



HAL
open science

Multi Autonomic Management for Optimizing Energy Consumption in Cloud Infrastructures

Frederico Guilherme Alvares de Oliveira Junior

► **To cite this version:**

Frederico Guilherme Alvares de Oliveira Junior. Multi Autonomic Management for Optimizing Energy Consumption in Cloud Infrastructures. Software Engineering [cs.SE]. Université de Nantes, 2013. English. NNT : ED 503-186 . tel-00853575

HAL Id: tel-00853575

<https://theses.hal.science/tel-00853575v1>

Submitted on 22 Aug 2013

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.

Thèse de Doctorat

Frederico Guilherme
ÁLVARES DE OLIVEIRA
JÚNIOR

*Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
sous le label de l'Université de Nantes Angers Le Mans*

École doctorale : 503 (STIM)

Discipline : Informatique et applications

Spécialité : Informatique

Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 09 avril 2013

Thèse n° : ED 503-186

Multi Autonomic Management for Optimizing Energy Consumption in Cloud Infrastructures

JURY

Président : **M. Pascal MOLLI**, Professeur, Université de Nantes
Rapporteurs : **M. Jean-Louis PAZAT**, Professeur, Institut National des Sciences Appliquées de Rennes
M^{me} Françoise BAUDE, Professeur, Université de Nice-Sophia Antipolis
Examineur : **M^{me} Fabienne BOYER**, Maître de conférences, Université Joseph Fourier, Grenoble
Directeur de thèse : **M. Pierre COINTE**, Professeur, École des Mines de Nantes
Co-directeur de thèse : **M. Thomas LEDOUX**, Maître de conférences, École des Mines de Nantes

Acknowledgements

I would like to express my sincerest appreciation to my advisors Dr. Thomas Ledoux and Dr. Pierre Cointe for the constant motivation, encouragement and scientific guidance all over the Ph.D. research as well as for the enormous help with the thesis writing.

I am very grateful to the French Ministry of Higher Education and Research (*Ministère de l'Enseignement supérieur et de la Recherche, MESR*) for funding this research work.

I am very thankful to Dr. Fabienