle est à **true** lorsque les actions de retrait sont inhibées. L'automate à gauche représente le contrôle des actions d'ajout de serveurs. Cet automate est semblable à celui du contrôle des retraits. Le passage de l'état Enable à l'état Disable et réciproquement est contrôlé via l'entrée cUp. L'état courant du contrôle des actions d'ajout est indiqué par la sortie disabledUp.

L'automate au centre représente le comportement du gestionnaire self-sizing. Il est constitué de quatre états. Le gestionnaire est initialement dans l'état UpDown dans lequel il peut exécuter aussi bien des opérations d'ajout que des opérations de retrait. Il est dans l'état Down quand le nombre maximum de serveurs actifs autorisé est atteint. Dans cet état il ne peut exécuter que des opérations de retrait. Le gestionnaire est dans l'état Up lorsque le nombre minimum de serveurs actifs autorisé est atteint. Dans cet état il ne peut exécuter que des opérations d'ajout. L'exécution d'une opération d'ajout est représentée par l'état Adding. Contrairement aux ajouts, l'exécution des opérations de retrait est considérée instantanée dans le modèle. L'occurrence d'une sous-charge (underload à **true**), lorsque le gestionnaire est dans l'état UpDown ou dans l'état Down, entraîne l'exécution d'une opération de retrait (rem à **true**) si les opérations de retrait sont autorisées (disabledDown à **false**). Dans l'état UpDown, le gestionnaire passe dans l'état Up lorsque le nombre minimum de serveur est atteint (min_machine à **true**). L'occurrence d'une surcharge (overload à **true**), lorsque le gestionnaire est dans l'état UpDown ou dans l'état Up, entraîne l'exécution d'une opération d'ajout (add à **true**) si les opérations d'ajout sont autorisées (disabledUp à **false**). A l'exécution d'un ajout le gestionnaire passe dans l'état Adding où aucune autre opération d'ajout ou de retrait de serveur ne peut être entamée. A la fin de l'opération d'ajout, il passe dans l'état Down si le nombre maximum de serveurs est atteint (max_machine à **true**) sinon il retourne dans l'état UpDown.

4.3.1.2 Modélisation de l'état global des Dvfs

Dans cette solution de coordination, aucun contrôle n'est effectué sur l'exécution des Dvfs locaux. Seul leur état global courant est important pour pouvoir autoriser ou empêcher les opérations d'ajout. Pour cela nous utilisons une sonde pour collecter l'état courant de l'ensemble des Dvfs locaux. Chaque Dvfs fournit deux sorties booléennes **min** étant à vrai lorsque la fréquence minimale est atteinte et **max** étant à vrai lorsque la fréquence maximale est atteinte. La sonde renvoie deux valeurs, l'une étant la conjonction de toutes les valeurs des sorties **min** des Dvfs et l'autre la conjonction de toutes les valeurs des sorties **max**.

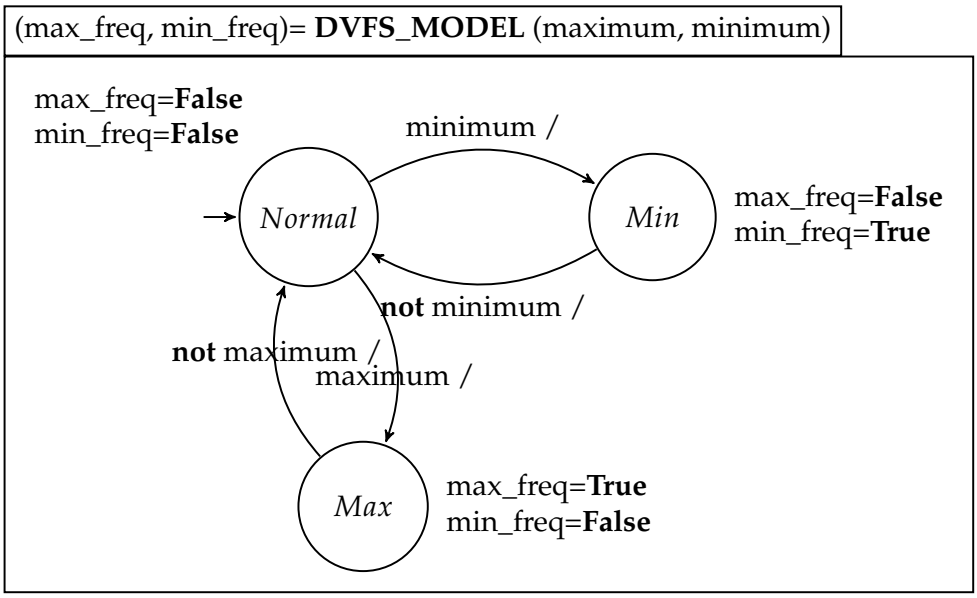

FIGURE 4.4 – Modèle global du mode d'exécution des Dvfs

La figure 4.4 présente l'automate qui modélise l'état global de l'ensemble des gestionnaires Dvfs qui s'exécutent sur les machines qui hébergent les serveurs actifs. L'automate est constitué de trois états : `Normal`, `Min` et `Max`. Initialement dans l'état `Normal`, l'automate va dans l'état `Max` quand tous les Dvfs ont atteint la fréquence maximale. Depuis l'état `Normal`, il va dans l'état `Min` lorsque tous les Dvfs ont atteint la fréquence minimum. Il retourne dans l'état `Normal` lorsque, au moins un des Dvfs n'a atteint ni la fréquence maximale ni la fréquence minimale. Cet automate a deux sorties, `max_freq` qui est à **`true`** dans l'état `Max` et `min_freq` qui est à **`true`** dans l'état `Min`.

4.3.2 Spécification de la coordination

La figure 4.5 décrit la coexistence des gestionnaires.

```
(max_freq, min_freq, disabledUp, disabledDown, add, rem, adding, max, min)
 = COEXISTENCE (minimum, minimum,
                cUp, cDown, overload, underload, max_machine, min_machine, added)
(max_freq, min_freq) = DVFS_MODEL (minimum, minimum);
(disabledUp, disabledDown, add, rem, adding, max, min) =
    SIZING_MODEL (cUp, cDown, overload, underload, max_machine, min_machine, added);
```

FIGURE 4.5 – Composition des modèles des gestionnaires self-sizing et Dvfs

4.3.2.1 Stratégie de coordination

Une stratégie pour garantir une optimisation efficace des ressources et éviter des actions inutiles consiste à empêcher l'ajout d'un nouveau serveur tant qu'il est possible d'augmenter la fréquence d'exécution des processeurs des machines qui hébergent les serveurs actifs. En effet une surcharge détectée par le gestionnaire self-sizing peut être considérée importante à traiter que si les processeurs des machines actives sont à leur fréquence maximale. Dans ce cas, il est nécessaire d'ajouter un nouveau serveur puisque les machines actives ont atteint leur capacité maximale.

– **Ignorer la surcharge détectée des serveurs dupliqués** — Si les processeurs des machines actives ne sont pas à leur fréquence maximale.

4.3.2.2 Spécification du contrat

Pour réaliser la stratégie de coordination, nous définissons formellement un objectif de contrôle. Cet objectif de contrôle est exprimé via la sorties des automates et déclaré sous forme de contrat :
1. `((not max_freq and disabledUp) or (max_freq and not disabledUp))`.

4.3.2.3 Programme final

Le programme final, représenté à la figure 4.6, associe le contrat défini au modèle de la coexistence des gestionnaires **COEXISTENCE**.

```
(max_freq, min_freq, disabledUp, disabledDown, add, rem, adding, max, min)
  = COORD (minimum, minimum,
           overload, underload, max_machine, min_machine, added)
enforce ((not max_freq and disabledUp) or (max_freq and not disabledUp))
with cUp, cDown
 (max_freq, min_freq, disabledUp, disabledDown, add, rem, adding, max, min)
  = COEXISTENCE (minimum, minimum,
                 cUp, cDown, overload, underload, max_machine, min_machine, added)
```

FIGURE 4.6 – Coordination de gestionnaires self-sizing et Dvfs

Les variables cUp et cDown du modèle du gestionnaire self-sizing sont déclarées variables contrôlables sur lesquelles agirent pour le respect de la stratégie de coordination. Dans cet exemple aucun contrôle n'est défini pour les actions de retrait. La variable cDown aura toujours la valeur **true** puisque l'objectif ne concerne pas les retraits. Les retraits seront toujours autorisées dans ce modèle coordonné.

4.4 Expérimentations

L'objectif de ces expérimentations est d'évaluer le comportement du contrôleur obtenu. Il s'agit de montrer que le contrôleur généré assure la politique de coordination définie, bien que le système considéré soit petit et qu'il soit possible d'implémenter manuellement l'objectif de contrôle. Nous avons utilisé trois charges différents pour chaque exécution : **Workload1** *(4750 requêtes/sec)*, **Workload2** *(5000 requêtes/sec)* et **Workload3** *(5542 requêtes/sec)*. Chaque charge est définie en deux phases, une première phase qui consiste en une charge croissante (pendant environ 3 minutes), puis une seconde phase durant laquelle la charge est constante. Pour chaque charge, nous avons effectué une exécution non coordonnée et une autre exécution durant laquelle les gestionnaires sont coordonnés. A chaque exécution, chaque machine qui héberge un serveur dupliqué actif débute avec la fréquence minimale. Les charges **Workload1** et **Workload2** peuvent être traitées par un serveur à la fréquence maximale alors que la charge **Workload3** nécessite deux serveurs dupliqués.

4.4.1 Configuration

La plate-forme expérimentale est constituée de trois machines ayant les mêmes caractéristiques (processeurs et capacité en mémoire). Les machines (`node0`, `node1` et `node2`) sont connectées en réseau. Les machines `node1` et `node2` ont deux niveaux de fréquence de processeurs : 800Mhz étant la fréquence minimale et 1.20Ghz étant la fréquence maximale. Le système administré est constitué d'un serveur `Apache` [1] et de deux serveurs `Tomcat` [2]. Le serveur `Apache` représente le point d'entrée du système. Il reçoit toutes les requêtes à traiter et les répartit entre les serveurs `Tomcat` actifs. Le serveur `Apache` est utilisé comme équilibreur de charge. Il est exécuté sur la machine `node0`. Les machines `node1` et `node2` hébergent les serveurs `Tomcat`. Nous utilisons `Jmeter` pour simuler les clients qui émettent les requêtes `HTTP` sur le système administré.

4.4.2 Calibrage des seuils des gestionnaires

Des expérimentations ont été réalisées pour déterminer les seuils maximal et minimal des gestionnaires self-sizing et Dvfs. Ces expérimentations ont été faites de manière empirique. Le seuil maximal, appelé T^{max}, est fixé manuellement et ne change pas pour les types de gestionnaires. Le seuil minimal, appelé T^{min}, est calculé dynamiquement.

4.4.2.1 Seuil maximal pour self-sizing et Dvfs

Une machine qui utilise son processeur à 100% passe tout son temps à exécuter des opérations. Cela indique que la machine a atteint sa charge maximale. Lorsque la charge reçue est supérieure à la charge maximale, la machine sature et sa performance se dégrade. Il est donc préférable d'envisager un seuil maximal inférieur à 100%.

Nous avons choisi de manière arbitraire 90% comme valeur pour T^{max}. Nous avons observé qu'une machine utilisant 90% de son processeur commence à saturer, mais traite les requêtes avec un délai acceptable. Cela permet de récupérer la charge CPU de la machine dans un délai suffisamment court et de réagir pour éviter une dégradation trop importante de la performance en

1. `http://httpd.apache.org/`
2. `http://tomcat.apache.org/`

haussant la fréquence CPU ou en ajoutant un nouveau serveur hébergé par une autre machine.

4.4.2.2 Seuil minimal pour self-Sizing et Dvfs

Nous avons utilisé différentes charges de travail suivant le même profil (une phase de montée en puissance suivie d'une phase constante), pour observer l'impact des opérations d'administration des gestionnaires sur la charge CPU. Ce qui diffère entre les charges de travail est l'intensité, c'est-à-dire le nombre de requêtes injectées. Pour évaluer le facteur de variation de la charge CPU, les opérations d'administration sont exécutées manuellement une fois que la charge est constante et stable. L'objectif est de déterminer si le facteur de variation de la charge CPU est le même pour chaque charge de travail. Cela permet de déduire une équation pour le calcul du seuil minimal en fonction du seuil maximal. Pour le gestionnaire self-sizing, la formule devrait également prendre en compte le nombre de serveurs dupliqués actifs.

4.4.2.2.1 Seuil minimal (T^{min}) pour self-sizing. Nous avons réalisé des expérimentations pour observer l'impact des opérations du gestionnaire self-sizing sur la charge moyenne des machines qui exécutent les serveurs dupliqués. Les opérations d'ajout et de retrait de serveurs sont exécutées une fois que la charge en entrée est constante et stable.

La figure 4.7 présente des expérimentations dans lesquelles on ajoute un serveur dupliqué. Initialement un seul serveur est actif. L'ajout d'un second serveur fait baisser la charge CPU moyenne des machines. Nous notons toutefois, que cette diminution n'est pas de moitié par rapport à la charge observée avant l'ajout. Ce résultat est retrouvé pour toutes les charges testées («*charge-cpu-{1,2,3,4}*»). Puisque les requêtes sont distribuées équitablement, on espérait observer une baisse de moitié, mais la charge moyenne obtenue est toujours supérieure à la charge théorique attendue.

La figure 4.8 présente des expérimentations dans lesquelles on retire un serveur dupliqué. Chaque exécution débute avec deux serveurs dupliqués. Lorsqu'un serveur est arrêté, la charge sur le serveur restant augmente mais ne double pas.

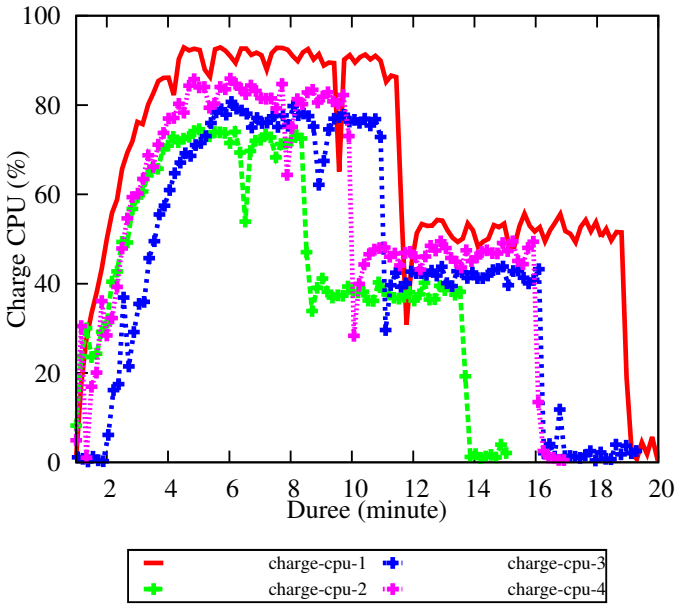

FIGURE 4.7 – Seuil minimal pour self-sizing : ajout de serveur

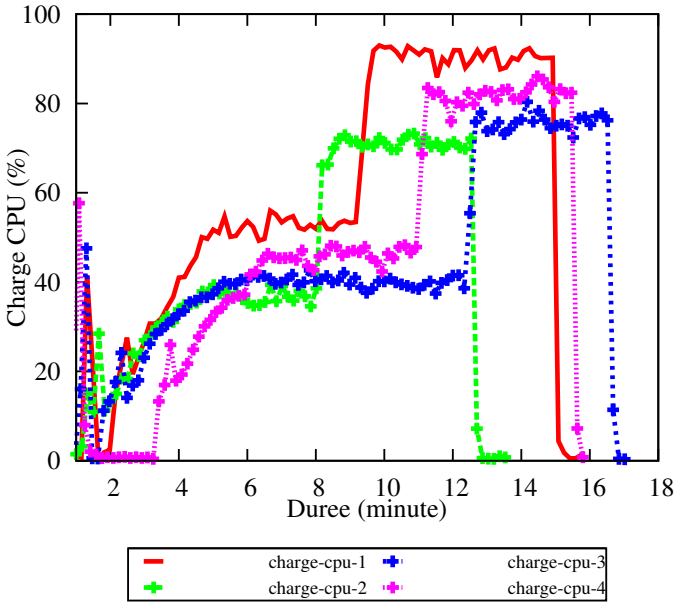

FIGURE 4.8 – Seuil minimal pour self-sizing : retrait de serveur

Cela signifie que pour un système dont le degré de réplication ne dépasse pas deux, le calcul du seuil minimal peut se faire selon la formule suivante :

$$T^{min} = T^{max}/2$$

Pour un système avec un degré de réplication supérieur à deux, on souhaitera retirer un serveur le plus tôt possible. Dans ce cas, le seuil minimal peut être calculé comme suit :

$$\mathtt{T}^{\mathtt{min}} + \frac{\mathtt{T}^{\mathtt{min}}}{(\mathtt{n}-1)} < \mathtt{T}^{\mathtt{max}}$$

Où n est le nombre de serveurs actifs. En d'autres termes :

$$\mathtt{T}^{\mathtt{min}} < \mathtt{T}^{\mathtt{max}} * \frac{(\mathtt{n}-1)}{\mathtt{n}} \quad \longrightarrow \quad \mathtt{T}^{\mathtt{min}} = [\mathtt{T}^{\mathtt{max}} * \frac{(\mathtt{n}-1)}{\mathtt{n}}] - C$$

Où C est une marge qui permet de maintenir T^{min}, à la fois suffisamment haut pour retirer le plus tôt possible un serveur, et à la fois suffisamment bas par rapport à T^{max} pour éviter des oscillations de la charge entre T^{max} et T^{min}. Pour éviter que T^{min} soit trop proche de T^{max}, une valeur maximale peut être fixée pour T^{min}. Dans ce cas, la valeur maximale de T^{min} est utilisée chaque fois que la valeur calculée de T^{min} est supérieure.

4.4.2.2.2 Seuil minimal (T^{min}) pour Dvfs. Nous avons utilisé une machine munie d'un processeur Dual-core ayant une fréquence minimale de 800Mhz et une fréquence maximale de 1.2Ghz. Une charge qui sature la machine en fréquence minimale pourrait être supportée en fréquence maximale. Théoriquement la machine est supposée pouvoir supporter 1.5 fois plus de charge en fréquence maximale qu'en fréquence minimale. Nous avons effectué des expérimentations en utilisant le même profil de charge mais avec différentes intensités. Durant ces expérimentations, la fréquence CPU de la machine est modifiée pour observer l'impact de ces changements sur la charge CPU de la machine.

La figure 4.9 montre les variations de la charge CPU entre la fréquence maximale et la fréquence minimale. En fréquence minimale, la hausse de la fréquence fait baisser l'utilisation du CPU. Cependant pour la même charge en entrée, le rapport entre l'utilisation du CPU en fréquence maximale et l'utilisation du CPU en fréquence minimale est toujours inférieur au rapport entre les deux fréquences CPU. Le rapport semble constant et est inférieur à 1,5 sur notre plate-forme. La baisse observée est toujours inférieure à la baisse «théorique»

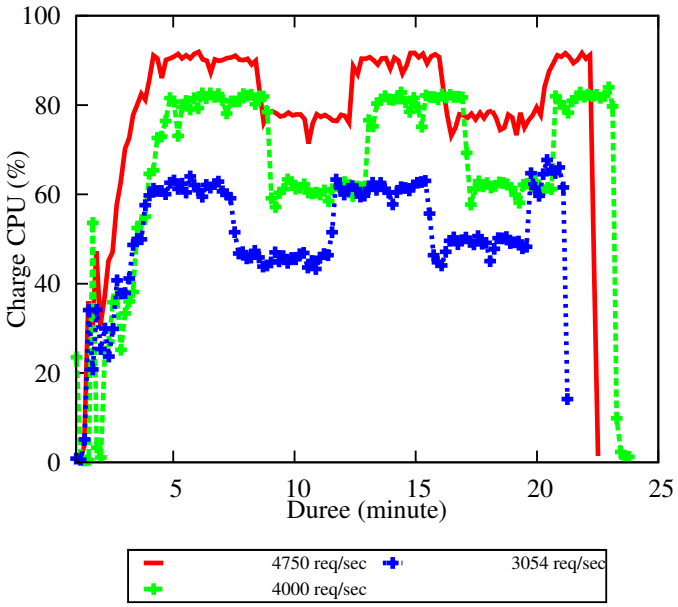

FIGURE 4.9 – Seuil minimal pour Dvfs

attendue. Cela permet de définir le seuil Minimal en fonction du seuil maximal et le rapport entre deux valeurs consécutives de fréquences CPU. Le seuil minimal peut être calculé comme suit :

$$T^{min} = T^{max} * \frac{\texttt{next lower frequency}}{\texttt{current frequency}}$$

4.4.3 Évaluation

Cette section présente l'évaluation du comportement du contrôleur généré pour la coordination des Dvfs et du self-sizing. Initialement un seul serveur `Tomcat` est actif. Le deuxième serveur `Tomcat`, en fonction de la charge, sera ajouté ou enlevé par le gestionnaire self-sizing. Les exécutions durent 20 minutes. L'injection de charge est arrêtée après les 20 minutes.

4.4.3.1 Comportement non coordonné

Lors des exécutions non coordonnées, la détection d'une surcharge conduit à la hausse de la fréquence CPU de la machine qui héberge le premier serveur `Tomcat`. La surcharge a également à l'ajout du second serveur `Tomcat`.

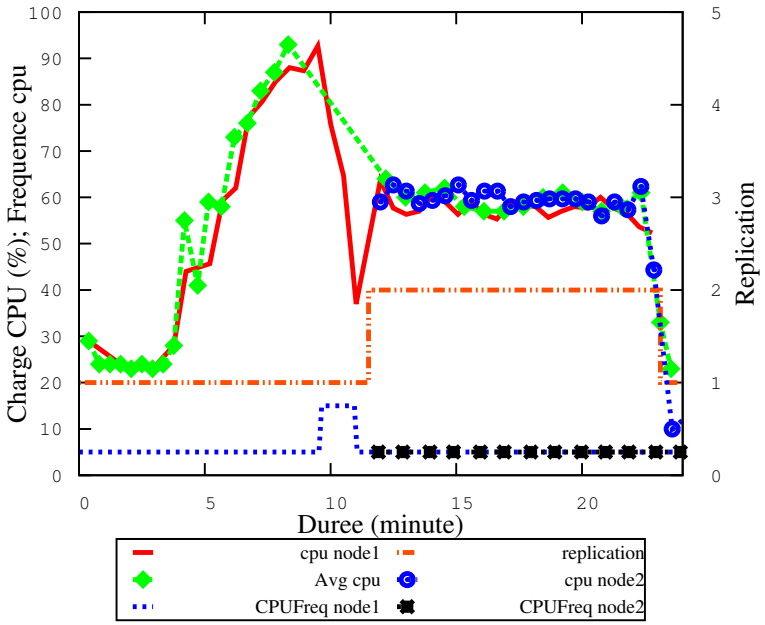

FIGURE 4.10 – Exécution non coordonnée avec : *4750 requêtes/sec*

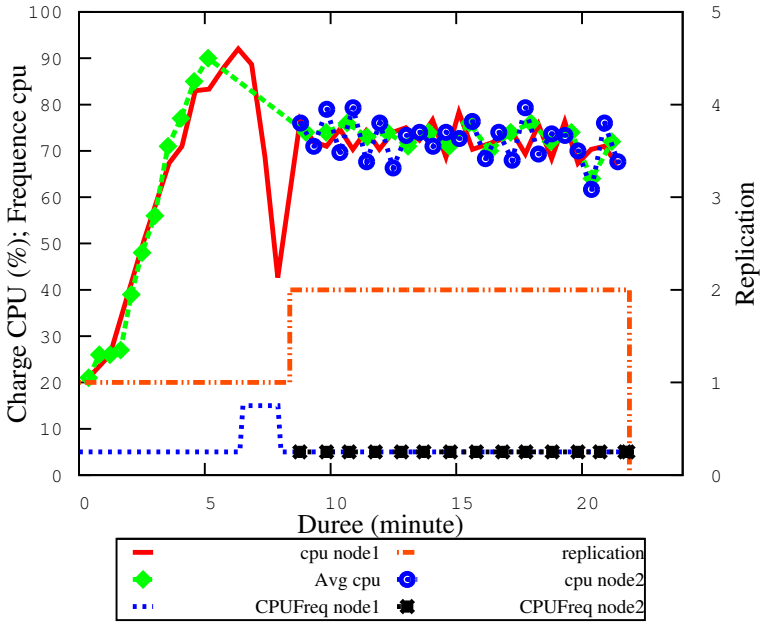

FIGURE 4.11 – Exécution non coordonnée avec : *5000 requêtes/sec*

Les figures 4.10 et 4.11 présentent les exécutions non coordonnées pour les charges **`Workload1`** et **`Workload2`** respectivement. Lors de ces exécutions, la détection d'une surcharge déclenche une hausse de fréquence CPU et l'ajout

du second serveur `Tomcat`. Sur la figure 4.10 la surcharge est détectée par self-sizing environ 8 minutes après le début de l'injection de charge. Cela a conduit à la réaction du gestionnaire qui a ajouté le second `Tomcat` (11 min). Durant cette opération, le Dvfs sur la machine qui héberge le premier `Tomcat` a détecté la surcharge et a haussé la fréquence CPU de la machine (`CPUFreq_node1: 9 min`). Une fois le second serveur `Tomcat` intégré (`node2: 11min`), la charge CPU au niveaux des deux machines actives est autour de 60 pour cent. La fréquence CPU de la première machine est baissée. Le même comportement est observé sur la figure 4.11. Ces deux charges peuvent être traitées avec un seul serveur. Cependant sans coordination, deux serveurs `Tomcat` sont actifs et les machines qui les hébergent sont à la fréquence minimale.

4.4.3.2 Comportement coordonné

Contrairement aux exécutions non coordonnées, pour les charges **`Workload1`** et **`Workload2`**, le gestionnaire `self-sizing` ne réagit pas à la détection de surcharge durant l'exécution coordonnée.

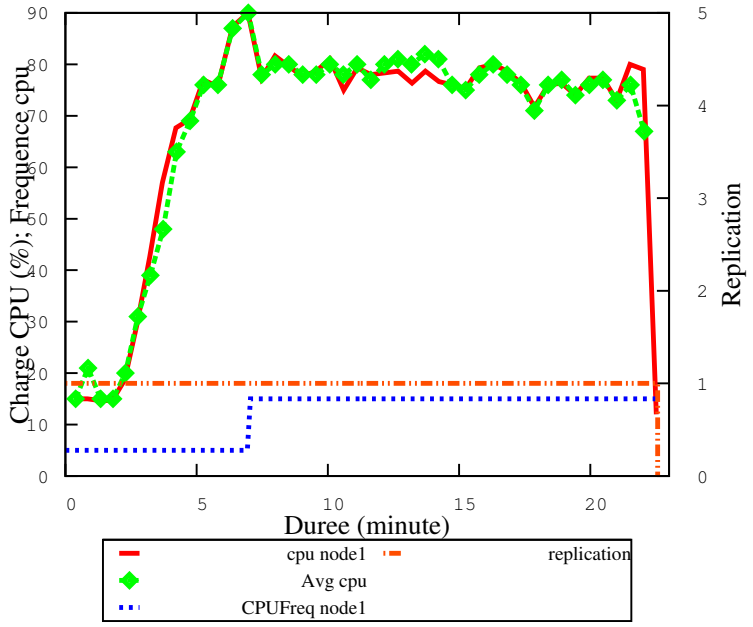

FIGURE 4.12 – Exécution coordonnée avec : *4750 requêtes/sec*

Les figures 4.12 et 4.13 présentent les exécutions coordonnées pour les charges **`Workload1`** et **`Workload2`** respectivement. Sur la figure 4.12 la surcharge

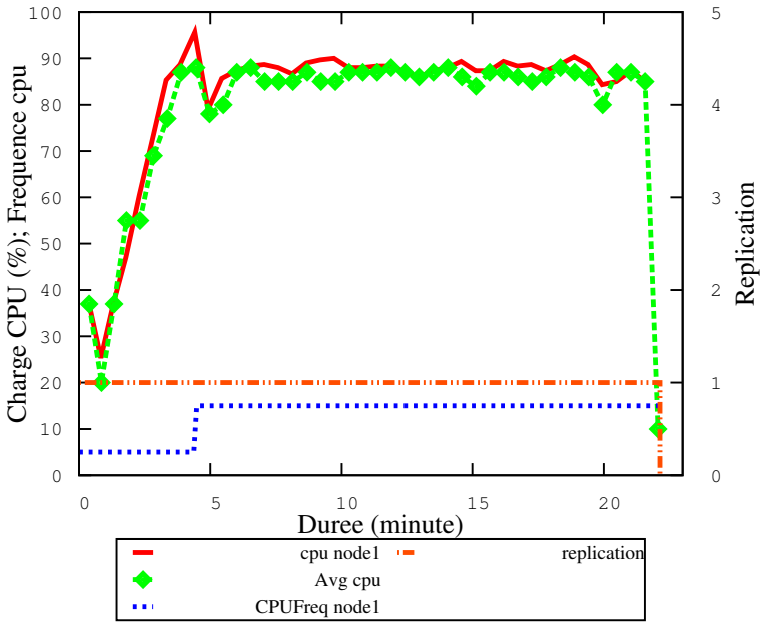

FIGURE 4.13 – Exécution coordonnée avec : *5000 requêtes/sec*

est détectée 7 minutes après le début de l'injection de charge. Cependant un seul serveur `Tomcat` est resté actif durant toute la durée de l'expérimentation. La fréquence CPU de la machine qui héberge le serveur `Tomcat` est haussée par le Dvfs qui s'exécute sur la machine (`CPUFreq_node1: 8 min`). Le même comportement est observé sur la figure 4.13.

Ajout d'un serveur lorsque nécessaire. Le traitement de la charge **`Workload3`** requiert deux serveurs actifs.

Pour la charge **`Workload3`**, dans l'exécution non coordonnée (Figure 4.14) comme dans l'exécution coordonnée (Figure 4.15), le deuxième serveur `Tomcat` est ajouté. Après l'ajout du second `Tomcat`, les machines qui hébergent les deux `Tomcat` sont à la fréquence minimale. Cependant, contrairement à l'exécution non coordonnée, durant l'exécution coordonnée l'ajout du second serveur `Tomcat` (*environ 9 min*) est effectué après que la fréquence maximale du serveur `Tomcat` ait été atteinte. Durant l'exécution coordonnée présentée à la figure 4.15, self-sizing n'a pas réagit à la première détection d'une surcharge (*environ 4 min*). Le Dvfs a réagit en haussant la fréquence des processeurs de la machine hébergeant le serveur `Tomcat` actif. Cependant la surcharge a persisté après que

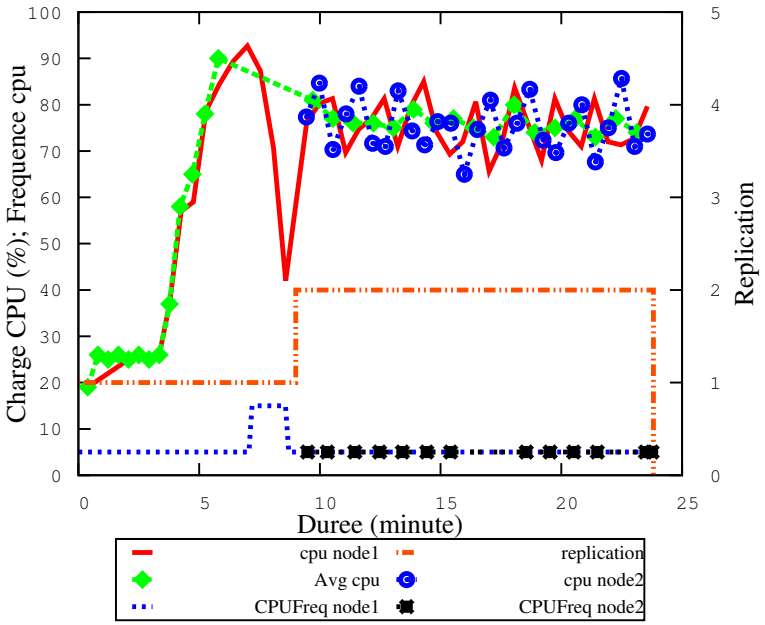

FIGURE 4.14 – Exécution non coordonnée avec : *5542 requêtes/sec*

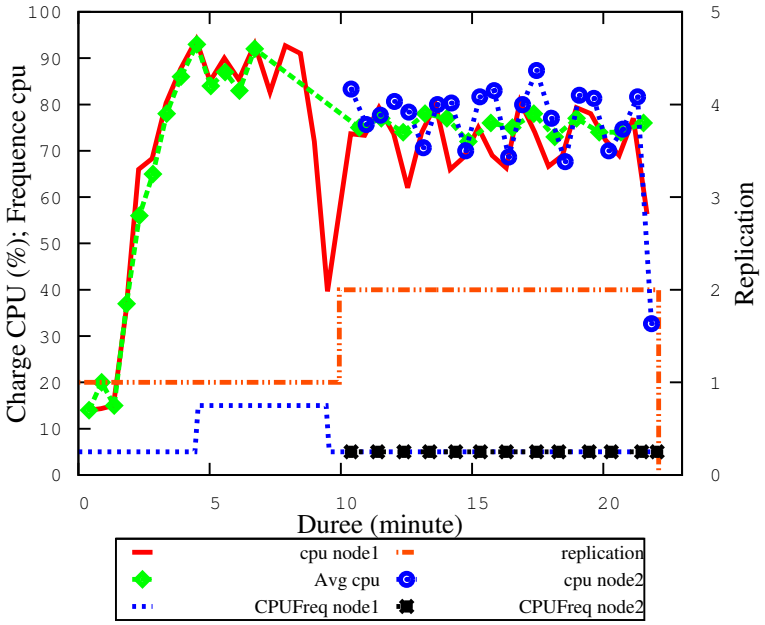

FIGURE 4.15 – Exécution coordonnée avec : *5542 requêtes/sec*

la machine soit à la fréquence maximale. Cela a conduit à l'ajout du second serveur `Tomcat` par self-sizing (*9 min*).

4.5 Conclusion

Le contrôleur de coordination généré n'empêche pas l'ajout d'un nouveau serveur `Tomcat` lorsque cela est nécessaire. Il est en mesure d'assurer le respect de la politique de coordination. Contrairement aux exécutions non coordonnées, où les comportements indésirables ont été observés, on constate que les exécutions coordonnées respectent la politique définie. Les opérations d'ajout d'un nouveau serveur `Tomcat` ne sont effectuées que lorsque le serveur `Tomcat` actif a atteint sa fréquence CPU maximale alors que la charge continue à augmenter menant à la surcharge de ce serveur. Cependant, Il est important que la fréquence d'échantillonnage et la communication soient suffisamment rapide pour pouvoir détecter et traiter les montées de charge efficacement.

Ce chapitre présente un exemple simple pour expliquer la mise en oeuvre de notre approche. Nous verrons dans le chapitre suivant un exemple plus compliqué et plus réaliste.

5

Gestion du dimensionnement dynamique et de la réparation d'un système multi-tiers

Contents

5.1 Gestionnaires autonomes non coordonnés 79
5.1.1 Gestionnaire d'auto-dimensionnement : Self-sizing . . 79
5.1.2 Gestionnaire d'auto-réparation : Self-repair 80
5.2 Problèmes d'administration d'un système multi-tiers 81
5.3 Conception du contrôleur de coordination 84
5.3.1 Modélisation du contrôle des gestionnaires 84
5.3.1.1 Modélisation du contrôle de self-sizing 84
5.3.1.2 Modélisation du contrôle de self-repair 84
5.3.2 Spécification de la coordination 86
5.3.2.1 Stratégie de coordination 86
5.3.2.2 Spécification du contrat 87
5.3.2.3 Programme final 89
5.4 Expérimentations 90
5.4.1 Configuration 91
5.4.2 Évaluation 91
5.4.2.1 Comportement non coordonné 92
5.4.2.2 Comportement coordonné 95
5.5 Conclusion 98

Dans ce chapitre nous utilisons notre approche pour la coordination de gestionnaires autonomes pour garantir la performance, l'optimisation de ressources et la disponibilité d'un système distribué. La performance peut être considérée comme étant la capacité à répondre à plusieurs requêtes simultanément dans un délai acceptable, et la disponibilité comme étant la capacité à résister aux pannes. Pour assurer la performance et la disponibilité d'un système, ce dernier est généralement basé sur la réplication. Les serveurs sont dupliqués sur des machines distinctes et les requêtes sont distribuées aux instances de serveurs par un répartiteur de charge. Cela permet d'améliorer la performance et la tolérance aux pannes.

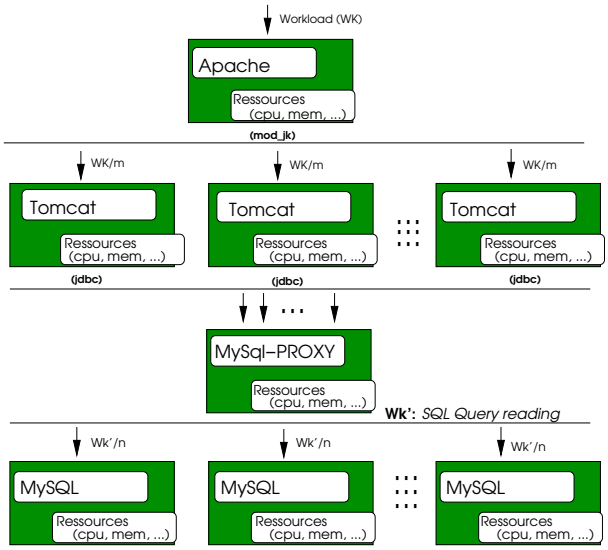

FIGURE 5.1 – Application JEE

Un exemple de système distribué que nous considérons est le système multi-tiers JEE présenté à la figure 5.1. Il est constitué d'un serveur web `Apache` [1], de serveurs d'application `Tomcat` [2] dupliqués, d'un serveur `Mysql-proxy` [3] et de serveurs de bases de données `Mysql` [4] également dupliqués. Les requêtes entrantes sont reçues par le serveur `Apache`. Ce dernier les distribue aux serveurs

1. `http://httpd.apache.org/`
2. `http://tomcat.apache.org/`
3. `http://dev.mysql.com/doc/refman/5.1/en/mysql-proxy.html`
4. `http://www.mysql.com/`

`Tomcat` pour leur traitement. Les serveurs `Tomcat` accèdent aux bases de données via le serveur `Mysql-proxy` qui est un répartiteur de charge pour les serveurs `Mysql`. Le serveur `Mysql-proxy` distribue équitablement les requêtes de lecture aux serveurs `Mysql`.

L'une des difficultés lors du déploiement de ce genre de système est le dimensionnement. La variation du nombre de requêtes à traiter fait qu'il peut être difficile d'estimer le nombre de serveurs dupliqués à utiliser lors du démarrage du système. Une configuration statique du nombre de serveurs peut conduire la plupart du temps à une estimation abusive du nombre de serveurs. Cela peut, peut-être, permettre d'avoir de bonnes performances, mais avec un coût très élevé, e.g., consommation énergétique élevée. Ajuster dynamiquement le degré de réplication à l'exécution permet d'allouer le nombre nécessaire de serveurs en fonction du nombre de requêtes à satisfaire. De plus l'état des serveurs doit être surveillé en permanence pour détecter les pannes. Il est nécessaire de réparer les pannes afin d'éviter de perdre tous les serveurs. Pour cela, des gestionnaires comme self-sizing et self-repair peuvent être utilisés pour la gestion du dimensionnement dynamique et la réparation de serveurs d'un système multi-tiers. Toutefois, la coordination de ces gestionnaires peut être nécessaire pour éviter des opérations incohérentes. En effet l'occurrence de panne dans un tier répliqué peut avoir un impact sur la charge des serveurs restants au niveau du tier. Elle peut également avoir un impact sur la charge des tiers qui lui succèdent dans la chaîne de traitement. Cela peut conduire à une mauvaise interprétation de la charge à traiter et une mauvaise évaluation du nombre de serveurs nécessaires pour traiter les requêtes.

5.1 Gestionnaires autonomes non coordonnés

Les gestionnaires assurent la gestion de la disponibilité, la performance et l'optimisation des ressources de calcul. Ces derniers sont conçus indépendamment.

5.1.1 Gestionnaire d'auto-dimensionnement : Self-sizing

Nous ré-utilisons le même gestionnaire décrit à la section 4.1.1.

5.1.2 Gestionnaire d'auto-réparation : Self-repair

Le gestionnaire self-repair, représenté dans la figure 5.2, est dédié à la restauration d'un système ou les éléments constituant le système suite à l'occurrence de pannes. Il traite les pannes franches de machines. Il a une connaissance de la structure du système administré. Il connaît l'ensemble des machines sur lesquelles s'exécutent les éléments logiciels constituant le système, ainsi que l'ensemble des ressources matérielles non utilisées et disponibles pour permettre la reconfiguration du système.

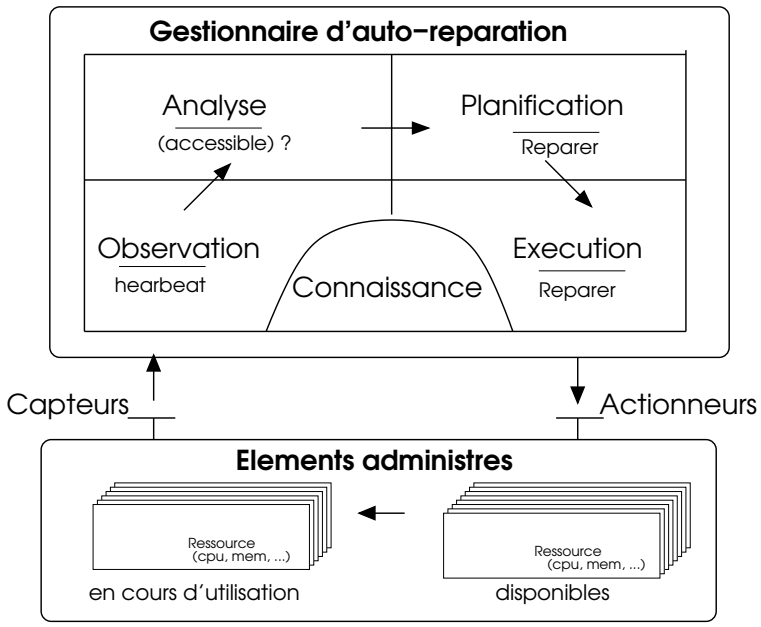

FIGURE 5.2 – Gestionnaire d'auto-réparation

Comme le montre la figure 5.2, des sondes contactent périodiquement les machines en cours d'utilisation afin de vérifier leur accessibilité. Les sondes utilisées dans cet exemple sont de type `Ping`. Si aucune réponse n'est reçue après l'écoulement du temps de latence alors la machine qui ne répond pas est considérée en panne. Lorsqu'une machine est considérée comme en panne, le gestionnaire détermine les éléments logiciels et matériels qui sont affectés par cette panne. Il détermine ensuite les logiciels que la machine défaillante exécutait et ceux qui sont liés à ces logiciels. Cette analyse est effectuée sur la base de la connaissance que le gestionnaire a de l'état courant et de la structure de système. Sans cette connaissance, la restauration du système ne peut être

réalisée car l'information nécessaire pour la reconstruction serait perdue avec la défaillance. Une fois que les éléments impactés par la panne sont identifiés, le gestionnaire planifie la reconstruction du système. Cette reconstruction consiste à redémarrer sur d'autres machines disponibles les logiciels qui s'exécutaient sur la machine défaillante et rétablir les liaisons entre les éléments. L'exécution des opérations de reconfiguration est effectuée via les actionneurs fournis par le système administré qui permettent l'allocation de machines, le déploiement et la configuration des éléments logiciels et matériels du système.

Ce gestionnaire permet la disponibilité du système administré en restaurant le service après une défaillance d'une machine. Dans le cas d'un système basé sur la réplication de serveurs, il permet la restauration du degré de redondance des serveurs. Cela permet de tolérer jusqu'à *m-1* pannes de serveurs durant le temps moyen de réparation (MTTR).

5.2 Problèmes d'administration d'un système multi-tiers

Dans un système basé sur une architecture multi-tiers, une panne d'un serveur d'un des tiers peut affecter le tier et ceux qui suivent dans la chaîne de traitement des requêtes.

L'occurrence d'une panne au niveau du tier peut entraîner une baisse de charge au niveau des serveurs des autres tiers qui suivent. Ces derniers risquent de ne plus recevoir autant de requêtes à traiter qu'avant la panne. Cela peut causer une sous-charge au niveau de ces tiers. Par exemple, sur la figure 5.3, la panne du serveur `Apache` entraîne une baisse de charge au niveau des tiers `Tomcat`, `Mysql-Proxy` et `Mysql`. Ces derniers ne reçoivent plus de requêtes car le serveur `Apache` est l'entrée du système. Sur la figure 5.4, la panne du serveur `Mysql-Proxy` entraîne une baisse de charge au niveau du tier `Mysql` car le serveur `Mysql-Proxy` reçoit les requêtes à transmettre aux serveurs `Mysql`.

Dans le cas d'un tier basé sur la répartition de charge, la panne d'un des serveurs peut entraîner une surcharge des autres serveurs. Lorsqu'un des serveurs dupliqués tombe en panne la charge qu'il doit traiter est répartie entre les autres serveurs restants. Cela peut entraîner une hausse de charge au niveau de ces derniers et peut causer la saturation des machines qui les exécutent. Sur

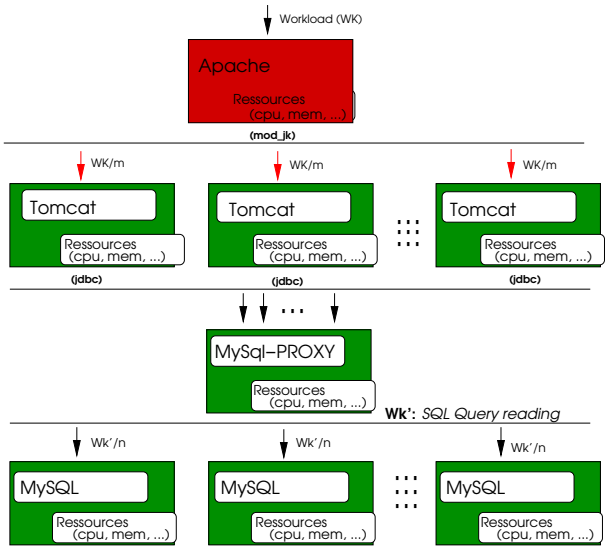

FIGURE 5.3 – Panne du serveur `Apache`

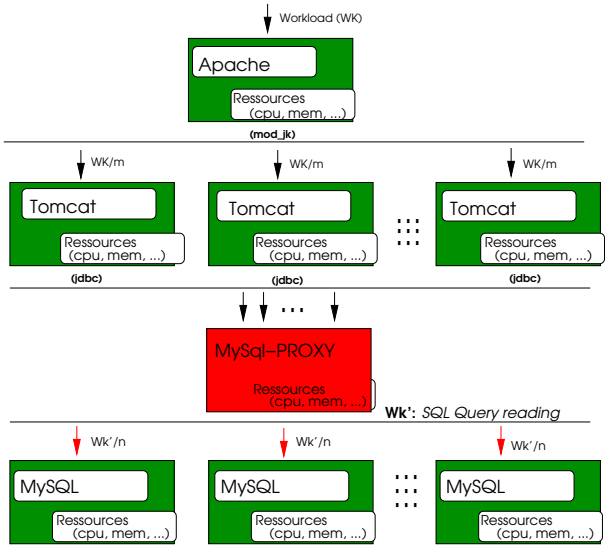

FIGURE 5.4 – Panne du serveur `Mysql-Proxy`

la figure 5.5, la panne d'un serveur `Tomcat` entraîne une hausse de la charge des autres serveurs `Tomcat` actifs. Sur la figure 5.6, la panne d'un serveur `mysql` entraîne une hausse de la charge des autres serveurs `Mysql` actifs.

Lorsque des instances des gestionnaires self-sizing et self-repair sont utilisées pour gérer les différents tiers d'un système de ce type, les pannes peuvent entraîner des opérations d'administration inutiles. En effet l'occurrence d'une

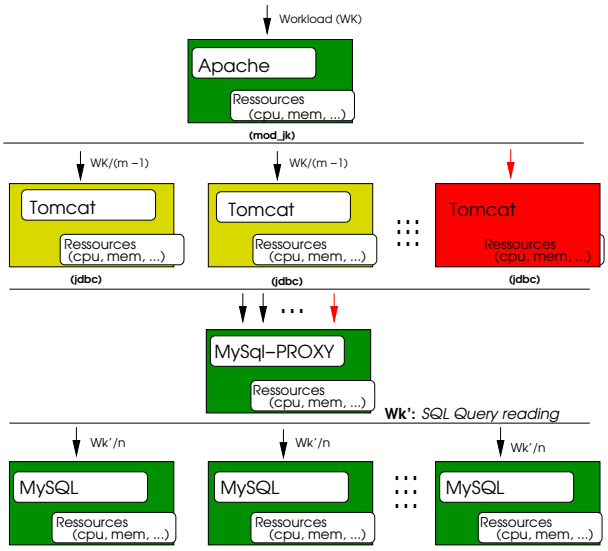

FIGURE 5.5 – Panne d'un serveur `Tomcat`

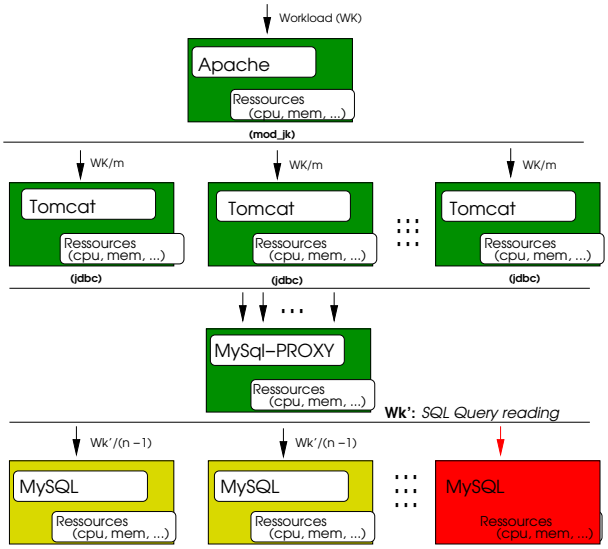

FIGURE 5.6 – Panne d'un serveur `MySQL`

panne d'un serveur d'un tier, détectée par le self-repair gérant ce tier, entraîne l'exécution d'une opération de restauration du serveur. Cependant cette panne peut occasionner une surcharge des autres serveurs durant la réparation. Dans ce cas le self-sizing dédié à la gestion du dimensionnement du tier où la panne s'est produite peut entamer une opération d'ajout d'un nouveau serveur alors que la panne est entrain d'être traitée. Si la charge n'a pas varié alors il y aura

un serveur en trop et qui sera probablement arrêté par le self-sizing après la réparation. La panne peut également conduire à la baisse de la charge des tiers suivants. Si les instances de self-sizing au niveau de ces tiers détectent une sous-charge, ils vont enlever des serveurs pour optimiser les ressources. Mais une fois la panne réparée, si la charge n'a pas varié cela peut conduire à la saturation des tiers jusqu'à ce que les serveurs soient relancés par les self-sizing. Durant cette période une dégradation de la performance du système peut être observée.

5.3 Conception du contrôleur de coordination

Cette section présente la conception d'un contrôleur de coordination des instances de self-repair et self-sizing pour l'administration du système multi-tiers présenté à la figure 5.1. Nous modélisons le contrôleur de coordination comme la composition des comportements des gestionnaires autonomes à laquelle est associée une politique de coordination pour éviter les comportements incohérents.

5.3.1 Modélisation du contrôle des gestionnaires

Cette section décrit les modèles des gestionnaires autonomes. Chaque gestionnaire est modélisé par un ou plusieurs automates qui décrivent son comportement et le contrôle des actions d'administration qu'il peut exécuter.

5.3.1.1 Modélisation du contrôle de self-sizing

Nous ré-utilisons le modèle de self-sizing défini à la section 4.3.1.1.

5.3.1.2 Modélisation du contrôle de self-repair

Le modèle de self-repair est constitué de deux automates, représentés à la figure 5.7.

L'automate à gauche modélise le contrôle des actions de réparation. Il a deux états : `Enable` et `Disable`. L'état `Enable`, état initial, indique que les actions de réparation sont autorisées et l'état `Disable` indique que les actions de réparation

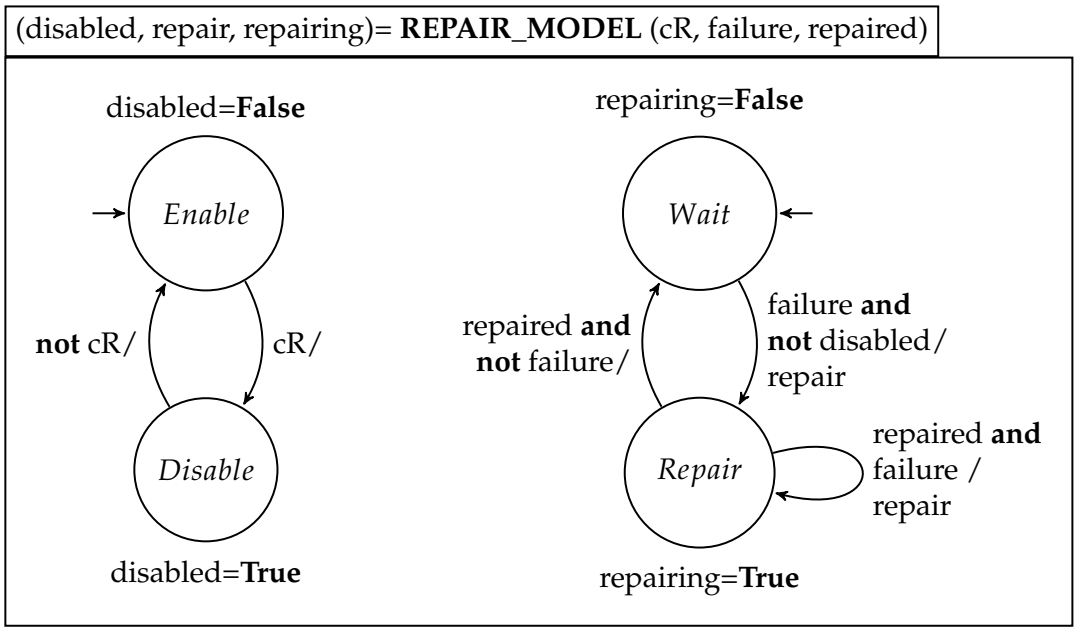

FIGURE 5.7 – Modéle de contrôle de self-repair

sont interdites. Le changement d'états est contrôlé par l'entrée *cR*. Lorsqu'elle est à **true** les actions sont interdites. L'état du contrôle des actions de réparation est indiqué en sortie par la variable d'état `disabled` qui est à **true** lorsque les actions sont interdites.

L'automate à droite modélise le comportement du gestionnaire. Il est constitué de deux états : `Wait` et `Repair`. L'état `Wait`, état initial, représente l'état dans lequel self-repair attend la détection d'une panne. L'état `Repair` représente l'état dans lequel le gestionnaire est en train d'effectuer la réparation de la panne. L'occurrence d'une panne est représentée par l'entrée `failure` à **true**. Dans l'état `Wait`, à l'occurrence d'une panne, self-repair réagit, si autorisée, en produisant l'action `repair` et se met dans l'état `Repair`. La fin de la réparation est représentée par l'entrée `repaired` à **true**. Si aucune panne est détectée, self-repair retourne dans l'état d'attente `Wait`. En cas de panne, il réagit en réparant la panne. La sortie `repairing` indique l'état courant de l'automate. Elle est à **true** lorsque l'automate est dans l'état `Repair`.

5.3.2 Spécification de la coordination

La coexistence des gestionnaires self-sizing et self-repair est représentée par la composition d'instances des automates qui décrivent leur comportement, illustrée à la figure 5.8. Nous avons quatre instances du modèle du gestionnaire self-repair et deux instances du modèle du gestionnaire self-sizing.

```
(A_disabled, A_repair_server, A_repairing,
 T_disabled, T_repair_server, T_repairing,
 T_disabledUp, T_disabledDown, T_add_server, T_remove_server, T_adding, T_max, T_min,
 ...) = MULTITIER (A_c, T_c, P_c, M_c, T_cUp, T_cDown, M_cUp, M_cDown,
                   A_failure, A_repaired,
                   T_failure, T_repaired,
                   T_Underload, T_Overload, T_max_machine, T_min_machine...)
```

```
(A_disabled, A_repair_server, A_repairing) = REPAIR_MODEL (A_c, A_failure, A_repaired);

(T_disabled, T_repair_server, T_repairing) = REPAIR_MODEL (T_c, T_failure, T_repaired);
(T_disabledUp, T_disabledDown, T_add_server, T_remove_server, T_adding, T_max, T_min) =
    SIZING_MODEL (T_cUp, T_cDown, T_Underload, T_Overload, T_max_machine, T_min_machine...);

(P_disabled, P_repair_server, P_repairing) = REPAIR_MODEL (P_c, P_failure, P_repaired);

(M_disabled, M_repair_server, M_repairing) = REPAIR_MODEL (M_c, M_failure, M_repaired);
(M_disabledUp, M_disabledDown, M_add_server, M_remove_server, M_adding, M_max, M_min) =
    SIZING_MODEL (M_cUp, M_cDown, M_Underload, M_Overload, M_max_machine, M_min_machine...);
```

FIGURE 5.8 – Composition des modèles de self-sizing et self-repair

Les entrées et sorties des instances d'automates sont renommées, en ajoutant un préfixe, pour distinguer les gestionnaires qu'ils représentent : "`A_`" pour le gestionnaire dédié au tier `Apache`, "`T_`" pour le tier `Tomcat`, "`P_`" pour le tier `Mysql-proxy`, et " `M_`" pour le tier `Mysql`. Par exemple, l'entrée `A_failure` représente une panne du serveur `Apache`. Les entrées de la composition **`MULTITIER`** correspondent à l'union de toutes les entrées des automates contenus. Les sorties de la composition correspondent également à l'union des sorties des automates.

5.3.2.1 Stratégie de coordination

La stratégie de coordination consiste à empêcher les gestionnaires self-sizing d'exécuter des actions d'ajout ou de retrait de serveurs en cas de panne. En effet, une panne d'un serveur d'un tier peut entraîner une surcharge au niveau du tier et une sous-charge au niveau des tiers qui suivent. De ce fait, traiter d'abord la panne avant de prendre en compte les événements de surcharge ou de sous-charge peut être plus pertinent. Cela peut empêcher des réactions inutiles

qui conduisent à des ajouts et/ou des retraits de serveurs. Le contrôle des actions des gestionnaires self-sizing est basé sur les activités des gestionnaires self-repair. La stratégie est décrite ci-dessous de manière textuelle :

Condition 1 : Ignorer la surcharge détectée au niveau du tier `Tomcat` — En cas de panne dans le tier `Tomcat`, toute surcharge dans ce tier est ignorée tant que la réparation n'est pas terminée.

Condition 2 : Ignorer la sous-charge détectée au niveau du tier `Tomcat` — En cas de panne dans le tier `Apache`, toute sous-charge dans le tier `Tomcat` est ignorée tant que la réparation de la panne n'est pas terminée.

Condition 3 : Ignorer la surcharge détectée au niveau du tier `Mysql` — En cas de panne dans le tier `Mysql`, toute surcharge dans ce tier est ignorée tant que la réparation n'est pas terminée.

Condition 4 : Ignorer la sous-charge détectée au niveau du tier `Mysql` — En cas de panne dans les tiers `Apache`, `Tomcat` et `Mysql-proxy`, toute sous-charge dans le tier `Mysql` est ignorée tant que la réparation de la panne n'est pas terminée.

5.3.2.2 Spécification du contrat

Nous définissons la stratégie sous forme de propriétés d'invariance. Ces propriétés sont exprimées via les sorties des automates à la figure 5.8. Certaines propriétés sont spécifiées dans le contrat et assurées par le contrôleur généré. D'autres sont manuellement programmées. Ci-dessous nous décrivons les spécifications formelles de la stratégie de coordination.

Condition 1. Cette condition consiste à ignorer les surcharges détectées dans le tier `Tomcat` en cas de panne d'un serveur `Tomcat`. Lors de la réparation d'un `Tomcat`, le gestionnaire self-sizing chargé du dimensionnement dynamique de ce tier ne doit pas réagir aux surcharges. Cela est exprimé par :

$invariant_{1.1}$ = `T_repairing` **XOR not** `T_disabledUp`

`T_repairing` étant à **`true`** exprime le fait que la réparation d'un serveur `Tomcat` est en cours. (**not** `T_disabledUp`) étant à **`true`** exprime le fait que les opérations d'ajout de self-sizing sont autorisées. Empêcher (**not** `T_disabledUp`)

d'être à **true** quand T_repairing est à **true** assure qu'aucune opération d'ajout ne sera exécutée durant une réparation.

$invariant_{1.2}$ = **not** (T_repair_server **and** T_add_server)

T_repair_server étant à **true** exprime le fait que self-repair entame une opération de réparation. T_add_server étant à **true** exprime le fait que self-sizing entame une opération d'ajout. Cette propriété empêche, au niveau du tier Tomcat, l'activation d'une opération d'ajout et d'une opération de réparation dans la même réaction.

Condition 2. Cette condition consiste à ignorer les sous-charges détectées dans le tier Tomcat en cas de panne du serveur Apache. Cela implique l'inhibition des opérations de retrait de serveurs au niveau du tier Tomcat lorsque le serveur apache est en train d'être réparé :

$invariant_{2.1}$ = A_repairing **XOR not** T_disabledDown

Cette propriété permet d'inhiber les opérations de retrait au niveau du tier Tomcat lorsque le serveur Apache est en cours de réparation.

$invariant_{2.2}$ = **not** (A_repair_server **and** T_remove_server)

Cette propriété empêche l'activation d'une opération de retrait de serveur au niveau du tier Tomcat et d'une opération de réparation du serveur Apache dans la même réaction.

Condition 3. Cette condition consiste à ignorer les surcharges détectées dans le tier Mysql en cas de panne d'un serveur Mysql en cours de réparation. Lors de la réparation d'un Mysql, le gestionnaire self-sizing chargé du dimensionnement dynamique de ce tier ne doit pas réagir aux surcharges. Cela est exprimé par :

$invariant_{3.1}$ = M_repairing **XOR not** M_disabledUp

Cette propriété est similaire à $invariant_{1.1}$.

$invariant_{3.2}$ = **not** (M_repair_server **and** M_add_server)

Cette propriété est similaire à $invariant_{1.2}$.

Condition 4. Cette condition consiste à ignorer les sous-charges détectées dans le tier Mysql en cas de panne du serveur Apache, du serveur Mysql-Proxy

ou d'un serveur Tomcat et en cours de réparation. Cela implique l'inhibition des opérations de retrait de serveurs au niveau du tier Mysql lorsqu'un serveur au niveau des tiers qui précédent est en train d'être réparé :

Soit APT_repairing correspondant à (A_repairing **or** P_repairing **or** T_repairing), et APT_repair_server correspondant à (A_repair_server **or** P_repair_server **or** T_repair_server).

$invariant_{4\cdot1}$ = APT_repairing **XOR not** M_disabledDown

La propriété $invariant_{4\cdot1}$ permet l'inhibition des opérations de retrait au niveau du tier Mysql lorsque des réparation sont en cours au niveau des autres tiers.

$invariant_{4\cdot2}$ = **not** (APT_repair_server **and** M_remove_server)

La propriété $invariant_{4\cdot2}$ empêche l'activation d'une opération de retrait de serveur au niveau du tier Mysql et d'une opération de réparation d'un serveur dans les autres tiers, dans la même réaction.

5.3.2.3 Programme final

```
(A_repair_server, T_repair_server, T_add_server, T_remove_server...) =
    COORDINATED_MULTITIER (A_failure, A_repaired,
                           T_failure, T_repaired,
                           T_Underload, T_Overload, T_max_machine, T_min_machine...)
enforce (invariant1·1 and invariant2·1 and invariant3·1 and invariant4·1 and
         invariant1·2 and invariant2·2 and invariant3·2 and invariant4·2)
with A_c, T_c, P_c, M_c, T_cUp, T_cDown, M_cUp, M_cDown

 APT_failure = A_failure or P_failure or T_failure;
     ...
  T_Overload' = not T_failure and T_Overload;
 T_Underload' = not A_failure and T_Underload;
  M_Overload' = not M_failure and M_Overload;
 M_Underload' = not APT_failure and M_Underload;
     ...
(A_disabled, A_repair_server, A_repairing,
 T_disabled, T_repair_server, T_repairing,
 T_disabledUp, T_disabledDown, T_add_server, T_remove_server, T_adding, T_max, T_min,
 ...) = MULTITIER (A_c, T_c, P_c, M_c, T_cUp, T_cDown, M_cUp, M_cDown,
                   A_failure, A_repaired,
                   T_failure, T_repaired,
                   T_Underload', T_Overload', T_max_machine, T_min_machine...);
```

FIGURE 5.9 – Coordination des instances de self-sizing et self-repair

La figure 5.9 décrit le modèle de la coordination des gestionnaires. Ce modèle est constitué du modèle de la coexistence des gestionnaires auquel

est associé un contrat exprimant la stratégie de coordination. A la compilation, `Heptagon/BZR` génère la logique de contrôle qui restreint la composition **`MULTITIER`** aux comportements qui respectent les propriétés $invariant_{1.1}$, $invariant_{2.1}$, $invariant_{3.1}$ et $invariant_{4.1}$. Cette logique de contrôle est automatiquement intégrée dans le modèle global.

Propriétés programmées manuellement. Les propriétés $invariant_{1.2}$, $invariant_{2.2}$, $invariant_{3.2}$ et $invariant_{4.2}$ sont assurées par programmation du code `Heptagon/BZR` qui les réalise et non par SCD. Elles sont vérifiées à la compilation. La compilation réussit si ces propriétés sont satisfaites.

Pour la propriété $invariant_{1.2}$, nous définissons une variable `T_Overload'` qui permet de filtrer les surcharges en fonction des pannes détectées. `T_Overload'` est à **`true`** lorsque `T_Overload` est à **`true`** et `T_failure` est à **`false`**. Nous remplaçons `T_Overload` par `T_Overload'` pour la notification d'une surcharge au modèle du self-sizing du tier Tomcat. La valeur de `T_Overload'` est définie par : « **`not`** `T_failure` **`and`** `T_Overload` ». Cela permet de ne pas notifier une surcharge lorsqu'une panne est détectée dans la même réaction. Cette expression permet d'assurer la propriété $invariant_{1.2}$. Le même principe est utilisé pour les autres propriétés programmées manuellement.

5.4 Expérimentations

L'objectif des expérimentations est d'évaluer le comportement du contrôleur de coordination construit pour coordonner les gestionnaires self-sizing et self-repair afin d'éviter des décisions d'administration incohérentes.

La plate-forme expérimentale est constituée de machines munies de processeurs Dual-Core de 1.66Ghz et 1.9Go de mémoire, de machines munies de processeurs Dual-Core de 2,53 Ghz et 3.4Go de mémoire, et des machines munies de processeurs Dual-Core de 1.20Ghz et 1.5Go de mémoire. Les machines sont inter-connectées via un réseau Ethernet (1 Gbit/s). Pour l'ensemble des expérimentations, une seule instance de serveur `Apache` et une instance de serveur `Mysql-Proxy` sont utilisées. Les tiers `Tomcat` et `Mysql` sont basés sur la réplication. Chaque machine héberge un seul serveur.

Quatre instances du gestionnaire self-repair sont utilisées, chaque tier est

géré par une instance pour la gestion des pannes. Les tiers basés sur la réplication sont les tiers `Tomcat` et `Mysql`. Deux instances du gestionnaire self-sizing sont utilisées, chacun de ces deux tiers est géré par une instance pour le dimensionnement dynamique du nombre de serveurs dupliqués actifs. Le contrôleur de coordination généré est responsable de la coordination des actions des quatre instances de self-repair et des deux instances de self-sizing.

5.4.1 Configuration

L'évaluation a été réalisée avec l'application multi-tiers JEE de référence RUBiS [17]. Elle implante un site de vente aux enchères [50] et définit plusieurs types d'interactions Web (e.g., l'enregistrement de nouveaux utilisateurs, la navigation, l'achat ou la vente d'objets). Le déploiement de RUBiS est basé sur une architecture distribuée constituée d'un front-end et d'un back-end. Le front-end est un cluster constitué des serveurs d'application `Tomcat` dupliqués et d'un serveur web `Apache` comme équilibreur de charge. Et le back-end est un cluster constitué des serveurs de bases de données `Mysql` et du serveur `Mysql-proxy` comme équilibreur de charge pour les serveurs `Mysql`. L'évaluation représente un réel problème qui peut être rencontré dans un environnement de *cloud computing* avec des capacités de recouvrement et des capacités d'élasticité.

Initialement lors de chaque exécution, le système est démarré avec un serveur au niveau de chaque tier, c'est à dire, un serveur `Apache`, un serveur `Tomcat`, un serveur de `Mysql-proxy` et un serveur `Mysql`. Nous injectons une charge croissante (correspondant à la période de la création des threads qui simulent les actions des clients) puis une charge constante (correspondant à l'achèvement de la création de l'ensemble des «clients/threads»). Nous attendons qu'il y ait deux serveurs `Tomcat` actifs et deux serveurs `Mysql` actifs pour déclencher des pannes.

5.4.2 Évaluation

Nous avons effectué des expérimentations durant lesquelles les gestionnaires ne sont pas coordonnés et d'autres durant lesquelles les gestionnaires sont coordonnés. La même configuration et le même profil de charge de travail sont utilisés lors des différentes exécutions aussi bien non coordonnées que

coordonnées. Les expérimentations non coordonnées permettent de voir le comportement des gestionnaires suite à l'occurrence d'une panne. Les expérimentations coordonnées permettent de voir si le contrôleur de coordination contrôle les gestionnaires afin d'assurer le respect de la politique de coordination.

Nous déclenchons des pannes lorsque la charge est constante pour voir leurs impacts au niveau des tiers et aussi pour voir comment les différentes instances des gestionnaires réagissent.

5.4.2.1 Comportement non coordonné

Comme dit plus haut, une panne au niveau d'un tier peut avoir un impact sur le tier mais également sur les tiers qui suivent.

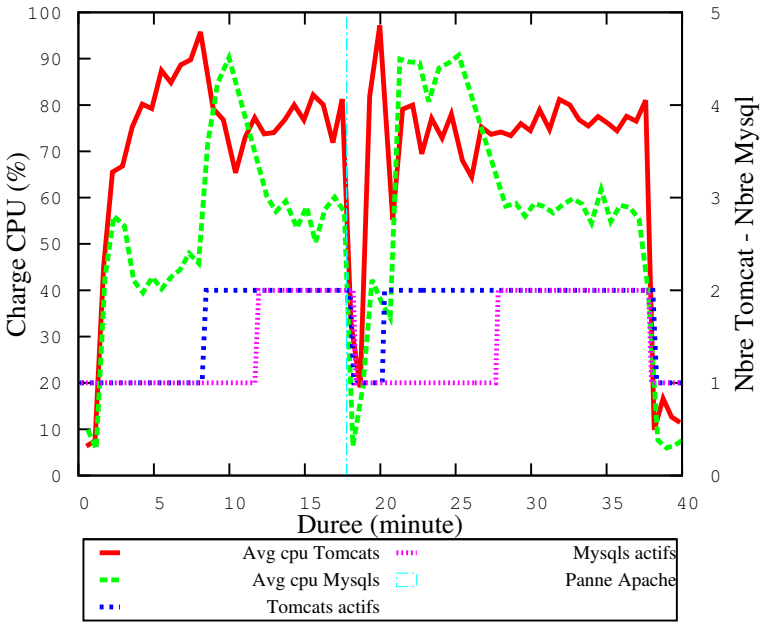

FIGURE 5.10 – Exécution non coordonnée : Panne du serveur Apache

La figure 5.10 présente une exécution durant laquelle le serveur `Apache` tombe en panne. L'occurrence de la panne du serveur `Apache`, 17 minutes après le début de l'exécution, provoque une baisse de charge au niveau des tiers `Tomcat` et `Mysql` (18 min). Une sous-charge est détectée au niveau de ces tiers et a conduit au retrait de serveurs dupliqués. Cependant après la réparation

du serveur `Apache` (19 min), la charge est redevenue normale et les serveurs précédemment retirés ont été démarrés à nouveau à cause d'une sur-charge (`Tomcat` : 20 min et `Mysql` : 25 min).

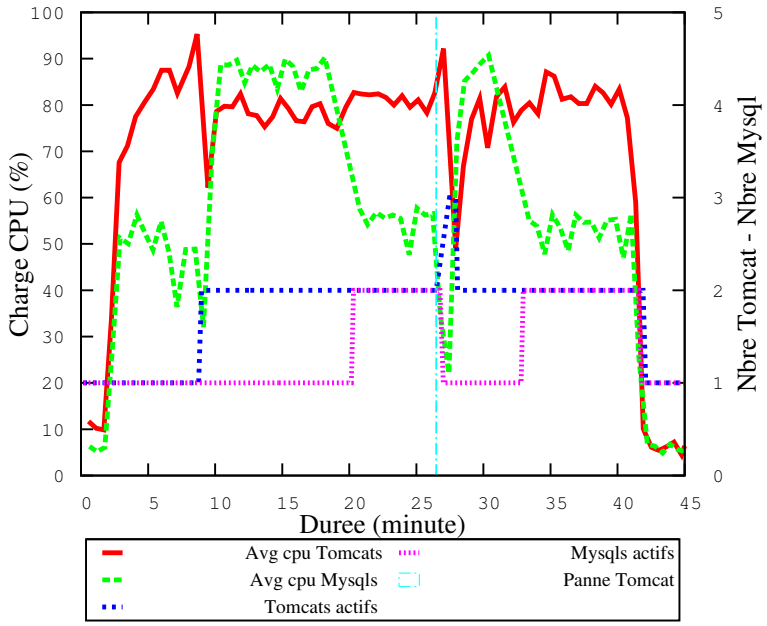

FIGURE 5.11 – Exécution non coordonnée : Panne d'un serveur Tomcat

La figure 5.11 présente une exécution durant laquelle un serveur `Tomcat` tombe en panne. Cette panne, survenue 26 minutes après le début de l'exécution, a provoqué un hausse de charge au niveau du tier (27 min) et aussi une baisse de charge au niveau du tier `Mysql` (27 min). La hausse de charge a conduit à une surcharge du tier `Tomcat` conduisant à l'ajout d'un nouveau serveur `Tomcat`. La baisse de charge au niveau du tier `Mysql` a conduit au retrait d'un serveur à cause d'une sous-charge des serveurs. Mais après la restauration (28 min) du serveur `Tomcat` tombé en panne, le serveur `Tomcat` précédemment ajouté est retiré et le serveur `Mysql` qui était retiré est rajouté à nouveau à cause d'une surcharge (31 min).

La figure 5.12 présente une exécution durant laquelle le serveur `Mysql-proxy` tombe en panne. L'occurrence de la panne du serveur `Mysql-proxy`, survenue 17 minutes après le début de l'exécution, provoque une baisse de charge au niveau du tier `Mysql` (18 min). Une sous-charge est détectée et a conduit au retrait d'un serveur `Mysql`. Cependant après la réparation du serveur `Mysql-proxy`

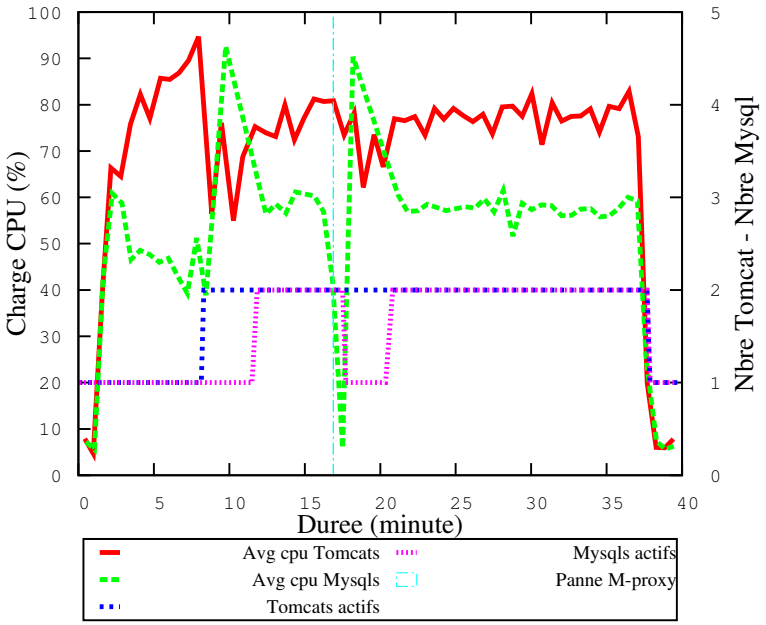

FIGURE 5.12 – Exécution non coordonnée : Panne du serveur Mysql-proxy

(19 min), la charge est redevenue normale et le serveur précédemment retiré a été démarré à nouveau à cause d'une sur-charge (20 min).

La figure 5.13 présente une exécution durant laquelle un serveur `Mysql` tombe en panne. Cette panne, survenue 18 minutes après le début de l'exécution, a provoqué une surcharge du serveur `Mysql` restant (20 min). Cela a conduit à un ajout d'un serveur `Mysql` alors que celui tombé en panne est en cours de restauration. Après la restauration, le serveur ajouté est retiré à cause d'une baisse de charge (22 min).

Les pannes peuvent entraîner des réactions des instances du gestionnaire self-sizing à cause de leur impact sur la répartition de la charge à traiter. Cependant, à moins que la charge en entrée ait changé, les actions de ces instances de self-sizing ne sont pas nécessaires. En effet, une fois les pannes réparées, le degré de réplication, observé avant les pannes, est restauré.

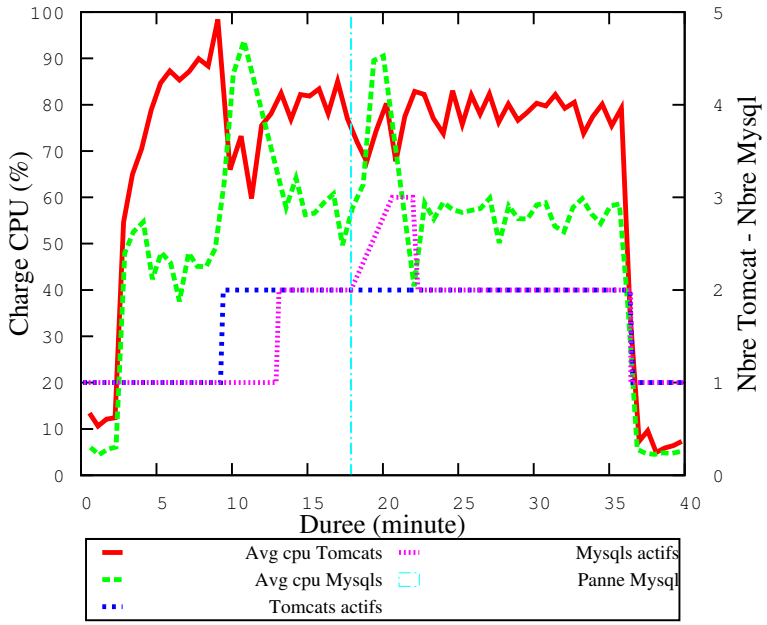

FIGURE 5.13 – Exécution non coordonnée : Panne d'un serveur Mysql

5.4.2.2 Comportement coordonné

Les figures suivantes présentent les exécutions durant lesquelles les instances de self-sizing et de self-repair sont coordonnées. La coordination est assurée par le contrôleur modélisé à la section 5.3.

La figure 5.14 présente une exécution durant laquelle le serveur `Apache` tombe en panne. L'occurrence de la panne (min. 19) provoque une diminution de la charge à la fois au niveau du tier `Tomcat` et au niveau du tier `Mysql`. Une sous-charge (10% de charge CPU) au niveau de ces tiers est observée cependant aucune opération de retrait de serveur est exécutée ni au niveau du tier `Tomcat` ni au niveau du `tier Mysql`. A la fin de la réparation du serveur `Apache`, la charge est redevenue normale au niveau des tiers `Tomcat` et `Mysql`.

La figure 5.15 présente une exécution durant laquelle un serveur `Tomcat` est en panne. La panne survient 17 minutes après le début de l'expérimentation et provoque une hausse de la charge du serveur `Tomcat` restant. Cette hausse de charge a conduit à une surcharge du serveur `Tomcat` restant (19 min). Cependant aucune opération d'ajout de serveur est exécutée par l'instance de self-sizing qui gère le tier `Tomcat`. Une baisse de la charge est également observée au

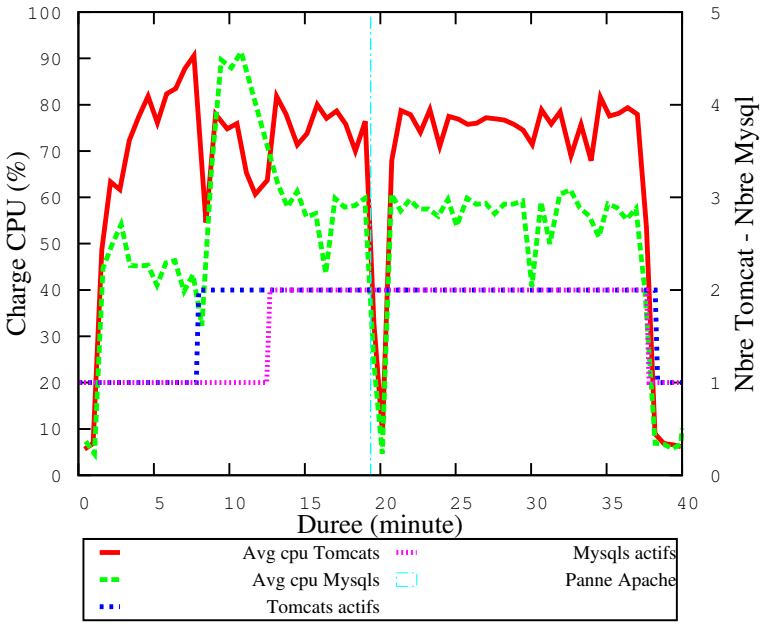

FIGURE 5.14 – Exécution coordonnée : Panne du serveur Apache

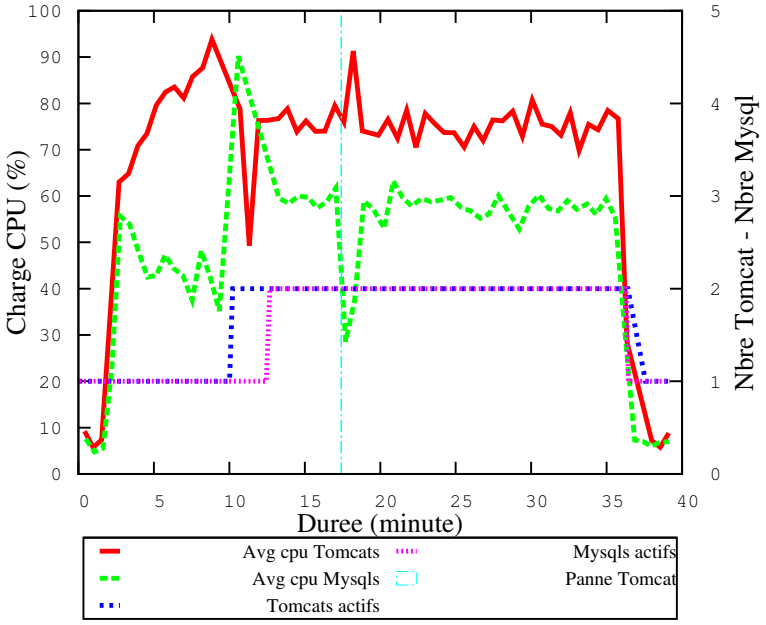

FIGURE 5.15 – Exécution coordonnée : Panne d'un serveur Tomcat

niveau du tier `Mysql` jusqu'en dessous du seuil minimal toléré. Mais sur ce tier aussi aucune opération de retrait de serveur est exécutée par l'instance de self-sizing qui le gère. La charge au niveau des tiers est redevenue normale après la réparation du serveur `Tomcat`.

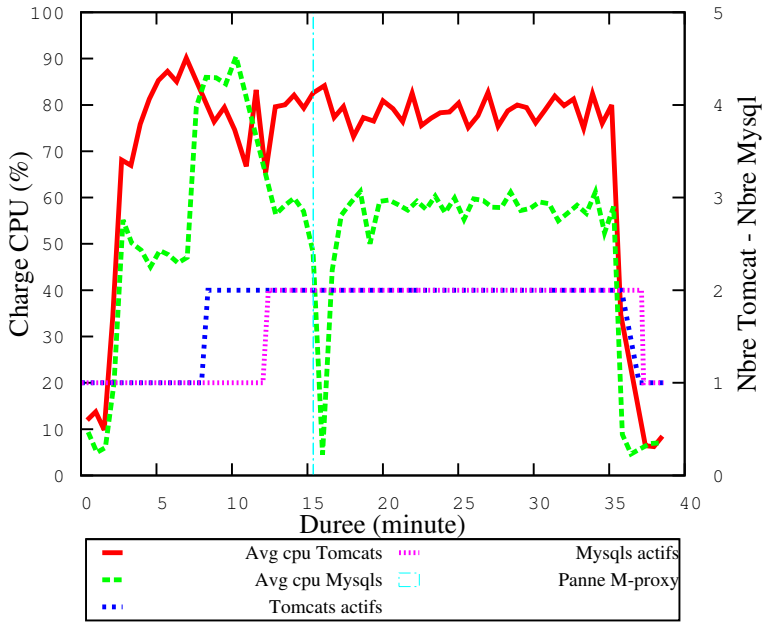

FIGURE 5.16 – Exécution coordonnée : Panne du serveur Mysql-proxy

La figure 5.16 présente une exécution durant laquelle on observe une panne du serveur `Mysql-proxy`. Cette panne survenue 16 minutes après le début de l'expérimentation a occasionné une baisse de charge au niveau du tier `Mysql`. Cependant aucune opération de retrait est effectuée sur le tier `Mysql` et la charge est redevenue normale après la réparation.

L'occurrence d'une panne de serveur `Mysql`, observée 17 minutes après le début de l'expérimentation sur la figure 5.17, occasionne une hausse de charge au niveau du `Mysql` restant. Cependant là également aucune opération d'ajout est observée. Le degré de réplication au niveau du tier `Mysql` n'a pas varié durant la réparation de la panne. La charge est redevenue normale après la réparation de la panne.

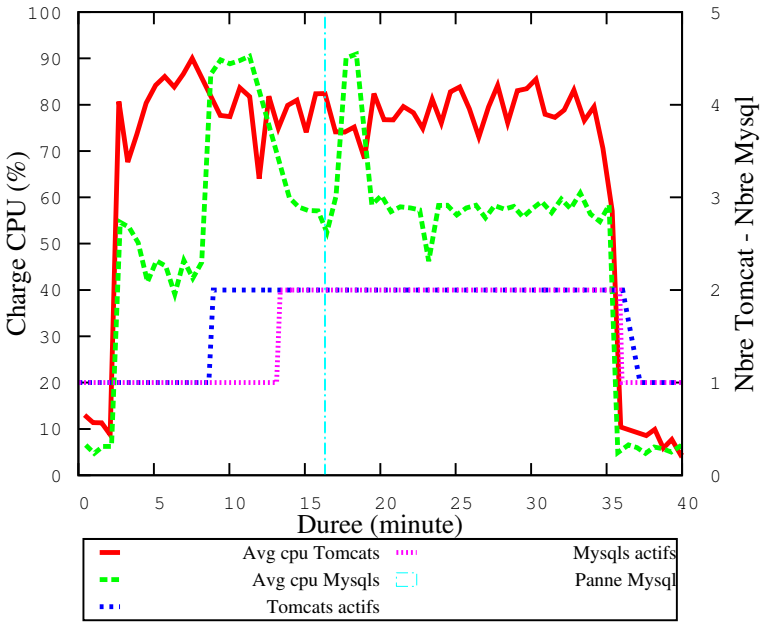

FIGURE 5.17 – Exécution coordonnée : Panne d'un serveur Mysql

5.5 Conclusion

Le contrôleur de coordination obtenu est en mesure de coordonner les gestionnaires afin d'assurer le respect de la politique de la coordination. Il empêche les gestionnaires self-sizing d'ajouter un nouveau serveur dans un tier où une panne est en cours de réparation. Il empêche également la suppression de serveurs au niveau des autres tiers. Le contrôleur permet d'éviter des opérations d'acquisition et de libération répétitives de machines. Cette propriété est particulièrement pertinente dans la gestion d'un centre de données, où les ressources de calcul sont partagées entre plusieurs applications clientes virtualisées.

Dans un centre de données, un quota de ressources de calcul est attribué à chacune des applications. Le quota affecté à une application peut être ajusté dynamiquement en fonction de sa charge de travail. Cependant, une acquisition inutile de ressources peut empêcher d'autres applications d'atteindre leurs objectifs de performance. Par ailleurs, la libération inutile de ressources nécessaires pour une application peut avoir un impact sur ses performances si les ressources sont affectées à d'autres applications.

6

Coordination modulaire pour la gestion d'applications multi-tiers et consolidation

Contents

6.1 Gestion des ressources d'un centre de données 100
6.1.1 Utilisation des ressources 100
6.1.2 Gestionnaire de consolidation de serveurs 101
6.2 Problèmes 101
6.3 Conception de la coordination modulaire 102
6.3.1 Modélisation des gestionnaires 103
6.3.1.1 Modélisation du gestionnaire self-sizing . . . 103
6.3.1.2 Modélisation du gestionnaire self-repair . . . 104
6.3.1.3 Modélisation du gestionnaire de consolidation 104
6.3.2 Spécification de la coordination 105
6.3.2.1 Stratégie de coordination 105
6.3.2.2 Spécification du contrat 106
6.3.2.3 Synthèse monolithique 106
6.3.2.4 Synthèse modulaire 107
6.3.2.5 Comparaison 111
6.4 Expérimentations 112
6.4.1 Configuration 112
6.4.2 Évaluation 113

6.5 Conclusion 115

L'objectif de cette étude de cas est de démontrer le passage à l'échelle de notre approche. Il présente une application de la coordination modulaire pour la gestion d'un centre de données. Nous modélisons la coordination modulaire des gestionnaires des applications hébergées dans le centre de données et du gestionnaire de consolidation.

Dans cette étude de cas, nous considérons que le centre de données héberge un ensemble d'applications de type multi-tiers JEE. Chacune des applications est gérée de manière autonome par deux instances de self-sizing et quatre instances de self-repair.

6.1 Gestion des ressources d'un centre de données

6.1.1 Utilisation des ressources

Les ressources disponibles dans un centre de données sont généralement virtualisées et partagées entre plusieurs applications. Les logiciels applicatifs sont exécutés dans des machines virtuelles. Un centre de données héberge des applications qui appartiennent souvent à des clients différents. Ces applications sont généralement soumises à des contraintes de qualité de service. Des contrats sont établis entre le propriétaire du centre de données et les clients. Dans ces contrats sont définis des requis de niveau de qualité de services que le propriétaire du centre de données doit garantir.

Les besoins en ressources des applications hébergées dans un centre de données varient généralement en fonction de leur charge de travail. Le dimensionnement statique des applications et un placement statique des machines virtuelles qui exécutent les applications peuvent mener à la non satisfaction des contraintes de qualité de service (SLAs) des applications ou à un gaspillage d'énergie. La puissance de calcul peut être dimensionnée en fonction des besoins en ressources des applications pour éviter un gaspillage d'énergie tout en garantissant leur performance.

6.1.2 Gestionnaire de consolidation de serveurs

Le gestionnaire de consolidation de serveurs est dédié à l'ajustement dynamique de la puissance de calcul fournie dans un centre de données virtualisé. La puissance de calcul repose sur un ensemble de serveurs physiques interconnectés. L'objectif de ce gestionnaire est d'éviter le gaspillage de ressources tout en satisfaisant les besoins des applications en ressource.

Le gestionnaire connaît, au moyen de sondes, la configuration courante du placement des machines virtuelles sur les serveurs physiques. Il connaît l'état de chaque machine virtuelle, l'utilisation des ressources qui leur sont affectées et la charge de chaque serveur physique. Périodiquement il évalue la capacité disponible et la capacité utilisée par les machines virtuelles. Il planifie une diminution de la capacité lorsque les serveurs physiques sont sous-utilisés, ou bien une augmentation de la capacité lorsque les applications requièrent plus de ressources qu'il n'y a de disponible.

Dans ce travail, nous utilisons **VMware DRS/DPM** [31,32] pour la gestion de la puissance de calcul dans un centre de données expérimental virtualisé et basé sur **VMware**. Ce gestionnaire peut planifier des actions de migration pour fournir plus de ressources aux machines virtuelles surchargées ce qui peut nécessiter le démarrage de serveurs physiques. Lorsque les serveurs physiques sont sous-utilisées, le gestionnaire peut également planifier des actions de migration de machines virtuelles afin d'arrêter certains serveurs. Ce gestionnaire peut être configuré de sorte à qu'il retourne le plan qu'il a généré. L'exécution du plan est à valider par l'administrateur du centre de données. L'exécution d'un plan peut être retardée. La contrôlabilité du gestionnaire est considéré ici seulement à gros grains : une intéressante perspective serait d'envisager un contrôle plus fin sur les opérations de consolidation exécutées séquentiellement. Cependant cela nécessite de déterminer des points de synchronisation appropriés.

6.2 Problèmes

L'exécution d'un plan de consolidation prend du temps. Son efficacité dépend essentiellement de la configuration courante de l'environnement vir-

tualisé durant l'opération. Lorsque des machines virtuelles sont instanciées ou supprimées pendant une opération de consolidation, il peut arriver des incohérences.

Lorsque le plan de consolidation consiste à diminuer la puissance de calcul disponible, des serveurs physiques vont être arrêtés et les machines virtuelles qu'ils hébergeaient vont être déplacées vers les serveurs actifs restants. Cela augmente la charge des serveurs restants et diminue les ressources disponibles sur ces serveurs. Si des machines virtuelles sont instanciées pendant l'exécution d'un plan de consolidation qui consiste à arrêter des serveurs, ces machines virtuelles peuvent être instanciées sur des serveurs qui vont être arrêtés ou sur ceux qui vont rester actifs. Cela conduit à la perte d'instances de machines virtuelles lorsqu'elles sont instanciées sur des serveurs arrêtés et peut mener à des échecs de migration ou d'instanciation de machines virtuelles si les serveurs restants ne disposent pas d'assez de ressources. Dans le cas d'un plan de consolidation qui consiste à augmenter la puissance de calcul, des serveurs physiques vont être redémarrés pour satisfaire les exigences des machines virtuelles. Cependant lorsque, durant l'exécution, des machines virtuelles sont arrêtées, les ressources fournies par les serveurs redémarrés peuvent ne plus être nécessaires. Cela conduit à du gaspillage de ressources.

La charge de travail des applications hébergées dans un centre de données varie tout au long de leur durée de vie. Cette variation influe sur les besoins en ressources des applications. L'utilisation des gestionnaires comme self-sizing et self-repair dans la gestion d'applications multi-tiers virtualisées peut mener à des instanciations et des suppressions de machines virtuelles. Ces opérations peuvent être entamées à tout moment. Par conséquent les incohérences décrites ci-dessus peuvent être constatées lorsque des instances de self-sizing et de self-repair coexistent avec un gestionnaire de consolidation. De plus entre les instances de self-sizing et self-repair d'une application multi-tiers il peut y avoir des incohérences (Section 5.2).

6.3 Conception de la coordination modulaire

Dans cette section, nous présentons les modèles des gestionnaires autonomes. Puis, nous détaillons la spécification du contrôle des gestionnaires de manière

modulaire et hiérarchique.

6.3.1 Modélisation des gestionnaires

Nous présentons des modèles des gestionnaires self-sizing et self-repair plus simples et différents de ceux utilisés dans les chapitres précédents. Cependant les mêmes propriétés de contrôle peuvent être appliquées et vérifiées sur ces modèles.

6.3.1.1 Modélisation du gestionnaire self-sizing

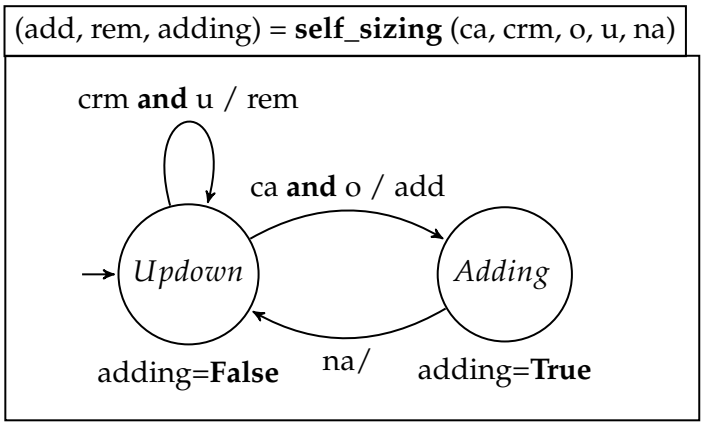

FIGURE 6.1 – Modèle du gestionnaire self-sizing

La figure 6.1 modélise le gestionnaire self-sizing. Initialement dans l'état `Updown`, le gestionnaire entame un retrait (`rem` à vrai) à l'occurrence d'une sous-charge (`u` à vrai) si l'action est autorisée (`crm` à vrai). Toujours dans l'état `Updown`, le gestionnaire entame un ajout de serveur (`add` à vrai) à l'occurrence d'une surcharge (`o` à vrai) et passe dans l'état `Adding` si l'opération d'ajout est autorisée (`ca` à vrai). L'état `Adding` indique l'exécution de l'ajout du serveur. Dans cet état le gestionnaire ne peut effectuer aucune autre opération. Il retourne à l'état `Updown` à la fin de l'ajout indiquée par `na` à vrai. L'état courant du gestionnaire est indiqué par la variable d'état `adding`.

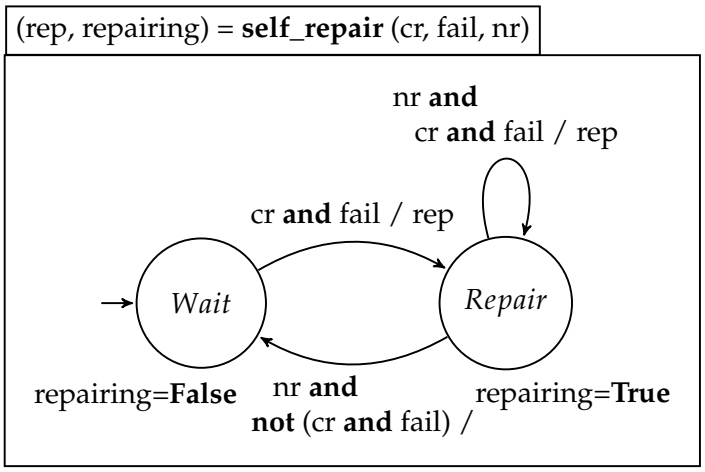

FIGURE 6.2 – Modèle du gestionnaire self-repair

6.3.1.2 Modélisation du gestionnaire self-repair

La figure 6.2 représente le modèle du gestionnaire self-repair. Initialement dans l'état `Wait`, le gestionnaire attend la détection d'une panne. A l'occurrence d'une panne (`fail` à vrai), le gestionnaire entame la réparation de la panne (`rep` à vrai) et passe dans l'état `Repair` si l'opération de réparation est autorisée (`cr` à vrai). L'état `Repair` indique l'exécution de la réparation du serveur tombé en panne. Dans cet état le gestionnaire ne peut effectuer aucune autre opération. Une fois l'opération terminée, `nr` à vrai, s'il y a une autre panne, le gestionnaire entame la réparation de la nouvelle panne si autorisée ; sinon il retourne dans l'état `Wait`.

6.3.1.3 Modélisation du gestionnaire de consolidation

La figure 6.3 présente le modèle du gestionnaire de consolidation. L'automate décrit le comportement et la contrôlabilité du gestionnaire. Dans l'état `Idle`, l'état initial, l'occurrence de l'événement `i` indique la nécessité d'augmenter la puissance de calcul. Le gestionnaire entame l'exécution du plan de consolidation (`si`) si l'action est autorisée (`ci` est à vrai), sinon il se met en attente d'autorisation (`WaitI`). Lorsque l'action est autorisée, le gestionnaire exécute le plan de consolidation (`I`). L'occurrence de l'événement `d` indique la nécessité de diminuer la puissance de calcul. Le gestionnaire entame l'exécution du plan de consolidation (`sd`) si l'action est autorisée (`cd` est à vrai), sinon

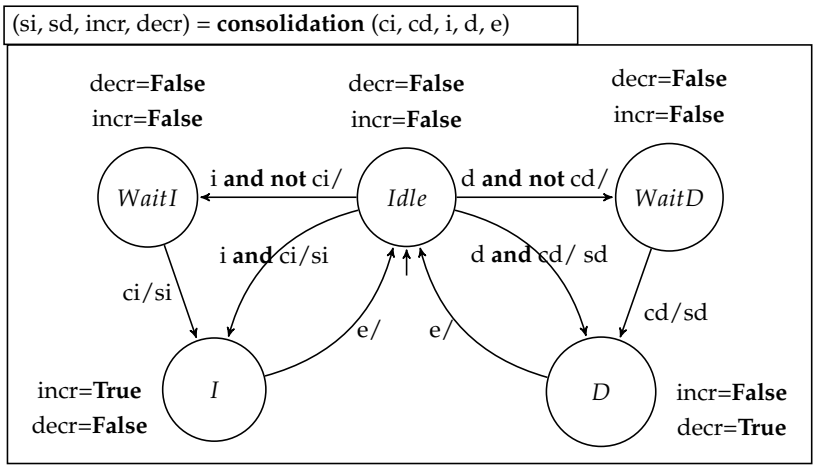

FIGURE 6.3 – Modèle du gestionnaire de consolidation

il se met en attente d'autorisation (`WaitD`). Lorsque l'action est autorisée, le gestionnaire exécute le plan de consolidation (`D`).

L'activité du gestionnaire est indiquée par les variables d'état `Incr` et `Decr`. `Incr` est à vrai lorsque le gestionnaire exécute un plan de consolidation pour augmenter la puissance de calcul. La variable `Decr` est, quant à elle, à vrai lorsque le gestionnaire exécute un plan de consolidation pour diminuer la puissance de calcul.

6.3.2 Spécification de la coordination

6.3.2.1 Stratégie de coordination

D'abord, nous définissons la stratégie de coordination des gestionnaires d'une application multi-tiers qui permet d'éviter une administration incohérente. Ensuite, nous définissons la stratégie de coordination des gestionnaires des applications et du gestionnaire de consolidation.

– Au niveau application multi-tiers :

1. Au niveau d'un tier dupliqué, inhiber les ajouts quand une réparation est en cours.
2. Dans un tier dupliqué avec aiguilleur de charge, inhiber les retraits quand l'aiguilleur «*load balancer*» est en cours de réparation.

3. De manière plus générale, dans une application multi-tiers, inhiber les retraits de serveurs dans un tier lorsqu'une réparation est en cours dans un des tiers précédents.

– Au niveau centre de données :

1. Eviter de faire des ajouts ou des réparations pendant une opération de consolidation.
2. Lorsque des instanciations de machines virtuelles (ajout/réparation) ou des suppressions (retrait) sont en cours, attendre avant d'exécuter un plan de consolidation.

6.3.2.2 Spécification du contrat

Les stratégies de coordination (section 6.3.2.1) sont décrites par les objectifs de contrôle ci-dessous :

– Au niveau application multi-tiers :

1. M1 : **not** (repairing **and** add)
2. M2 : **not** (repairingL **and** rem)
3. M3 : **not** (repairing$_{pred}$ **and** rem$_{succ}$)

– Au niveau centre de données :

1. DC1 : **not** ((Incr **or** Decr) **and** (repairing* **or** adding* **or** rem*))
2. DC2 :
   not ((repairing* **or** adding*) **and** sd) and **not** (rem* **and** si)

6.3.2.3 Synthèse monolithique

Nous avons effectué la spécification monolithique du contrôle afin d'évaluer les avantages de l'approche modulaire.

Avec la spécification monolithique, le contrôle est centralisé au niveau du modèle global, comme le montre la figure 6.4. Toutes les instances de modèles des gestionnaires impliqués sont groupées dans un seul modèle global. Tous les objectifs de contrôle sont définis sur le modèle global. De ce fait un seul contrôleur est construit pour assurer les objectifs locaux et les objectifs globaux. Cela peut être fastidieux et complexe quand plusieurs gestionnaires sont considérés. La structure de cette coordination est présentée à la figure 6.10.

$(\ldots) = \textbf{Main_node}\ (\ldots)$

enforce *all contracts*

with *all controllable variables*

$(\texttt{rep}_1, \texttt{repairing}_1) = \textbf{self_repair}\ (c'_1, \texttt{fail}_1, \texttt{nr}_1);$
...
$(\texttt{rep}_N, \texttt{repairing}_N) = \textbf{self_repair}\ (c'_N, \texttt{fail}_N, \texttt{nr}_N);$
$(\texttt{add}_1, \texttt{rem}_1, \texttt{adding}_1) = \textbf{self_sizing}\ (ca_1, \ldots);$
...
$(\texttt{add}_M, \texttt{rem}_M, \texttt{adding}_M) = \textbf{self_sizing}\ (ca_M, \ldots);$
$(\texttt{si}, \texttt{sd}, \texttt{Incr}, \texttt{Decr}) = \textbf{consolidation}\ (\texttt{ci}, \texttt{cd}, \texttt{i}, \texttt{d}, \texttt{e});$

FIGURE 6.4 – Synthèse monolithique

6.3.2.4 Synthèse modulaire

Avec l'approche modulaire, nous construisons un contrôle hiérarchique de bas en haut. Nous construisons des modèles simples dans lesquels les objectifs de "bas niveau" sont spécifiés et assurés par un contrôleur local. Puis nous réutilisons ces modèles de contrôle dans la spécification de contrôle dans laquelle les objectifs de plus "haut niveau" doivent être assurés en plus de ceux de "bas niveau".

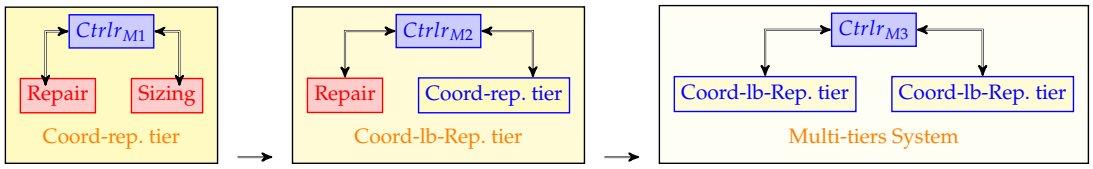

FIGURE 6.5 – Réutilisation de modèles de contrôle

La figure 6.5 présente la réutilisation de modèle dans le cas du contrôle d'une application multi-tiers. Nous construisons d'abord le modèle de contrôle d'une instance de self-sizing et d'une instance de self-repair qui gèrent un tier répliqué. Nous réutilisons ce modèle dans la construction du modèle de contrôle d'une instance de self-repair pour la gestion de l'équilibreur de charge placé en frontal d'un tier dupliqué, et l'instance de self-sizing et l'instance de self-repair qui gèrent le tier dupliqué. Le modèle de contrôle d'une application multi-tiers est construit en utilisant deux instances de ce modèle.

Tier dupliqué. Le modèle décrit à la figure 6.6 modélise la coordination d'une instance du gestionnaire self-sizing et d'une instance du gestionnaire self-repair. Ces instances de gestionnaires agissent sur le même tier dupliqué. Le contrat associé au modèle est constitué de quatre objectifs. L'un des objectifs correspond à la stratégie de coordination locale : (**not** (repairing **and** add)). Les autres objectifs correspondent à l'application du contrôle reçu de l'extérieur via les variables d'entrée cr′, ca′ et crm′.

```
(...) = coord_repl_tier (cr', fail, nr, ca', crm', o, u, na)
enforce (not (repairing and add))
        and LongActions(cr', rep, repairing)
        and LongActions(ca', add, adding)
        and (crm' or not rem)
with cr, ca, crm
    (rep, repairing)  = self_repair (cr, fail, nr);
    (add, rem, adding) = self_sizing (ca, crm, o, u, na);
```

FIGURE 6.6 – Tier dupliqué

Tier dupliqué avec aiguilleur en frontal. Le modèle décrit à la figure 6.7 modélise la coordination d'une instance de self-repair pour la gestion de l'équilibreur de charge placé en frontal d'un tier dupliqué, et l'instance de self-sizing et l'instance de self-repair qui gèrent le tier dupliqué. Le modèle est composé d'une instance du modèle du self-repair qui représente le gestionnaire chargé de la réparation de l'aiguilleur de charge, et d'une instance du modèle de la coordination décrite à la figure 6.6. Cette dernière est réutilisée pour la stratégie de coordination du self-sizing et du self-repair du tier dupliqué. Elle est composée avec une instance du modèle du self-repair qui représente le self-repair qui gère la réparation de l'équilibreur de charge. La stratégie de coordination locale à assurer est : **not** (repairingL **and** remove) et l'application du contrôle externe.

Multi-tiers. La figure 6.8 présente le modèle de coordination des gestionnaires qui gèrent les différents tiers. Ce modèle est composé de deux instances du modèle présenté à la figure 6.7.

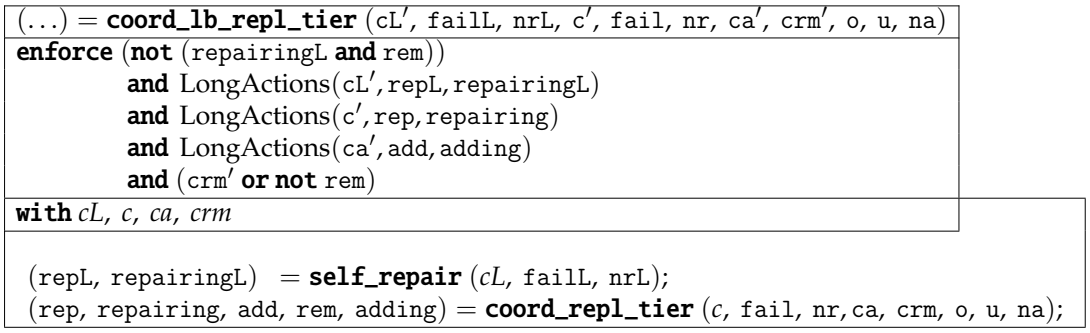

FIGURE 6.7 – Tier dupliqué avec aiguilleur en frontal

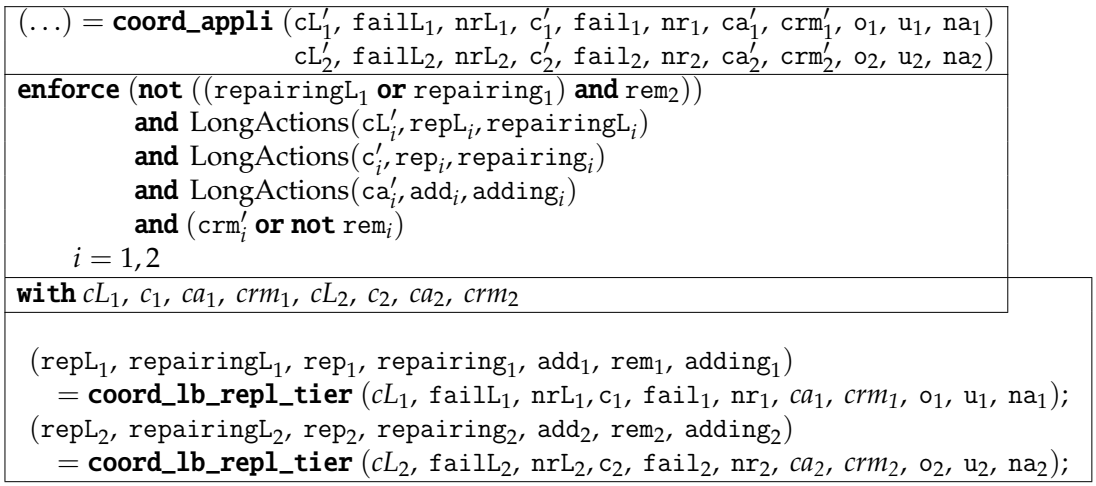

FIGURE 6.8 – Multi-tiers

La stratégie de coordination consiste à empêcher un retrait de serveurs. Ceci est exprimé comme suit : (**not** ((repairingL$_1$ **or** repairing$_1$) **and** rem$_2$))

Centre de données. La coordination de l'ensemble des gestionnaires présents dans un centre de données est construite progressivement de manière hiérarchique. La figure 6.9 présente le modèle de coordination des gestionnaires de deux applications multi-tiers et du gestionnaire de consolidation. Le modèle de contrôle est composé du modèle du gestionnaire de consolidation et de deux instances du modèle de contrôle des gestionnaires d'une application multi-tiers. A ce niveau le contrat ne contient que les objectifs qui concernent le centre de données. A partir de ce modèle, la coordination du gestionnaire de consolidation et des gestionnaires des autres applications peut être construite en

composant ce modèle avec une instance du modèle 6.8. Le modèle résultant sera composé avec une autre instance du modèle 6.8 et ainsi de suite. Cela permet une spécification hiérarchique du contrôle du gestionnaire de consolidation et des gestionnaires d'un nombre N d'applications multi-tiers.

Cependant cet exemple démontre le passage à l'échelle de l'approche modulaire sachant que les applications au niveau d'un centre de données sont dynamiques. Certaines peuvent être déployées ou arrêtées à tout moment au quel cas il faudrait envisager des ajouts et suppressions de modèles de gestionnaires au niveau du modèle global. Mais ce comportement n'est pas considéré dans le modèle de contrôle décrit dans ce travail.

$(\ldots)$ = **two_appli_in_data_center** $(\ldots)$

enforce (**not** ((Incr **or** Decr) **and** ($\texttt{repairing}_{ij}$ **or** $\texttt{adding}_{ij}$ **or** $\texttt{rem}_{ij}$))
and (**not** (($\texttt{repairing}_{ij}$ **or** $\texttt{adding}_{ij}$) **and** sd) **and not** ($\texttt{rem}_{ij}$ **and** si))
and LongActions($\texttt{cL}'_{ij}$, $\texttt{repL}_{ij}$, $\texttt{repairingL}_{ij}$)
and LongActions($\texttt{c}'_{ij}$, $\texttt{rep}_{ij}$, $\texttt{repairing}_{ij}$)
and LongActions($\texttt{ca}'_{ij}$, $\texttt{add}_{ij}$, $\texttt{adding}_{ij}$)
and ($\texttt{crm}'_{ij}$ **or not** $\texttt{rem}_{ij}$))
$i = 1,2; j = 1..2$

with cL_{11}, c_{11}, ..., crm_{22}, ci, cd

$(\ldots)$ = **coord_appli** (cL_{11}, c_{11}, ca_{11}, crm_{11}, ..., cL_{21}, c_{21}, ca_{21}, crm_{21}, ...)
$(\ldots)$ = **coord_appli** (cL_{12}, c_{12}, ca_{12}, crm_{12}, ..., cL_{22}, c_{22}, ca_{22}, crm_{22}, ...)
(si, sd, Incr, Decr) = **consolidation** (ci, cd, i, d, e);

FIGURE 6.9 – Centre de données avec deux applications multi-tiers.

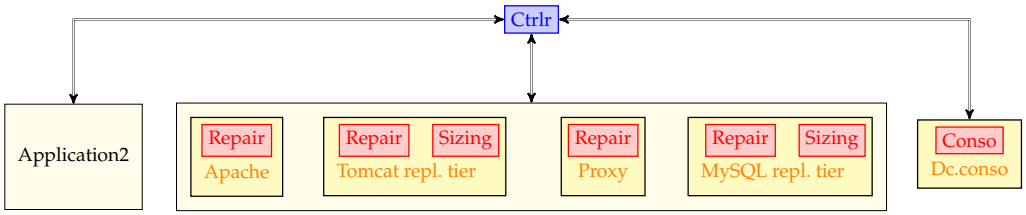

FIGURE 6.10 – Conception monolithique de la coordination

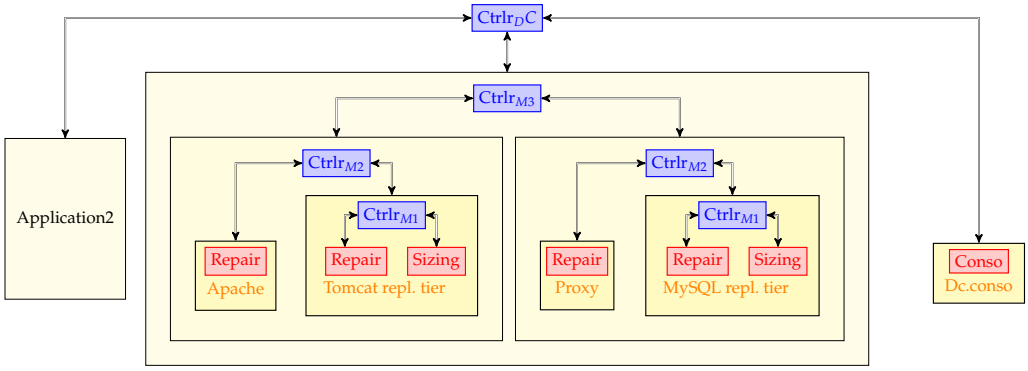

FIGURE 6.11 – Conception modulaire de la coordination

6.3.2.5 Comparaison

Les avantages de l'approche modulaire peuvent être considérés en terme de spécification et de coûts. L'approche modulaire permet une spécification décentralisée et hiérarchique du contrôle ; au lieu d'une spécification centralisée – le cas monolithique comme le montre la figure 6.10 – dans laquelle tous les automates sont d'un côté, et tous les contrats de l'autre côté. L'approche modulaire simplifie la spécification de contrôle d'un système large par la réutilisation de modèles dans des contextes différents. Cela améliore également la lisibilité de la spécification et facilite les modifications.

nb. app.	Durée de la synthèse		Utilisation de la mémoire	
	monolithique	modulaire	monolithique	modulaire
1	0s	5s	-	-
2	49s	11s	-	-
3	42m24s	24s	34.81MB	-
4	$>$ 2 days	1m22s	$>$149,56MB	-
5	-	4m30s	-	20,37MB
6	-	13m24s	-	53,31MB
7	-	25m57s	-	77,50MB
8	-	50m36s	-	115,59MB
9	-	2h11m	-	236,59MB
10	-	9h4m	-	479,15MB

TABLE 6.1 – SCD : durée de la synthèse et la mémoire utilisée

Nous avons effectué une étude comparative des approches monolithique et modulaire en ce qui concerne la complexité combinatoire de synthèse de contrôleur et les coûts (CPU et mémoire) de la compilation des contrôleurs pour différentes tailles de système (c'est à dire, un nombre variable d'applications). Les résultats présentés par le tableau 6.1 montrent que pour un petit nombre d'applications, l'utilisation de la mémoire n'est pas très significative, et la durée de la compilation est relativement acceptable. Cependant à partir de quatre applications, l'approche monolithique atteint les limites de l'explosion combinatoire des techniques d'exploration d'espace d'états. Le calcul n'est pas terminé après plus de deux jours. L'approche modulaire présente de meilleurs résultats même si les coûts continuent à croitre quand le nombre d'applications augmente. Nous constatons que l'approche monolithique est exponentiellement coûteuse, alors que l'approche modulaire continue à produire des résultats, montrant ainsi la possibilité d'adresser de larges systèmes.

Cependant, bien que le tableau 6.1 montre la durée totale de la compilation, en ce qui concerne la synthèse modulaire, la synthèse de la logique de contrôle de chaque contrat est effectuée de façon indépendante. Un modèle constitué d'un ensemble de modèles équipés d'un contrat ne requiert que la spécification du contrat défini dans les modèles pour la synthèse de la logique de contrôle. De ce fait la compilation des différents modèles peut être exécutée en parallèle ce qui réduit la durée totale de synthèse de l'ensemble des logiques de contrôle à intégrer dans les modèles. De plus, la recompilation d'un modèle est nécessaire uniquement lorsque son interface (entrées, sorties) et son contrat sont modifiés, ou bien ceux des sous-modèles qu'il utilise ; sinon, il peut être réutilisé tel quel.

6.4 Expérimentations

Nous avons réalisé des expérimentations pour le contrôle de deux applications dans notre centre de données expérimental.

6.4.1 Configuration

Notre centre de données expérimental est constitué d'un serveur de données et de six serveurs ESXi 5.1.0. Le serveur de données est muni de 24 CPU de

1.9Ghz, 63Go de mémoire et 12To de disque. Quatre des serveurs ESXi sont munis de 8 CPU de 2.4Ghz, 32Go de mémoire et 926Go de disque. Un serveur des serveurs ESXi est muni de 12 CPU de 2.4Ghz, 32Go de mémoire et 558Go de disque. Le dernier serveur ESXi est muni de 12 CPU de 1.89Ghz, 160Go de mémoire et 2.73To de disque. Nous avons utilisé des machines virtuelles munies de 1 vCPU, 2Go de mémoire et 8Go de disque.

6.4.2 Évaluation

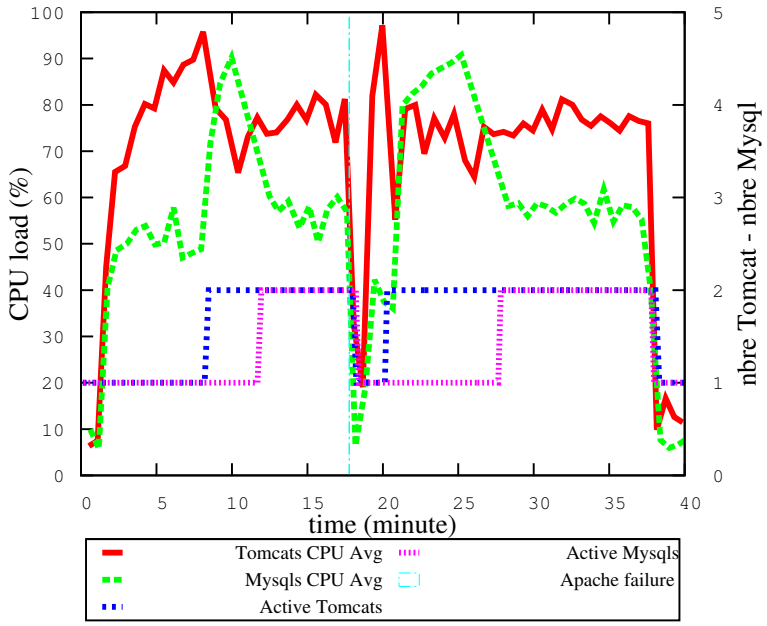

FIGURE 6.12 – Exécution non coordonnée : app 1 : Panne `Apache`

Les figures 6.12 et 6.13 présentent une exécution durant laquelle une panne s'est produite sur chaque application. Les gestionnaires des applications ne sont pas coordonnés. Sur la figure 6.12, il s'agit d'une panne du serveur `Apache` qui s'est produite 17 minutes après le début de l'expérimentation. La panne a conduit à un retrait de serveurs au niveau des tiers `Tomcat` et `Mysql` à cause d'une sous-charge. Cependant après la réparation du serveur `Apache`, le degré de réplication des tiers `Tomcat` et `Mysql` est restauré (21 min and 28 min).

Sur la figure 6.13, il s'agit d'une panne d'un serveur `Tomcat` produite 19 minutes après le début. Cette panne a conduit au retrait d'un serveur `Mysql` qui

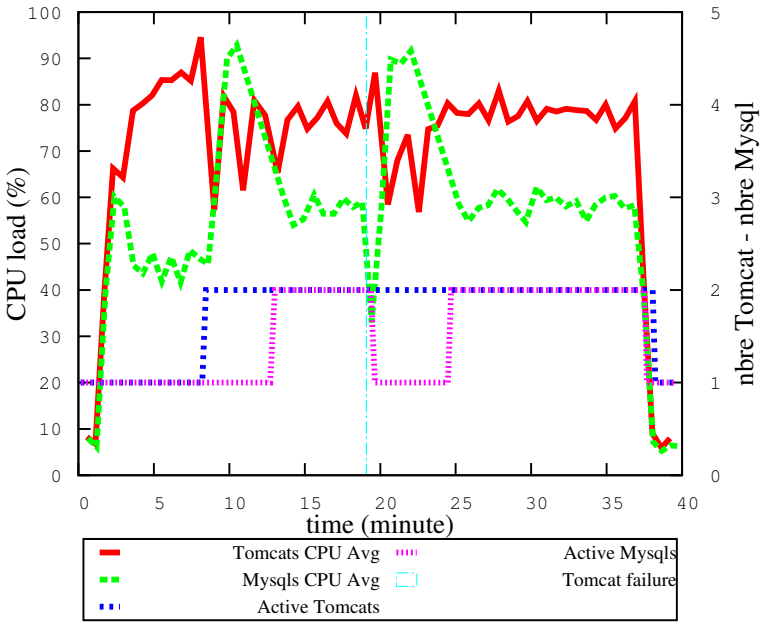

FIGURE 6.13 – Exécution non coordonnée : app 2 : Panne `Tomcat`

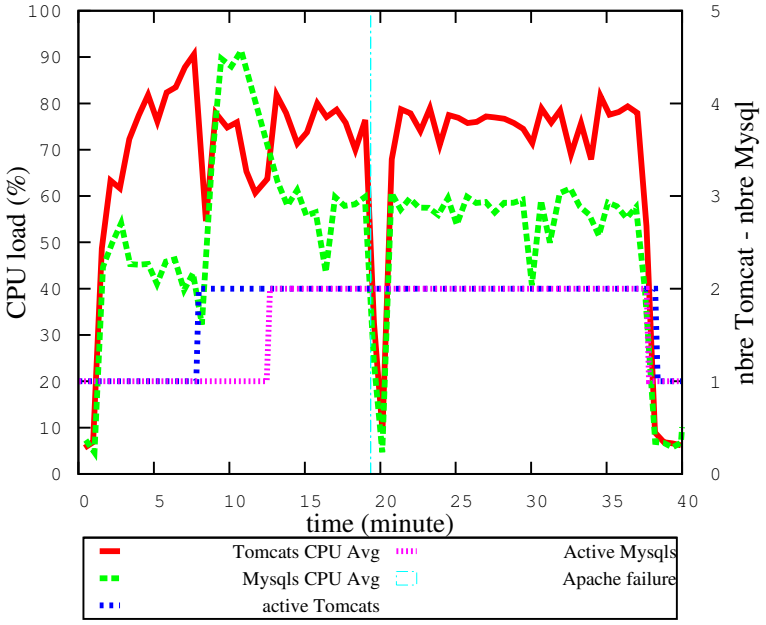

FIGURE 6.14 – Exécution coordonnée : app 1 : Panne `Apache`

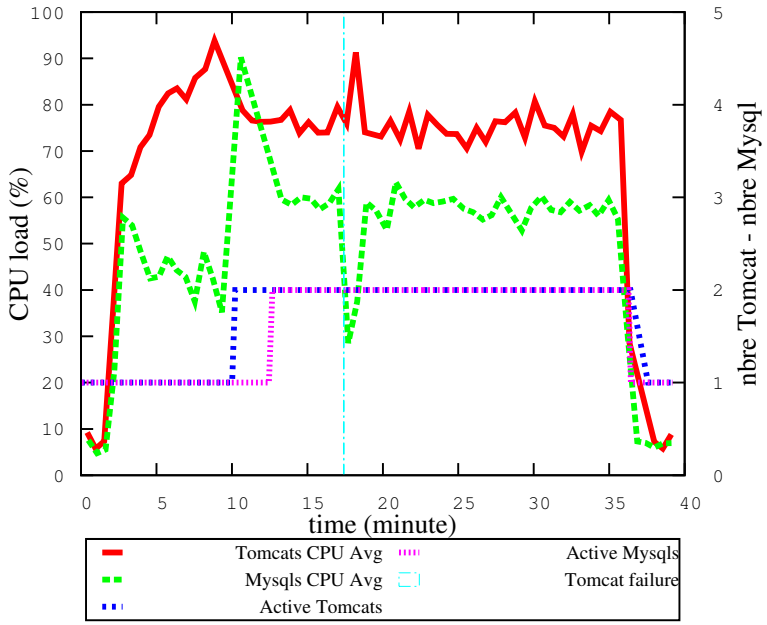

FIGURE 6.15 – Exécution coordonnée : app 2 : Panne `Tomcat`

a été rajouté après la réparation de la panne (25 min).

Les figures 6.14 et 6.15 présentent une exécution similaire durant laquelle les gestionnaires des deux applications sont coordonnées. Durant cette exécution, la panne du serveur `Apache`, 20 minutes après le début, conduit à une sous-charge au niveau des tiers `Tomcat` et `Mysql` ; mais aucun serveur n'est arrêté au niveau de ces tiers comme le montre la figure 6.14. De même, sur la figure 6.15, après la panne d'un serveur `Tomcat` survenue 17 minutes après le début, il y a une sous-charge au niveau du tier `Mysql` ; mais aucun serveur n'est arrêté. On remarque également que la charge moyenne des serveurs `Tomcat`, observée après la panne, excède le seuil maximal (90%) ; mais là aussi aucun serveur `Tomcat` n'est rajouté.

6.5 Conclusion

Les contrôleurs de coordination, obtenus avec la spécification modulaire, assurent le respect des objectifs de coordination définis. Chaque contrôleur,

dans la hiérarchie de contrôleurs, assure de manière cohérente le respect de la stratégie de coordination qu'il gère. Les contrôleurs de niveau supérieur assurent les objectifs globaux en appliquant un contrôle sur les contrôleurs sous-jacents.

Toutefois le code généré est structuré pour être exécuté de manière centralisée. Dans le chapitre suivant, nous étudions comment nous pouvons restructurer la hiérarchie de contrôleurs afin de pouvoir l'exécuter de manière distribuée.

7

Exécution distribuée des contrôleurs modulaires

Contents

7.1 Exécution distribuée de contrôleurs 119
7.1.1 Exécution distribuée synchronisée 119
7.1.1.1 Principe 120
7.1.1.2 Implémentation 120
7.1.2 Exécution distribuée désynchronisée 122
7.1.2.1 Principe 122
7.1.2.2 Implémentation 123
7.2 Exemple : Gestion d'une application multi-tiers 124
7.2.1 Exécution distribuée totalement synchronisée 124
7.2.1.1 Modélisation 124
7.2.1.2 Décomposition 125
7.2.2 Exécution distribuée partiellement synchronisée 126
7.2.2.1 Modélisation 126
7.2.2.2 Décomposition 129
7.2.3 Exécution distribuée désynchronisée 130
7.2.3.1 Modélisation 130
7.2.3.2 Décomposition 131
7.2.4 Comparaison 132
7.3 Expérimentation 133

7.3.1 Configuration 133
7.3.2 Évaluation 133
7.3.2.1 Durée de reconfiguration 133
7.3.2.2 Atteinte des objectifs de contrôle 134
7.4 Conclusion 140

La compilation d'une spécification modulaire produit une hiérarchie de contrôleurs. Chaque contrôleur est implémenté dans un programme distinct, e.g., une classe Java. Cependant, l'ensemble est structuré pour être exécuté de manière centralisée par défaut.

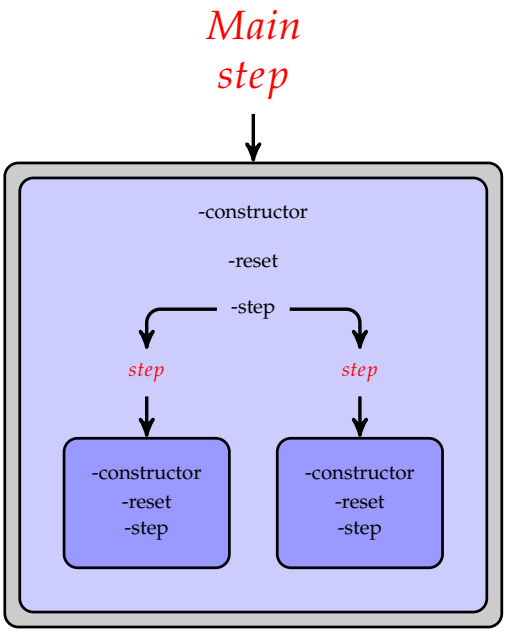

FIGURE 7.1 – Structure à l'exécution : Objet Java

A l'exécution, le contrôleur de plus haut niveau, contrôleur principal, instancie localement les contrôleurs sous-jacents sur lesquels il agit directement par appel de méthode. Ces derniers instancient ceux sur lesquels ils agissent et ainsi de suite. Cela est illustré à la figure 7.1. Toutefois, l'exécution centralisée présente des désavantages. Par exemple, en cas de défaillance, aucun contrôle, même partiel, ne peut être réalisé tant que la restauration n'est pas effectuée.

L'exécution distribuée des contrôleurs peut permettre d'éviter d'avoir un seul point potentiel de défaillance.

L'exécution distribuée d'une hiérarchie de contrôleurs peut permettre d'assurer un contrôle partiel en cas de défaillance. Lorsqu'un sous-ensemble des contrôleurs tombe en panne, les autres peuvent toujours assurer leurs objectifs locaux. De plus, lorsque le système à contrôler est large et réparti sur plusieurs sites, les contrôleurs peuvent être placés à proximité des sites qu'ils administrent. Cela réduit les délais de transmission des données de surveillance mesurées par les capteurs, ainsi que les délais de transmission des actions à appliquer par les actionneurs. Placer les contrôleurs à proximité des sites qu'ils administrent peut permettre de réduire les délais de réactivité par rapport à l'exécution centralisée.

Dans ce chapitre, nous allons étudier comment le code obtenu d'une spécification modulaire peut être restructuré pour l'exécuter de manière distribuée. Une exécution distribuée d'une hiérarchie de contrôleurs nécessite, au moins, que certains des contrôleurs soient instanciés de manière indépendante du contrôleur de haut niveau. Nous verrons dans ce chapitre trois approches d'exécution distribuée : totalement synchronisée, partiellement synchronisée, et désynchronisée.

7.1 Exécution distribuée de contrôleurs

Pour démontrer la possibilité d'exécuter de manière distribuée une hiérarchie de contrôleurs, nous supposons que le réseau est fiable. Nous supposons également qu'à l'exécution, ni les contrôleurs ni les machines qui les hébergent ne tombent en panne.

7.1.1 Exécution distribuée synchronisée

Comme le montre la figure 7.1, dans l'exécution centralisée, le contrôleur de plus haut niveau instancie les contrôleurs sous-jacents qu'il utilise pour assurer ses objectifs. Ces derniers sont locaux à celui-ci. Pour exécuter de manière distribuée une hiérarchie de contrôleurs, par exemple, les contrôleurs sous-

jacents au contrôleur de plus haut niveau (contrôleur principal) doivent être instanciés en dehors de ce dernier. Au niveau de celui-ci, cela nécessite la modification des appels locaux des méthodes *step* et *reset* des contrôleurs sous-jacents.

7.1.1.1 Principe

Les modifications à effectuer n'affectent pas l'implémentation des méthodes *step* et *reset* des contrôleurs. Par exemple, au niveau du contrôleur principal, les appels locaux des méthodes *step* et *reset* des sous-contrôleurs doivent simplement être remplacés par des appels distants. L'appel local de la méthode *reset* des sous-contrôleurs dans le contrôleur principal est implémenté dans la méthode *reset* de ce dernier. Étant donné que l'appel d'une méthode *reset* ne retourne pas de résultat, pour l'exécution distribuée, il suffit simplement de remplacer le code qui implémente l'appel local par celui qui implémente l'appel distant. Le même procédé est utilisé pour la modification de l'appel de la méthode *step* des sous-contrôleurs à exécuter de manière distribuée. L'appel d'une méthode *step* retourne un résultat. Toutefois, une structure de données est définie pour contenir le résultat. De ce fait cette même structure peut être utilisée sans modification.

Le code qui instancie localement les sous-contrôleurs dans le contrôleur principal doit également être remplacé par le code qui permet d'obtenir les références de ces derniers exécutés à distance. Ces références vont permettre de faire les appels distants. L'implémentation de l'exécution distribuée peut être effectuée de différentes manières. L'approche *Message Queuing* peut être utilisée, ou bien *Java-rmi*.

7.1.1.2 Implémentation

Nous utilisons *Java-rmi* pour montrer comment restructurer le code. Avec *Java-rmi* les modifications à effectuer sont simples puisque les objets instances locales de contrôleurs sont simplement transformés en références d'instances de contrôleurs distants. Ces références sont manipulées (appel des méthodes) de la même manière que des objets locaux.

La figure 7.2 représente la transformation du schéma d'exécution centralisée

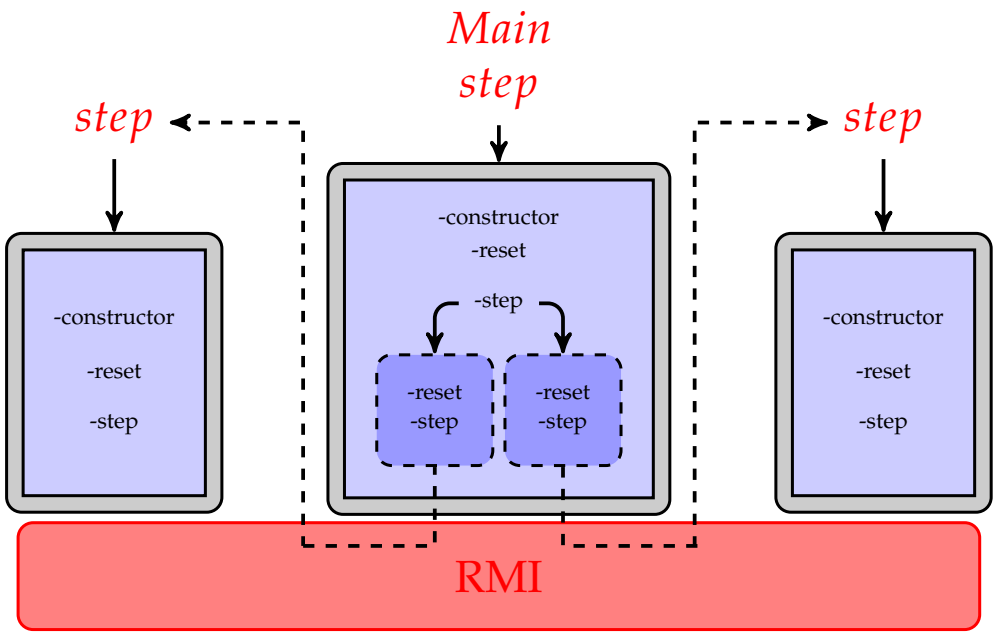

FIGURE 7.2 – Exécution distribuée synchronisée avec `Java rmi`

présenté dans la figure 7.1 pour une exécution distribuée synchronisée. Les trois contrôleurs sont exécutés sont trois sites distincts. Le contrôleur principal, au centre, ne manipule pas des objets instances de contrôleurs. Il manipule des objets références d'instances de contrôleurs distants. C'est à travers ces objets références qu'il fait l'appel des méthodes des instances des sous-contrôleurs.

Pour implémenter l'exécution distribuée avec *Java-rmi*, il faut définir une interface de type *RemoteInterface* pour chaque contrôleur. L'interface d'un contrôleur contient au moins la signature des méthodes *step* et *reset* accessibles à distance. L'implémentation des interfaces n'implique aucune modification des méthodes des contrôleurs. De plus, il faut enregistrer les objets instances des contrôleurs dans le *rmiregistry*. L'implémentation en Java d'un contrôleur contient un *constructeur* et les méthodes *step* et *reset*. Le code qui permet l'enregistrement d'un contrôleur dans le *rmiregistry* peut être ajouté dans le *constructeur*. Au niveau du contrôleur principal, par exemple, les objets instances des contrôleurs sous-jacents sont déclarés comme attributs et instanciés dans son *constructeur*. Ici, le type de chaque objet instance d'un contrôleur doit être remplacé par le type de l'interface du contrôleur. De plus, le code qui instancie les sous-contrôleurs doit être remplacé par le code qui récupère, à travers

rmiregistry, les références des contrôleurs distants correspondants. Avec *Java-rmi*, l'appel des méthodes *step* (resp. *reset*) des contrôleurs distants ne changent pas puisque les objets références sont manipulés comme des objets locaux. La communication distribuée est gérée par *Java-rmi*.

7.1.2 Exécution distribuée désynchronisée

Une spécification modulaire peut être adaptée pour obtenir un ensemble de contrôleurs à exécuter de manière distribuée désynchronisée. Dans ce cas, il n'y a pas de contrôleur principal pour assurer la synchronisation et pour orchestrer le contrôle global. Chaque contrôleur distant évolue à son rythme et reçoit les entrées concernant le sous-ensemble qu'il gère. Les contrôleurs distants assurent le contrôle global par échange de valeurs de contrôle de manière asynchrone. L'avantage de cette approche est le fait qu'il n'y ait aucune synchronisation entre les contrôleurs ce qui accroît leur réactivité.

7.1.2.1 Principe

Généralement une spécification modulaire produit une hiérarchie de contrôleurs avec un contrôleur principal qui assure le contrat global. Chacun des contrôleurs dans la hiérarchie assure un contrat local au sous-système qu'il contrôle. Ici, il s'agit de supprimer le contrôleur principal et de conserver les contrôleurs sous-jacents. Ces derniers vont assurer le contrat global par échange de valeurs de contrôle de manière asynchrone. La communication asynchrone est modélisée pour décrire les propriétés pertinentes du système de communication. Cela permettra, à la compilation, de vérifier si le contrat global est respecté en considérant les propriétés du système de communication. Cette méthode permet la modélisation d'un système localement synchrone et globalement asynchrone (GALS). Les étapes de conception sont décrites ci-dessous :

1. Nous modélisons le contrôleur de chaque sous-système. Cela correspond au modèle du sous-système auquel on associe un contrat contenant les objectifs de contrôle.
2. Nous modélisons ensuite le système de communication asynchrone utilisé pour la transmission de valeurs de contrôle.

3. Ces différents modèles sont composés pour représenter le système global. Nous déclarons le contrat global, avec les objectifs globaux mais sans variable contrôlable. A la compilation du modèle global, le contrat est vérifié par *model-checking*. Aucun contrôleur n'est construit pour le modèle global.

4. Pour l'exécution, le code correspondant au modèle global, ainsi que le code correspondant au modèle du système de communication ne sont pas utilisés. Le code des contrôleurs contenus dans le modèle global est utilisé, ainsi que le système réel pour la communication pour la mise en oeuvre des échanges de valeurs de contrôle.

7.1.2.2 Implémentation

L'implémentation de l'exécution distribuée désynchronisée n'implique aucune modification au niveau du code des contrôleurs. Ces derniers n'ont aucune référence locale les uns des autres. Un contrôleur reçoit les valeurs de contrôle d'un autre contrôleur comme entrées de sa méthode *step*. De plus, les valeurs de contrôle qu'il transmet sont contenues dans le résultat retourné par sa méthode *step*. De ce fait l'ajout du code pour la publication et/ou la réception des valeurs de contrôle à échanger ne nécessite pas de modification de l'implémentation des contrôleurs. Ce code peut même être placé ailleurs, par exemple dans le code qui fait l'appel de la méthode *step*. Nous utilisons l'approche *Message Queuing* pour montrer comment implémenter une exécution distribuée désynchronisée.

La figure 7.3 représente la transformation du schéma d'exécution centralisée présenté dans la figure 7.1 pour une exécution distribuée désynchronisée. Ici, le contrôle principal n'existe plus et il n'y a aucune synchronisation des *step* des contrôleurs. Ces derniers sont indépendants et chacun évolue à son propre rythme dicté par les événements survenus dans le sous-système qu'il gère. Pour l'échange de message, nous utilisons un type de serveur de messages, *RabbitMQ*. Ce type de serveur permet de définir des supports d'échange, *Topic*, où des messages sont publiés. L'abonnement à un *Topic* permet de recevoir les messages publiés dedans. Chaque contrôleur publie ses valeurs de contrôle et s'abonne pour recevoir les valeurs de contrôle de son vis-à-vis, dans l'exemple présenté à la figure 7.3.

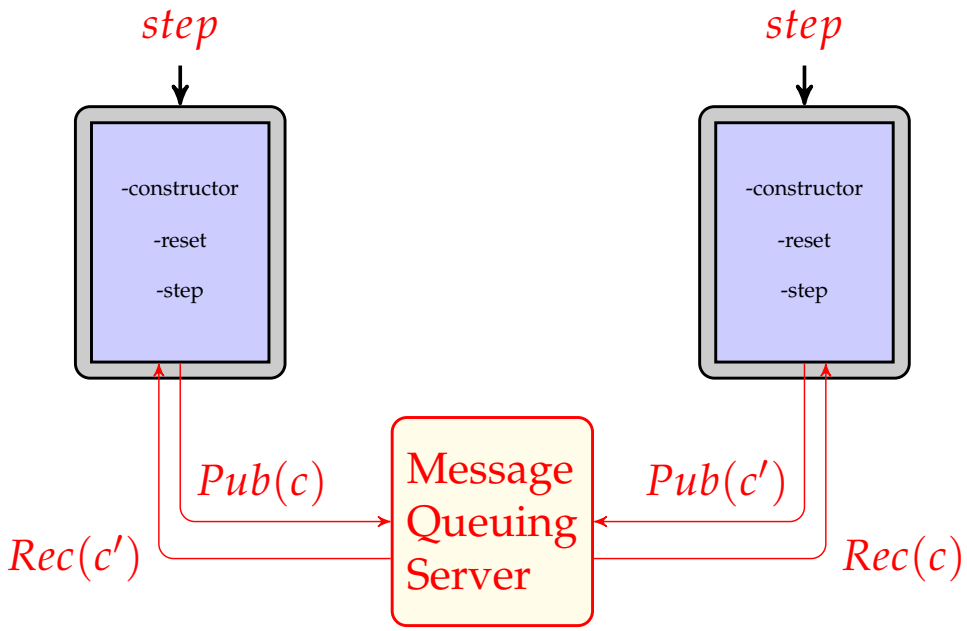

FIGURE 7.3 – Exécution distribuée désynchronisée avec *Message Queuing*

7.2 Exemple : Gestion d'une application multi-tiers

Cette section présente une application des approches pour l'exécution distribuée synchronisée et l'exécution distribuée désynchronisée. Nous considérons l'exemple de coordination modulaire des gestionnaires de l'application multi-tiers présentée dans le chapitre précédent. La conception de la coordination modulaire est présentée à la section 6.3.2.4. Ci-dessous nous présentons la transformation à faire sur le code obtenu pour l'exécuter de manière distribuée.

7.2.1 Exécution distribuée totalement synchronisée

Pour l'exécution distribuée totalement synchronisée, la spécification modulaire présentée à la section 6.3.2.4 ne nécessite aucune modification. Elle est réutilisée telle quelle.

7.2.1.1 Modélisation

Nous réutilisons la spécification modulaire de la coordination des gestionnaires self-sizing et self-repair décrite à la section 6.3.2.4 telle quelle. La figure 7.4

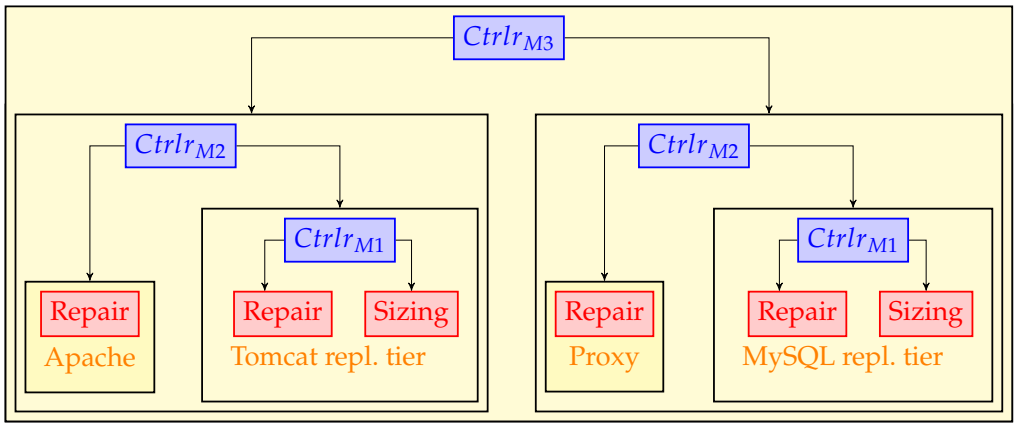

FIGURE 7.4 – Spécification modulaire

montre la structure de la spécification modulaire.

7.2.1.2 Décomposition

La compilation du modèle décrit à la figure 7.4 produit une hiérarchie de contrôleurs : **Ctrlr**$_{M1}$, **Ctrlr**$_{M2}$ et **Ctrlr**$_{M3}$. A l'exécution centralisée, le contrôleur principal **Ctrlr**$_{M3}$ instancie les contrôleurs **Ctrlr**$_{M2}$. Chaque contrôleur **Ctrlr**$_{M2}$ instancie un contrôleur **Ctrlr**$_{M1}$.

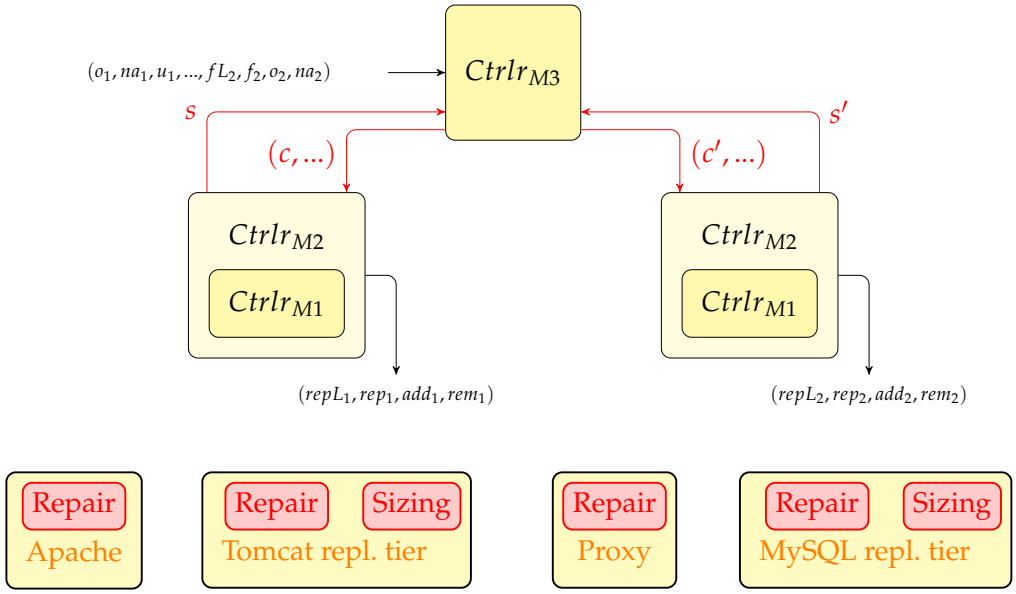

FIGURE 7.5 – Exécution distribuée totalement synchronisée

Ici, nous décomposons le code en trois parties à exécuter sur des machines différentes comme le montre la figure 7.5. Les contrôleurs $\mathbf{Ctrlr}_{M1}$ sont locaux aux contrôleurs $\mathbf{Ctrlr}_{M2}$. Sur cet exemple, l'exécution distribuée obtenue est totalement synchronisée. En effet toutes les entrées sont reçues par le contrôleur $\mathbf{Ctrlr}_{M3}$. Ce dernier, distribue les entrées reçues aux autres contrôleurs avec ses restrictions pour respecter son contrat. Une fois qu'il a calculé les valeurs de contrôle, il appelle la méthode *step* des contrôleurs $\mathbf{Ctrlr}_{M2}$ distants. Ces derniers vont, eux aussi, appliquer des restrictions pour atteindre leurs objectifs. L'appel des méthodes *step* est bloquant et totalement synchronisé.

7.2.2 Exécution distribuée partiellement synchronisée

Pour obtenir une exécution distribuée partiellement synchronisée, il faut distinguer les entrées nécessaires au contrôleur principal $\mathbf{Ctrlr}_{M3}$ pour réaliser son contrôle. Cela permet de ne transmettre à ce dernier que les entrées dont il a besoin. Les autres entrées seront directement transmises aux contrôleurs $\mathbf{Ctrlr}_{M2}$. De manière générale, il faut distinguer pour chaque entrée, le contrôleur de haut niveau qui doit la traiter en premier.

7.2.2.1 Modélisation

Pour une exécution partiellement synchronisée, les entrées qui ne sont pas nécessaires au contrôleur de haut niveau, par exemple $\mathbf{Ctrlr}_{M3}$, sont directement transmises aux contrôleurs qui les traitent, par exemple $\mathbf{Ctrlr}_{M2}$. Toutefois les contrats associés aux contrôleurs ne changent pas. Le code obtenu de la spécification décrite à la figure 7.4 peut être adapté pour une exécution distribuée partiellement synchronisée. Une approche est d'affecter une valeur par défaut aux entrées que le contrôle principal ne traitent pas lors de l'appel de sa méthode *step*. Les vraies valeurs de ces entrées seront transmises aux contrôleurs $\mathbf{Ctrlr}_{M2}$ qui les traitent.

Une autre approche est de modifier la spécification des contrôleurs pour n'avoir que les entrées de contrôle comme arguments de leur méthode *step*. Les autres entrées sont récupérées dans la méthode *step* via des appels de fonctions externes. Le langage `Heptagon/BZR`, tout comme les autres langages synchrones, permet d'appeler des fonctions externes dans la définition d'un

automate. Pour utiliser les fonctions externes, nous définissons un module qui fournit les signatures de ces dernières. Dans la suite, nous utilisons le terme «**System**» comme nom du module.

Tier dupliqué.

```
(...) = coord_repl_tier (cr', ca', crm')
enforce (not (repairing and add))
        and LongActions(cr', rep, repairing)
        and LongActions(ca', add, adding)
        and (crm' or not rem)
with cr, ca, crm
 fail = System.failure ();
 nr = System.repaired ();
 o = System.overload ();
 na = System.added ();
 u = System.underload ();
 (rep, repairing)  = self_repair (cr, fail, nr);
 (add, rem, adding) = self_sizing (ca, crm, o, u, na);
```

FIGURE 7.6 – Tier dupliqué

La figure 7.6 correspond au modèle de coordination d'un gestionnaire self-sizing et un gestionnaire self-repair pour un tier dupliqué. Ici les événements en entrée sont récupérées via les fonctions externes « *System.** ». Ces dernières sont implémentées en dehors du modèle. Chaque fonction externe retourne un **booléen** qui correspond à la valeur courante de l'événement qu'elle traite. Par exemple *System.overload* retourne `vrai` lorsqu'une sur-charge est détectée, sinon elle retourne `faux`. Cependant ce modèle expose les entrées de contrôle qui permettent sa réutilisation et d'inhiber les actions des gestionnaires.

Tier dupliqué avec aiguilleur de charge en frontal.

La figure 7.7 correspond au modèle de coordination de deux gestionnaires self-repair et un gestionnaire self-sizing pour un tier dupliqué avec un aiguilleur en frontal. Le modèle dans la figure 7.6 est réutilisé pour la construction de la hiérarchie. Toutefois, ici toutes les entrées ne sont pas transmises au modèle dans la figure 7.7.

```
(...) = coord_lb_repl_tier (cL', c', ca', crm')
enforce (not (repairingL and rem))
        and LongActions(cL', repL, repairingL)
        and LongActions(c', rep, repairing)
        and LongActions(ca', add, adding)
        and (crm' or not rem)
with cL, c, ca, crm

 failL = System.failureLb ();
 nrL = System.repairedLb ();
 (repL, repairingL)  = self_repair (cL, failL, nrL);
```

FIGURE 7.7 – Tier dupliqué avec aiguilleur en frontal

Multi-tiers.

```
(...) = coord_multitier (cL'_1, c'_1, ca'_1, crm'_1, cL'_2, c'_2, ca'_2, crm'_2)
enforce (not ((repairingL_1 or repairing_1) and rem_2))
        and LongActions(cL'_i, repL_i, repairingL_i)
        and LongActions(c'_i, rep_i, repairing_i)
        and LongActions(ca'_i, add_i, adding_i)
        and (crm'_i or not rem_i)
    i = 1,2
with cL_1, c_1, ca_1, crm_1, cL_2, c_2, ca_2, crm_2

  (repL_1, repairingL_1, rep_1, repairing_1, add_1, rem_1, adding_1)
    = coord_lb_repl_tier (cL_1, c_1, ca_1, crm_1);
  (repL_2, repairingL_2, rep_2, repairing_2, add_2, rem_2, adding_2)
        = coord_lb_repl_tier (cL_2, c_2, ca_2, crm_2);
```

FIGURE 7.8 – Multi-tiers

La figure 7.8 correspond au modèle de coordination de l'application multi-tiers. Ce dernier modèle ne reçoit aucun événement. Ses objectifs de contrôle sont définis sur les sorties des contrôleurs sous-jacents.

Dans cet exemple, seules les entrées de contrôle et les sorties sont utilisées pour exprimer les objectifs. De ce fait, toutes les autres entrées peuvent être récupérées via des appels de fonctions dans les méthodes *step*. Cependant, lorsque ces dernières sont utiles dans la déclaration des objectifs, il est nécessaire qu'elles soient reçues comme entrées de la méthode *step* pour pouvoir appliquer

la synthèse de contrôleur.

Dans tous les cas, il faut distinguer les entrées qui nécessitent l'appel de la méthode *step*, ici, du contrôleur $\mathbf{Ctrlr}_{M3}$. Dans cet exemple, ces entrées sont : $\texttt{failL}_1$ (panne `Apache`), $\texttt{nrL}_1$ (`Apache` réparé), $\texttt{failL}_1$ (panne `Tomcat`), $\texttt{nr}_1$ (`Tomcat` réparé), $\texttt{u}_2$ (sous-charge `Mysql`).

7.2.2.2 Décomposition

La compilation donne la même hiérarchie de contrôleurs que celle obtenue avec la coordination totalement synchronisée. Chacun des contrôleurs assure le respect des mêmes objectifs que son vis-à-vis dans la coordination totalement synchronisée.

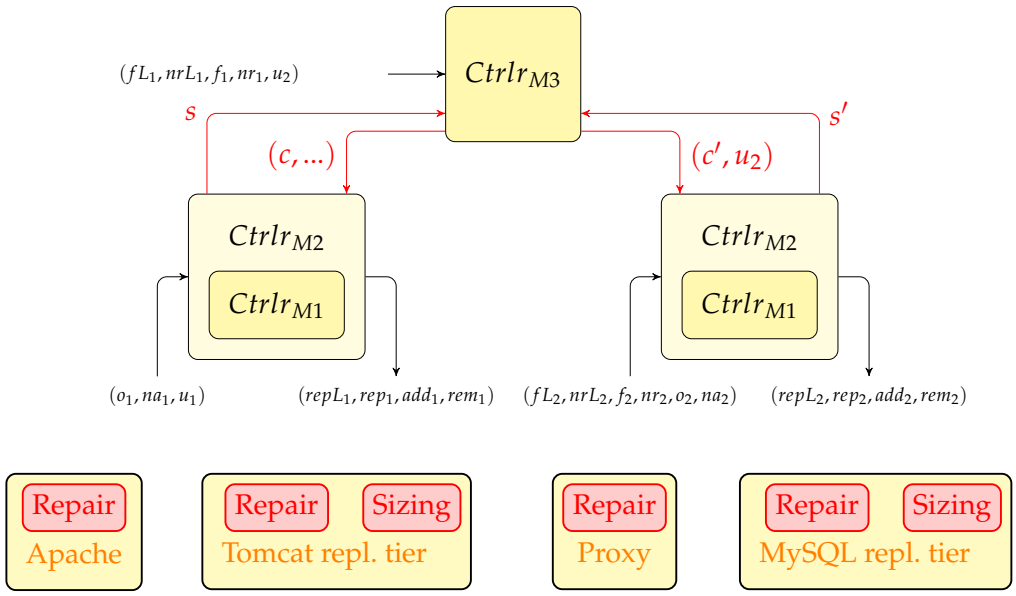

FIGURE 7.9 – Exécution distribuée partiellement synchronisée

Cependant, comme le montre la figure 7.9, le contrôleur principal $\mathbf{Ctrlr}_{M3}$ et les deux $\mathbf{Ctrlr}_{M2}$ reçoivent, chacun, une partie des entrées. De ce fait, les *step* sont partiellement synchronisés. Le contrôleur $\mathbf{Ctrlr}_{M3}$ n'initie la synchronisation que pour le respect de l'objectif global. Les entrées qui ne concernent pas le contrat du contrôleur $\mathbf{Ctrlr}_{M3}$ sont transmises directement aux contrôleurs $\mathbf{Ctrlr}_{M2}$ qui les traitent. Chaque contrôleur $\mathbf{Ctrlr}_{M2}$ évolue à son propre rythme en fonction de l'occurrence des entrées qu'il traite sans l'intervention du

contrôleur $\mathbf{Ctrlr}_{M3}$. Ces entrées ($\texttt{failL}_1$, $\texttt{nrL}_1$, $\texttt{fail}_1$, $\texttt{nr}_1$, $\texttt{u}_2$) sont transmises au contrôleur $\mathbf{Ctrlr}_{M3}$ qui calcule ses valeurs de contrôle. Ensuite, il appelle les méthodes *step* des contrôleurs $\mathbf{Ctrlr}_{M2}$, avec ses valeurs de contrôle. Ces derniers conservent les valeurs de contrôle reçues du contrôleur principal jusqu'à la prochaine synchronisation.

7.2.3 Exécution distribuée désynchronisée

Pour l'exécution distribuée désynchronisée, le contrat qui est défini dans le modèle global est assuré par les contrôleurs $\mathbf{Ctrlr}_{M2}$. Dans la coordination modulaire synchronisée, ce contrat est assuré par le contrôleur $\mathbf{Ctrlr}_{M3}$.

7.2.3.1 Modélisation

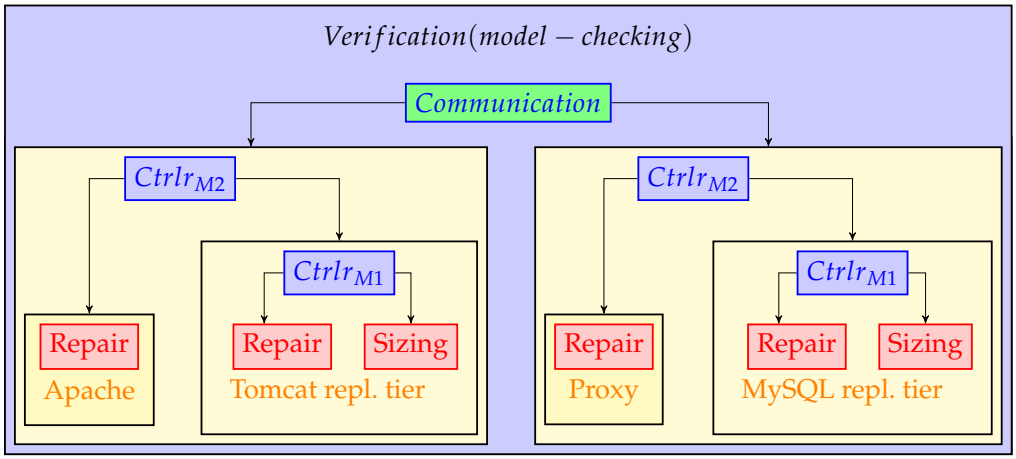

FIGURE 7.10 – Coordination modulaire désynchronisée

La figure 7.10 présente la structure de la spécification de la coordination modulaire désynchronisée. Dans cet exemple seul le modèle global change. Le modèle de communication asynchrone entre les deux sous-ensembles est ajouté. Les contrats définis pour les contrôleurs $\mathbf{Ctrlr}_{M2}$ (et $\mathbf{Ctrlr}_{M1}$) ne changent pas. Le contrat défini dans le modèle global et qui était assuré par le contrôleur $\mathbf{Ctrlr}_{M3}$ dans la spécification modulaire initiale, ici est assuré par programmation (échange de valeurs de contrôle).

7.2.3.2 Décomposition

Le contrôleur **Ctrlr**$_{M3}$ disparaît. Chaque contrôleur **Ctrlr**$_{M2}$ reçoit toutes les entrées concernant les gestionnaires qu'il contrôle. Dans cet exemple, le contrôleur **Ctrlr**$_{M2}$ qui gère les gestionnaires des tiers `Apache` et `Tomcat` envoie des valeurs de contrôle au contrôleur **Ctrlr**$_{M2}$ des gestionnaires des tiers `Mysql-Proxy` et `Mysql`. Ces valeurs de contrôle permettent d'inhiber les actions de retrait de serveurs `Mysql` en cas de panne au niveau `Apache` et/ou `Tomcat`. Ici la valeur transmise est :

- `c = not (ap_repairingLb or tom_repairing)`

  `ap_repairingLb` est à vrai lorsque le serveur `Apache` est en cours de réparation et `tom_repairing` est à vrai lorsqu'un serveur `Tomcat` est en cours de réparation.
- `c` est à vrai que lorsqu'une panne est en cours de réparation, auquel cas il faut inhiber les actions de retrait de serveurs `Mysql` tant que la réparation n'est pas terminée.

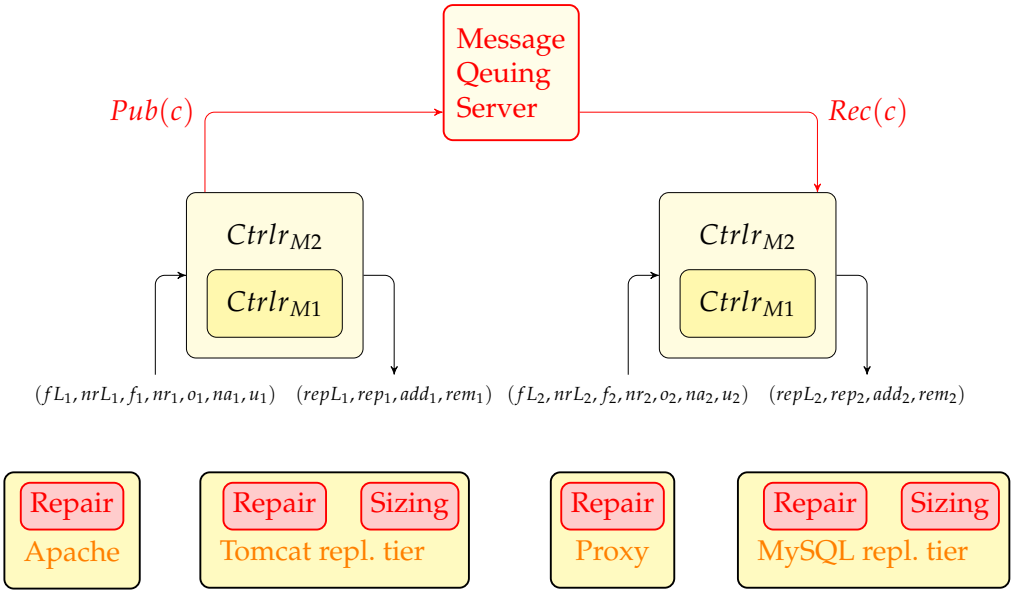

FIGURE 7.11 – Exécution distribuée désynchronisée

Nous utilisons un serveur de messages, *RabbitMQ*, pour mettre en oeuvre la communication entre les contrôleurs.

7.2.4 Comparaison

Durant l'exécution distribuée totalement synchronisée, comme pour l'exécution centralisée, tous les contrôleurs évoluent au même rythme que le contrôleur principal. Ce dernier orchestre tout le contrôle. Il reçoit toutes les entrées puis les transmet aux autres contrôleurs avec les valeurs de contrôle pour respecter ses objectifs. Cependant l'exécution distribuée peut ajouter un délai de réactivité supplémentaire à cause de la communication à distance. Durant l'exécution distribuée partiellement synchronisée, le contrôleur principal orchestre uniquement le contrôle pour assurer ses objectifs. Il ne reçoit que les entrées nécessaires pour calculer ses valeurs de contrôle. De plus, la synchronisation est partielle. Elle est effectuée pour transmettre des restrictions aux contrôleurs sous-jacents. Le reste des entrées, chacune, est transmis au contrôleur concerné. Pour le contrôle local, chaque contrôleur sous-jacent évolue à son propre rythme dicté par l'occurrence de ses entrées. Toutefois pour le contrôle global, les contrôleurs évoluent à la vitesse imposée par le contrôleur principal. Durant l'exécution distribuée désynchronisée, les contrôleurs sont totalement désynchronisés. Chacun évolue à son propre rythme et reçoit directement les entrées qu'il gère. Les contrôleurs communiquent par échange de valeurs de manière asynchrone. Cela pourrait conduire à des délais pour l'application du contrôle global.

Par ailleurs, comme dans l'exécution centralisée, dans l'exécution distribuée totalement synchronisée, les restrictions du contrôleur principal sont prises en compte immédiatement (dans la même réaction) par les contrôleurs sous-jacents. Dans l'exécution distribuée partiellement synchronisée, les restrictions du contrôleur principal sont également prises en compte immédiatement. Toutefois, puisque les contrôleurs sous-jacents ne sont pas totalement synchronisés avec le contrôleur principal, ces derniers peuvent faire un pas local pendant que le contrôleur principal initie un pas global. En raison des délais de communication entre le contrôleur principal et un contrôleur sous-jacent, l'ordre d'occurrence des entrées pourraient ne pas correspondre à l'ordre de leur réception par le contrôleur sous-jacent. Une partie de ses entrées est d'abord

transmise au contrôleur principal qui calcule des restrictions avant de transmettre les entrées aux autres contrôleurs. Pour éviter ce problème, un verrou peut être implémenté pour empêcher les contrôleurs sous-jacents de faire un pas local lorsque le contrôleur principal initie un pas global. Dans le cas de l'exécution distribuée désynchronisée, les restrictions peuvent ne pas être prises en compte immédiatement étant donné que les contrôleurs ne se synchronisent pas et que la communication est asynchrone.

7.3 Expérimentation

Les expérimentations présentées au chapitre précédent montrent que le code modulaire garantit les propriétés définies. L'objectif de cette section est d'évaluer le temps de réaction des contrôleurs de coordination lorsqu'ils sont exécutés sur des machines distinctes. Nous évaluons également le respect des objectifs, notamment le respect de l'objectif global dans le cas de l'exécution distribuée désynchronisée.

7.3.1 Configuration

Les expérimentations ont été réalisées dans notre centre de données expérimental, présenté à la section 6.4.1.

7.3.2 Évaluation

Nous avons injecté des charges de différentes intensités mais ayant le même profil (une montée de charge puis une charge constante). La montée de charge permet de déclencher des ajouts de serveurs. Une fois que les charges sont constantes et le degré de réplication stables, nous injectons des pannes pour observer le comportement des gestionnaires en présence des contrôleurs de coordination.

7.3.2.1 Durée de reconfiguration

Nous avons évalué la durée de reconfiguration dans l'exécution centralisée et dans l'exécution distribuée totalement synchronisée. Cette approche d'exé-

cution distribuée est le pire cas en terme de délai de réactivité, car bien que distribué, le contrôle reste centralisé au niveau du contrôleur principal. Nous considérons que la durée d'une reconfiguration correspond au temps écoulé entre l'émission de l'événement qui déclenche la reconfiguration (par une sonde) et la fin de la reconfiguration. Le tableau 7.1, donne la durée moyenne de reconfiguration dans l'exécution centralisée et l'exécution distribuée totalement synchronisée.

Mode d'exécution	**Durée moyenne de reconfiguration**
Centralisée	4.9 minutes
Distribuée totalement sync.	5.3 minutes

TABLE 7.1 – Durée moyenne de réaction

Nous avons observé que le traitement d'un événement (e.g., surcharge `Tomcat`) dure, en moyenne, *4.9 minutes* dans l'exécution centralisée. Dans l'exécution distribuée totalement synchronisée, cependant, il dure *5.3 minutes* en moyenne. Toutefois, nous avons constaté qu'il n'y a pas une variation très significative due à l'exécution distribuée totalement synchronisée. Par exemple, dans l'exécution centralisée, présentée dans la figure 7.12, le traitement de la surcharge du tier `Tomcat` a duré *5.149 minutes*. Dans l'exécution distribuée totalement synchronisée, présentée dans la figure 7.13, il a duré *5.156 minutes*.

Nous constatons également que la détection d'une panne d'un serveur `Tomcat` conduit à l'inhibition des actions d'ajout de serveurs aussi bien dans l'exécution centralisée que dans l'exécution distribuée totalement synchronisée. Dans la figure 7.12, une surcharge est ignorée durant la réparation d'une panne (*34 min*). Dans la figure 7.13, une surcharge est également ignorée durant la réparation d'un serveur (*22 min*).

7.3.2.2 Atteinte des objectifs de contrôle

Dans les exécutions présentées ci-dessous, nous rendons le serveur `Apache` inaccessible. Cela va conduire à la restauration de ce dernier. L'objectif de

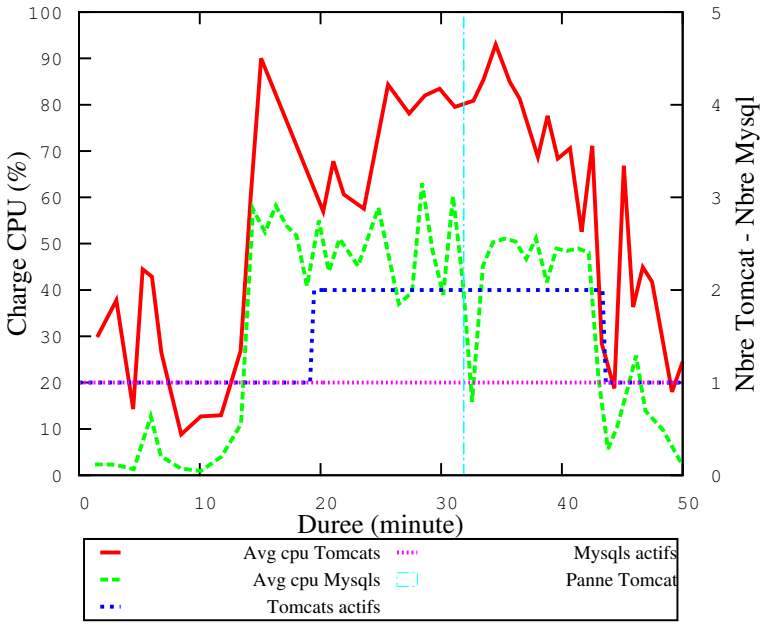

FIGURE 7.12 – Exécution centralisée

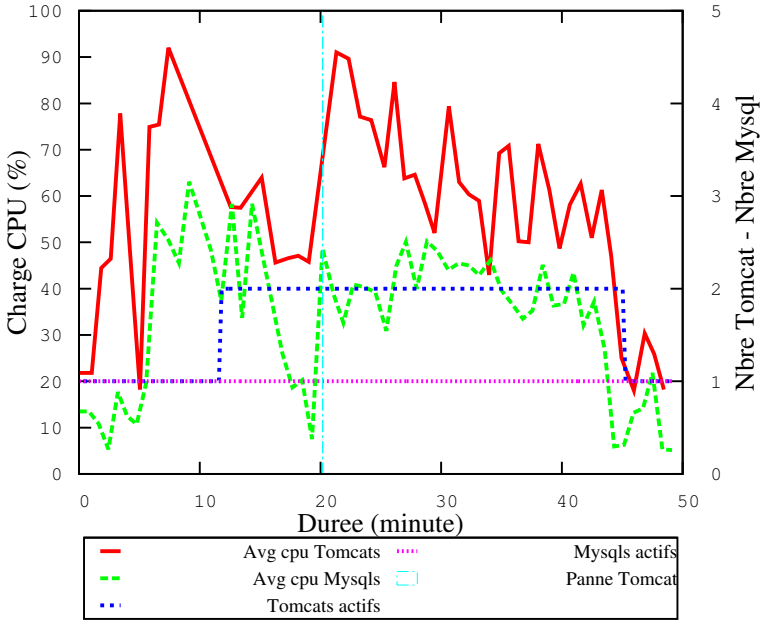

FIGURE 7.13 – Exécution distribuée totalement synchronisée

voir si les sous-charges qui se produisent dans les tiers `Tomcat` et `Mysql`, en cas de panne du serveur `Apache`, sont également ignorées dans les exécutions distribuées.

Exécution centralisée. Dans l'exécution centralisée, présentée dans la figure 7.14, la sous-charge détectée au niveau du tier `Mysql` (*33 min*) est ignorée à cause de la panne du serveur `Apache` en cours de restauration.

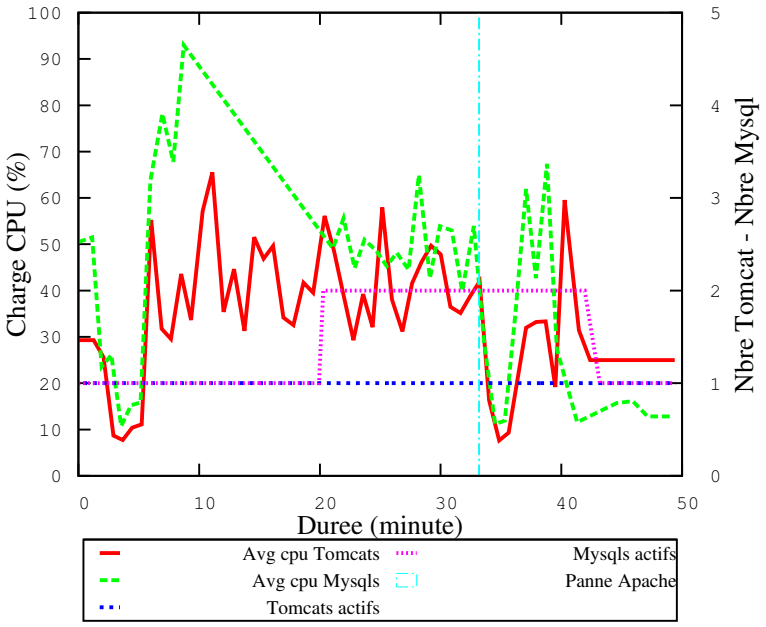

FIGURE 7.14 – Exécution centralisée : panne `Apache` et sous-charge `Mysql`

Exécution distribuée totalement synchronisée. Dans l'exécution distribuée totalement synchronisée, présentée dans la figure 7.15, la sous-charge au niveau du tier `Mysql` (*27.35 min*) est également ignorée à cause de la panne du serveur `Apache`.

Exécution distribuée partiellement synchronisée. Les figures 7.16, 7.17 et 7.18 présentent des exécutions durant lesquelles les contrôleurs sont exécutés de manière distribuée et partiellement synchronisée.

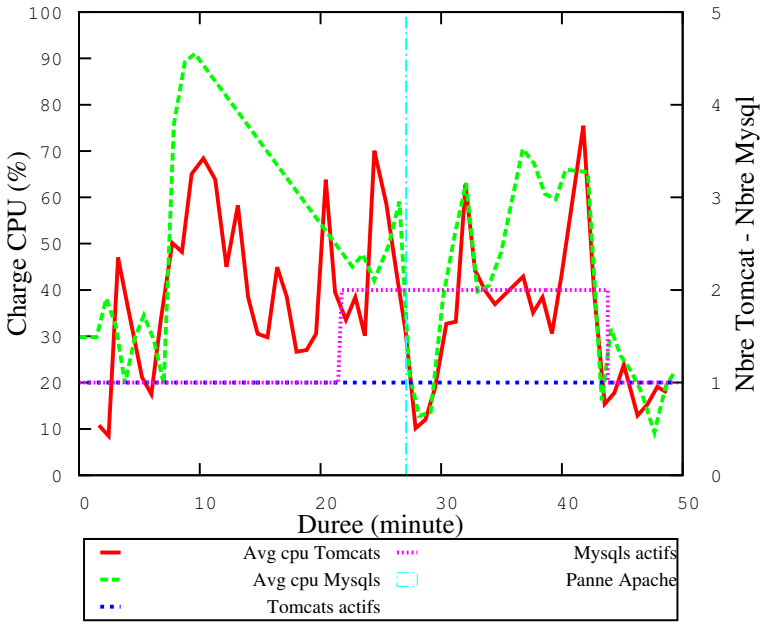

FIGURE 7.15 – Exécution distribuée totalement synchronisée : panne `Apache` et sous-charge `Mysql`

Dans les trois figures, nous observons qu'en cas de panne du serveur `Apache`, les sous-charges au niveau des tiers `Tomcat` et `Mysql` sont ignorées. Dans la figure 7.16, la sous-charge détectée au niveau du tier `Mysql` (*27.35 min*) est ignorée. Dans La figure 7.17, la sous-charge détectée au niveau du tier `Tomcat` est également ignorée. La figure 7.18 présente une exécution dans laquelle deux serveurs `Tomcat` ainsi que deux serveurs `Mysql` sont actifs. Nous constatons qu'une sous-charge est détectée dans les tiers `Tomcat` et `Mysql`. Toutefois, aucun retrait de serveur n'est effectué à cause de la panne du serveur `Apache`.

Le contrôleur principal a pu se synchroniser avec les contrôleurs distants, plus précisément, avec le contrôleur qui gère les gestionnaires des tiers `Apache` et `Tomcat`, pour permettre à ce dernier d'inhiber les actions de retrait de serveurs `Tomcat`.

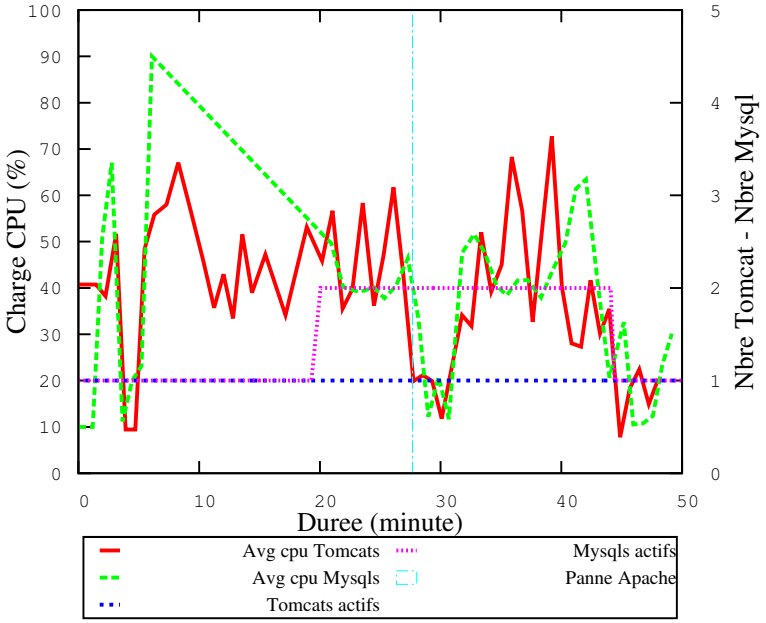

FIGURE 7.16 – Exécution distribuée partiellement synchronisée : panne `Apache` et sous-charge `Mysql`

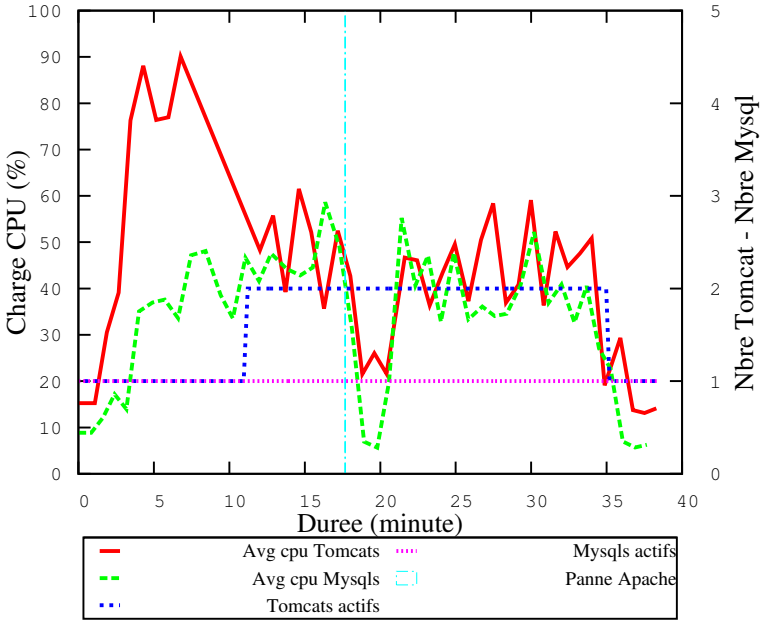

FIGURE 7.17 – Exécution distribuée partiellement synchronisée : panne `Apache` et sous-charge `Tomcat`

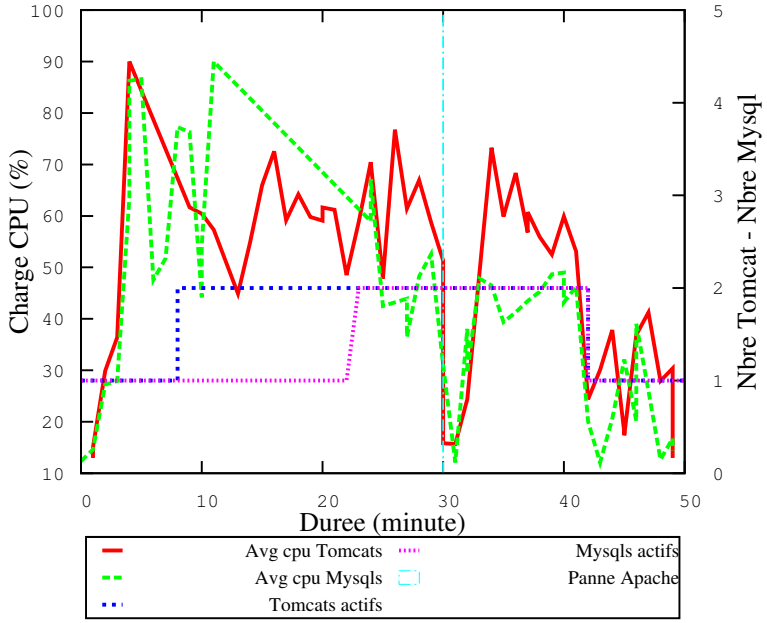

FIGURE 7.18 – Exécution distribuée partiellement synchronisée : panne `Apache` et sous-charge `Tomcat, Mysql`

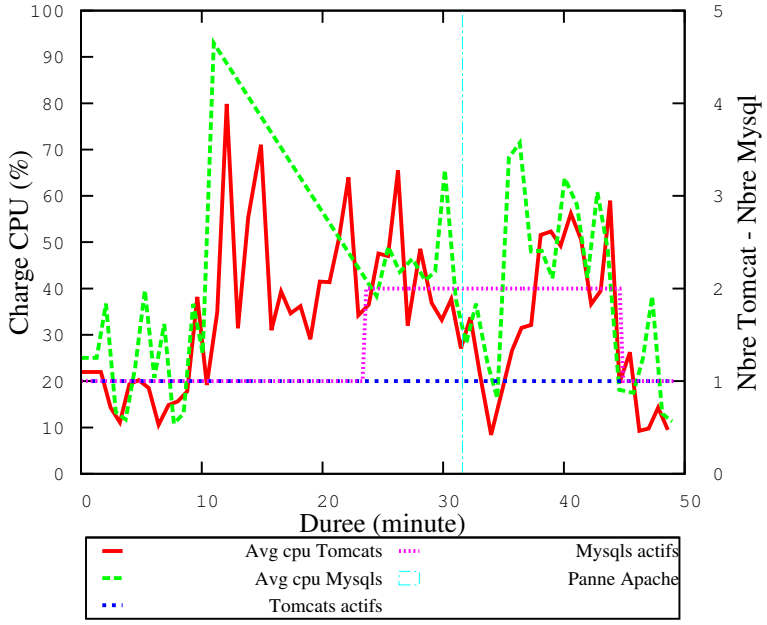

FIGURE 7.19 – Exécution distribuée désynchronisée : panne `Apache` et sous-charge `Mysql`

Exécution distribuée désynchronisée. Les figures 7.19 et 7.20 présentent le cas où les contrôleurs sont exécutés de manière distribuée et désynchronisée.

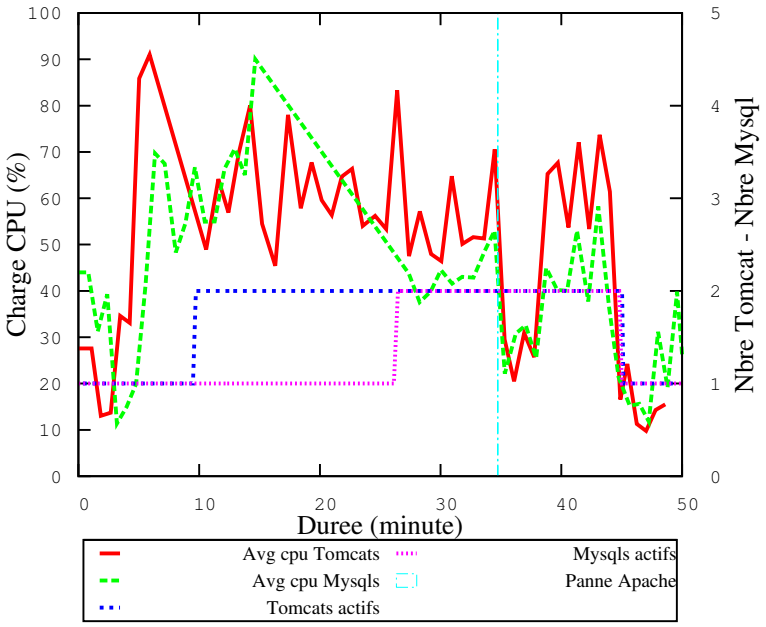

FIGURE 7.20 – Exécution distribuée désynchronisée : panne `Apache` et sous-charge `Tomcat, Mysql`

Nous constatons, dans ces exécutions, que les sous-charges au niveau du tier `Mysql` sont également ignorées suite à une panne du serveur `Apache`. Dans la figure 7.20, la sous-charge détectée dans le tier `Tomcat` est aussi ignorée.

Les valeurs de contrôleur émises par le contrôleur de coordination des tiers `Apache` et `Tomcat` sont reçues par le contrôleur de coordination des tiers `Mysql-proxy` et `Mysql` avant la détection de sous-charge. Cela a permis d'inhiber les actions de retrait de serveurs `Mysql`.

7.4 Conclusion

Nous constatons, sur notre plate-forme expérimentale, que le délai dû à la communication distante n'est pas très significatif. Nous constatons également

que les objectifs de coordination sont assurées dans les exécutions distribuées. Les sous-charges en cas de panne du serveur `Apache` sont ignorées aussi bien dans le tier `Tomcat` que dans le tier `Mysql`. Toutefois, les sous-charges, lorsqu'il n'y a pas de panne, sont traitées et conduisent au retrait de serveur, comme le montre la baisse du nombre de serveurs `Tomcat` et `Mysql` en fin d'exécution dans les figures 7.18 et 7.20.

De plus, dans les exécutions distribuées partiellement synchronisées et désynchronisées, la panne d'un contrôleur n'empêche pas les autres contrôleurs d'assurer leurs objectifs locaux.

8

Conclusion

L'automatisation des fonctions d'administration de systèmes informatiques a été étudiée dans plusieurs travaux de recherche. Ces travaux ont démontré la faisabilité de cette approche avec la conception de différents gestionnaires autonomes. Ces derniers assurent de manière cohérente les fonctions d'administration qu'ils implémentent. Aujourd'hui de nombreux gestionnaires autonomes sont disponibles, mais aucun n'implémente l'ensemble des fonctions d'administration nécessaires pour une gestion globale d'un système. La complexité de concevoir un gestionnaire complet rend nécessaire la coexistence de plusieurs gestionnaires pour une administration complète. Toutefois leur coordination est nécessaire pour assurer une cohérence des actions d'administration exécutées par les gestionnaires autonomes qui peuvent avoir des politiques contradictoires.

La coordination de gestionnaires autonomes requiert la synchronisation, partielle, de leurs activités et le contrôle de leurs actions d'administration pour éviter les incohérences. La théorie du contrôle fournit des techniques et des outils qui permettent de traiter ces aspects. De ce fait, nous proposons une approche basée sur le contrôle discret pour la conception et la validation de contrôleurs de coordination. Nous utilisons la programmation synchrone et la synthèse de contrôleur discret. Les langages synchrones sont des langages de haut niveau permettant une spécification formelle de système ; ils sont associés à des outils de vérification et de génération de code exécutable. La synthèse de

contrôleur, quant à elle, permet de raffiner une spécification incomplète en construisant une fonction de contrôle qui permet d'assurer le respect de propriétés non vérifiées par la spécification initiale, conçue par programmation synchrone par exemple. Nous proposons la synthèse modulaire pour la coordination de plusieurs gestionnaires autonomes. Cette technique permet une coordination modulaire et hiérarchique des gestionnaires. Cela permet également de réduire la complexité inhérente aux techniques de synthèse de contrôleur discret.

Pour la mise en oeuvre du contrôle, nous adoptons le modèle à composants. Chaque gestionnaire est encapsulé dans un composant qui fournit des fonctions d'introspection et de reconfiguration dynamique de l'état du gestionnaire. Les composants de gestionnaires à coordonner sont ensuite assemblés dans un composite dans lequel est intégré le contrôleur de coordination, obtenu par programmation synchrone et synthèse de contrôleur discret. Le contrôleur de coordination est connecté aux interfaces de contrôle des composants de gestionnaires pour restreindre ces derniers, à l'exécution, afin de garantir le respect de la politique de coordination.

Nous avons réalisé des expérimentations pour évaluer notre approche. Nous avons coordonné des gestionnaires autonomes qui assurent la performance, de la disponibilité et de l'optimisation des ressources d'un système multi-tiers. Les expérimentations réalisées dans ce travail de thèse montrent la faisabilité de notre approche pour la conception et la validation de contrôleurs de coordination pour assurer de manière cohérente les politiques de coordination définies.

Perspectives

Des améliorations sont en cours pour intégrer, en plus des aspects logiques, des aspects quantitatifs dans les algorithmes de synthèse de contrôleur [9]. Cela permettra de considérer des valeurs numériques dans la déclaration des objectifs de contrôle. Des poids représentant des fonctions de coûts pourront être associés aux états et transitions. Par ailleurs, l'expression de propriétés quantitatives permettra de comparer notre approche avec les approches basées sur des aspects quantitatifs qui utilisent, par exemple, des fonctions d'utilité.

Dans ce travail de thèse, les objectifs de contrôle sont relativement sim-

ples. Nous envisageons d'étudier des scénarios de contrôle plus élaborés, par exemple, un contrôle sur des séquences d'événements et/ou d'états.

Des améliorations sont également envisagées pour l'exécution distribuée. Les suppositions considérées dans ce travail de thèse, permettent de démontrer la faisabilité de l'exécution distribuée. Toutefois, en pratique, les pannes (machines, logiciels ou communication) ne peuvent pas être considérées inexistantes. De ce fait, nous prévoyons d'étudier la restauration du contrôle distribué de manière cohérente suite à des pannes de contrôleurs durant l'exécution.

La mise à jour des contrôleurs, durant l'exécution, est également une perspective intéressante à étudier. L'évolution de la structure du système à contrôler, les changements des objectifs de contrôle conduisent souvent à la recompilation du contrôleur. L'idée est de pouvoir faire cette recompilation à chaud, et intégrer les modifications dans le contrôleur pendant qu'il s'exécute ; tout cela sans devoir arrêter puis redémarrer le contrôleur ce qui conduit généralement à réinitialiser l'état de ce dernier. En effet, cette propriété pourrait être pertinente dans la gestion de centres de données où les applications hébergées et leurs gestionnaires autonomes sont instanciés et/ou arrêtés de manière dynamique.

Bibliographie

[1] Ahmad Al-Shishtawy and Vladimir Vlassov, *Elastman : Elasticity manager for elastic key-value stores in the cloud*, Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference (New York, NY, USA), CAC '13, ACM, 2013, pp. 7 :1–7 :10.

[2] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Vamis Xhagjika, *Libero : A framework for autonomic management of multiple non-functional concerns*, Euro-Par 2010 Parallel Processing Workshops (MarioR. Guarracino, Frédéric Vivien, JesperLarsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, eds.), Lecture Notes in Computer Science, vol. 6586, Springer Berlin Heidelberg, 2011, pp. 237–245 (English).

[3] Hua L. And, Hua Liu, Manish Parashar, and Salim Hariri, *A Component Based Programming Framework for Autonomic Applications*, Proc. of 1st International Conference on Autonomic Computing, 2004, pp. 10–17.

[4] C. André, F. Boulanger, and A. Girault, *Software implementation of synchronous programs*, IEEE International Conference on Application of Concurrency to System Design (Newcastle upon Tyne, UK), IEEE Computer Society, June 2001, pp. 133–142.

[5] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone, *The Synchronous Languages Twelve Years Later*, Proc. of the IEEE **91** (2003), no. 1.

[6] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot, *Synchronous programming with events and relations : the {SIGNAL} language and its*

semantics, Science of Computer Programming **16** (1991), no. 2, 103 – 149.

[7] Gérard Berry and Georges Gonthier, *The esterel synchronous programming language : Design, semantics, implementation*, Sci. Comput. Program. **19** (1992), no. 2, 87–152.

[8] Nicolas Berthier, Florence Maraninchi, and Laurent Mounier, *Synchronous programming of device drivers for global resource control in embedded operating systems*, Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems (New York, NY, USA), LCTES '11, ACM, 2011, pp. 81–90.

[9] Nicolas Berthier and Hervé Marchand, *Discrete Controller Synthesis for Infinite State Systems with ReaX*, IEEE International Workshop on Discrete Event Systems (Cachan, France), May 2014, pp. 420–427.

[10] S. Bouchenak, Noël Depalma, Daniel Hagimont, Sacha Krakowiak, and C. Taton, *Autonomic Management of Internet Services : Experience with Self-Optimization*, International Conference on Autonomic Computing (ICAC), Dublin, 12/06/2006-16/06/2006 (http ://www.ieee.org/), IEEE, juin 2006, p. (electronic medium) (anglais).

[11] Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sacha Krakowiak, Noël De Palma, Vivien Quéma, and Jean-Bernard Stefani, *Architecture-based autonomous repair management : Application to j2ee clusters.*, ICAC, IEEE Computer Society, 2005, pp. 369–370.

[12] Sara Bouchenak, Noël De Palma, Daniel Hagimont, and Christophe Taton, *Autonomic Management of Clustered Applications*, IEEE International Conference on Cluster Computing, CLUSTER '06, September 2006, pp. 1–11.

[13] Laurent Broto, Daniel Hagimont, Patricia Stolf, Noël Depalma, and Suzy Temate, *Autonomic management policy specification in Tune*, Annual ACM Symposium on Applied Computing (SAC), Fortaleza, Ceará, Brazil, 16/03/2008-20/03/2008 (http ://www.acm.org/), ACM, mars 2008, pp. 1658–1663 (anglais).

[14] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani, *The Fractal component model and its support in java*, Software – Practice and Experience (SP&E) **36** (2006), no. 11-12.

[15] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani, *The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems*, Softw. Pract. Exper. **36** (2006), no. 11-12, 1257–1284.

[16] Christos G. Cassandras and Stephane Lafortune, *Introduction to discrete event systems*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[17] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel, *Performance comparison of middleware architectures for generating dynamic web content*, Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware (New York, NY, USA), Middleware '03, Springer-Verlag New York, Inc., 2003, pp. 242–261.

[18] Shang-Wen Cheng, David Garlan, Bradley R. Schmerl, João Pedro Sousa, Bridget Spitnagel, and Peter Steenkiste, *Using architectural style as a basis for system self-repair*, Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture : System Design, Development and Maintenance (Deventer, The Netherlands, The Netherlands), WICSA 3, Kluwer, B.V., 2002, pp. 45–59.

[19] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet, *A conservative extension of synchronous data-flow with state machines*, Proceedings of the 5th ACM International Conference on Conference on Embedded Software (New York, NY, USA), EMSOFT '05, ACM, 2005, pp. 173–182.

[20] Rajarshi Das, Jeffrey O. Kephart, Charles Lefurgy, Gerald Tesauro, David W. Levine, and Hoi Chan, *Autonomic multi-agent management of power and performance in data centers*, Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent systems: industrial track (Richland, SC), AAMAS '08, International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 107–114.

[21] G. Delaval, N. De Palma, S.M.-K. Gueye, H. Marchand, and E. Rutten, *Discrete control of computing systems administration : A programming language supported approach*, Control Conference (ECC), 2013 European, July 2013, pp. 117–124.

[22] G. Delaval, É. Rutten, and H. Marchand, *Integrating discrete controller synthesis into a reactive programming language compiler*, Discrete Event Dynamic Systems **23** (2013), no. 4, 385–418.

[23] Gwenaël Delaval, Soguy Mak Karé Gueye, Éric Rutten, and Noël De Palma, *Modular coordination of multiple autonomic managers.*, CBSE'14, Proceedings of the 17th International ACM SIGSOFT Symposium on Component-Based Software Engineering (part of CompArch 2014), Marcq-en-Baroeul, Lille, France, June 30 - July 4, 2014 (Lionel Seinturier, Eduardo Santana de Almeida, and Jan Carlson, eds.), ACM, 2014, pp. 3–12.

[24] Gwenaël Delaval, Hervé Marchand, and Éric Rutten, *Contracts for modular discrete controller synthesis*, Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems (New York, NY, USA), LCTES '10, ACM, 2010, pp. 57–66.

[25] Gwenaël Delaval, Éric Rutten, and Hervé Marchand, *Integrating discrete controller synthesis into a reactive programming language compiler*, Discrete Event Dynamic Systems **23** (2013), no. 4, 385–418.

[26] Noël Depalma, B. Claudel, R. Lachaize, S. Bouchenak, and Daniel Hagimont, *Self-Protected System : an experiment*, Conference on Security and Network Architectures (SAR), Seignosse, France, 06/06/2006-09/06/2006 (http ://www.aw-bc.com/), Addison Wesley Longman, juin 2006, p. (electronic medium) (anglais).

[27] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet, *A modular memory optimization for synchronous data-flow languages*, Proc. of the ACM International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'12) (Beijing, China), June 2012.

[28] Soguy Mak-karé Gueye, Noël Palma, Eric Rutten, Alain Tchana, and Daniel Hagimont, *Discrete control for ensuring consistency between multiple autonomic managers*, Journal of Cloud Computing : Advances, Systems and Applications **2** (2013), no. 1, 16.

[29] Soguy Mak Karé Gueye, Noël De Palma, and Éric Rutten, *Component-based autonomic managers for coordination control.*, Coordination Models and Languages, 15th International Conference, COORDINATION 2013, Held as Part of the 8th International Federated Conference on Dis-

tributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings (Rocco De Nicola and Christine Julien, eds.), Lecture Notes in Computer Science, vol. 7890, Springer, 2013, pp. 75–89.

[30] Soguy Mak Karé Gueye, Noël De Palma, Éric Rutten, Alain Tchana, and Nicolas Berthier, *Coordinating self-sizing and self-repair managers for multi-tier systems.*, Future Generation Comp. Syst. **35** (2014), 14–26.

[31] Ajay Gulati, Anne Holler, Minwen Ji, Ganesha Shanmuganathan, Carl Waldspurger, and Xiaoyun Zhu, *Vmware distributed resource management : Design, implementation, and lessons learned*.

[32] Ajay Gulati, Ganesha Shanmuganathan, Anne Holler, and Irfan Ahmad, *Cloud-scale resource management : Challenges and techniques*, Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing (Berkeley, CA, USA), HotCloud'11, USENIX Association, 2011, pp. 3–3.

[33] N. Halbwachs, *Synchronous programming of reactive systems, a tutorial and commented bibliography*, Tenth International Conference on Computer-Aided Verification, CAV'98 (Vancouver (B.C.)), LNCS 1427, Springer Verlag, June 1998.

[34] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, *The synchronous dataflow programming language lustre*, Proceedings of the IEEE, 1991, pp. 1305–1320.

[35] David Harel and Amnon Naamad, *The statemate semantics of statecharts*, ACM Trans. Softw. Eng. Methodol. **5** (1996), no. 4, 293–333.

[36] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury, *Feedback Control of Computing Systems*, John Wiley & Sons, 2004.

[37] Jin Heo, Praveen Jayachandran, Insik Shin, Dong Wang, Tarek Abdelzaher, and Xue Liu, *OptiTuner: On Performance Composition and Server Farm Energy Minimization Application*, IEEE Trans. Parallel Distrib. Syst. **22** (2011), no. 11, 1871–1878.

[38] Timotheos Kastrinogiannis, Nikolay Tcholtchev, Arun Prakash, Ranganai Chaparadza, Vassilios Kaldanis, Hakan Coskun, and Symeon Papavassiliou, *Addressing stability in future autonomic networking.*, MONAMI (Kostas Pentikousis, Ramón Agüero Calvo, Marta García-Arranz, and Symeon Papavassiliou, eds.), Lecture Notes of the Institute for Com-

puter Sciences, Social Informatics and Telecommunications Engineering, vol. 68, Springer, 2010, pp. 50–61.

[39] Jeffrey O. Kephart and David M. Chess, *The vision of autonomic computing*, Computer **36** (2003), 41–50.

[40] Sanjay Kumar, Vanish Talwar, Vibhore Kumar, Parthasarathy Ranganathan, and Karsten Schwan, *vManage: loosely coupled platform and virtualization management in data centers*, Proceedings of the 6th International Conference on Autonomic Computing (New York, NY, USA), ICAC '09, ACM, 2009, pp. 127–136.

[41] Florence Maraninchi and Yann Rémond, *Mode-automata : a new domain-specific construct for the development of safe critical systems*, Sci. Comput. Program. **46** (2003), 219–254.

[42] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic, *Synthesis of discrete-event controllers based on the signal environment*, j. Discrete Event Dynamic System **10** (2000), no. 4.

[43] H. Marchand and M. Samaan, *Incremental design of a power transformer station controller using a controller synthesis methodology*, IEEE Trans. on Soft. Eng. **26** (2000), no. 8, 729 –741.

[44] Hervé Marchand, Patricia Bournai, Michel Le Borgne, and Paul Le Guernic, *Synthesis of Discrete-Event Controllers Based on the Signal Environment*, Discrete Event Dynamic Systems **10** (2000), 325–346.

[45] Ripal Nathuji and Karsten Schwan, *VirtualPower: coordinated power management in virtualized enterprise systems*, Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles (New York, NY, USA), SOSP '07, ACM, 2007, pp. 265–278.

[46] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath, *Load balancing and unbalancing for power and performance in cluster-based systems*, Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'01), September 2001.

[47] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu, *No "power" struggles: coordinated multi-level power management for the data center*, Proceedings of the 13th International Conference on Architectural Support for Programming Lan-

guages and Operating Systems (New York, NY, USA), ASPLOS XIII, ACM, 2008, pp. 48–59.

[48] P.J. Ramadge and W.M. Wonham, *Supervisory control of a class of discrete event processes*, SIAM J. on Control and Optimization **25** (1987), no. 1, 206–230.

[49] Ivan Rodero, Juan Jaramillo, Andres Quiroz, Manish Parashar, Francesc Guim, and Stephen Poole, *Energy-efficient application-aware online provisioning for virtualized clouds and data centers*, Green Computing Conference, August 2010, pp. 31–45.

[50] Ebada Sarhan, Atif Ghalwash, and Mohamed Khafagy, *Specification and implementation of dynamic web site benchmark in telecommunication area*, Proceedings of the 12th WSEAS International Conference on Computers (Stevens Point, Wisconsin, USA), ICCOMP'08, World Scientific and Engineering Academy and Society (WSEAS), 2008, pp. 863–867.

[51] Nikolay Tcholtchev, Ranganai Chaparadza, and Arun Prakash, *Addressing stability of control-loops in the context of the gana architecture : Synchronization of actions and policies*, IWSOS, 2009, pp. 262–268.

[52] Yin Wang, Terence Kelly, and Stéphane Lafortune, *Discrete control for safe execution of IT automation workflows*, Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (New York, NY, USA), EuroSys '07, ACM, 2007, pp. 305–314.

[53] Yiqiao Wang and John Mylopoulos, *Self-repair through reconfiguration : A requirements engineering approach*, Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (Washington, DC, USA), ASE '09, IEEE Computer Society, 2009, pp. 257–268.

Table des figures

2.1 Architecture d'un système autonome 11
2.2 Système de transitions 21
2.3 Modélisation avec Heptagon/BZR: Tâche différable 22
2.4 Modélisation avec `Heptagon/BZR`: programme BZR. 23
2.5 Exemple de composition parallèle 24
2.6 Exemple d'encapsulation 24
2.7 Système de transitions contrôlé 27
2.8 Heptagon/BZR contrat: exclusion mutuelle 28
2.9 Synthèse modulaire avec Heptagon/BZR. 29

3.1 Comportement d'un gestionnaire 37
3.2 Gestionnaire contrôlable 38
3.3 Modèle de la coexistence de gestionnaires 39
3.4 Spécification de stratégie de coordination 40
3.5 Spécification monolithique du contrôle 41
3.6 Spécification modulaire du contrôle 42
3.7 Composant Fractal 45
3.8 Composant composite 45
3.9 Composant de gestionnaire contrôlable 47
3.10 Composants de gestionnaires coordonnés 48
3.11 Composant composite 48
3.12 Extension de la contrôlabilité 51
3.13 Coordination hiérarchique 51

4.1 Gestionnaire d'auto-dimensionnement: self-sizing 57
4.2 Gestionnaire d'auto-régulation: Dvfs 58

4.3 Modèle de contrôle de self-sizing 61
4.4 Modèle global du mode d'exécution des Dvfs 63
4.5 Composition des modèles des gestionnaires self-sizing et Dvfs . 64
4.6 Coordination de gestionnaires self-sizing et Dvfs 65
4.7 Seuil minimal pour self-sizing: ajout de serveur 68
4.8 Seuil minimal pour self-sizing: retrait de serveur 68
4.9 Seuil minimal pour Dvfs 70
4.10 Exécution non coordonnée avec: *4750 requêtes/sec* 71
4.11 Exécution non coordonnée avec: *5000 requêtes/sec* 71
4.12 Exécution coordonnée avec: *4750 requêtes/sec* 72
4.13 Exécution coordonnée avec: *5000 requêtes/sec* 73
4.14 Exécution non coordonnée avec: *5542 requêtes/sec* 74
4.15 Exécution coordonnée avec: *5542 requêtes/sec* 74

5.1 Application JEE 78
5.2 Gestionnaire d'auto-réparation 80
5.3 Panne du serveur `Apache` 82
5.4 Panne du serveur `Mysql-Proxy` 82
5.5 Panne d'un serveur `Tomcat` 83
5.6 Panne d'un serveur `MySQL` 83
5.7 Modéle de contrôle de self-repair 85
5.8 Composition des modèles de self-sizing et self-repair 86
5.9 Coordination des instances de self-sizing et self-repair 89
5.10 Exécution non coordonnée: Panne du serveur Apache 92
5.11 Exécution non coordonnée: Panne d'un serveur Tomcat 93
5.12 Exécution non coordonnée: Panne du serveur Mysql-proxy . . . 94
5.13 Exécution non coordonnée: Panne d'un serveur Mysql 95
5.14 Exécution coordonnée: Panne du serveur Apache 96
5.15 Exécution coordonnée: Panne d'un serveur Tomcat 96
5.16 Exécution coordonnée: Panne du serveur Mysql-proxy 97
5.17 Exécution coordonnée: Panne d'un serveur Mysql 98

6.1 Modèle du gestionnaire self-sizing 103
6.2 Modèle du gestionnaire self-repair 104
6.3 Modèle du gestionnaire de consolidation 105

6.4 Synthèse monolithique 107
6.5 Réutilisation de modèles de contrôle 107
6.6 Tier dupliqué 108
6.7 Tier dupliqué avec aiguilleur en frontal 109
6.8 Multi-tiers 109
6.9 Centre de données avec deux applications multi-tiers. 110
6.10 Conception monolithique de la coordination 110
6.11 Conception modulaire de la coordination 111
6.12 Exécution non coordonnée: app 1: Panne `Apache` 113
6.13 Exécution non coordonnée: app 2: Panne `Tomcat` 114
6.14 Exécution coordonnée: app 1: Panne `Apache` 114
6.15 Exécution coordonnée: app 2: Panne `Tomcat` 115

7.1 Structure à l'exécution: Objet Java 118
7.2 Exécution distribuée synchronisée avec `Java rmi` 121
7.3 Exécution distribuée désynchronisée avec *Message Queuing* . . . 124
7.4 Spécification modulaire 125
7.5 Exécution distribuée totalement synchronisée 125
7.6 Tier dupliqué 127
7.7 Tier dupliqué avec aiguilleur en frontal 128
7.8 Multi-tiers 128
7.9 Exécution distribuée partiellement synchronisée 129
7.10 Coordination modulaire désynchronisée 130
7.11 Exécution distribuée désynchronisée 131
7.12 Exécution centralisée 135
7.13 Exécution distribuée totalement synchronisée 135
7.14 Exécution centralisée: panne `Apache` et sous-charge `Mysql` 136
7.15 Exécution distribuée totalement synchronisée: panne `Apache` et sous-charge `Mysql` 137
7.16 Exécution distribuée partiellement synchronisée: panne `Apache` et sous-charge `Mysql` 138
7.17 Exécution distribuée partiellement synchronisée: panne `Apache` et sous-charge `Tomcat` 138
7.18 Exécution distribuée partiellement synchronisée: panne `Apache` et sous-charge `Tomcat, Mysql` 139

7.19 Exécution distribuée désynchronisée: panne `Apache` et sous-charge `Mysql` 139
7.20 Exécution distribuée désynchronisée: panne `Apache` et sous-charge `Tomcat, Mysql` 140

Liste des tableaux

6.1 SCD: durée de la synthèse et la mémoire utilisée 111

7.1 Durée moyenne de réaction 134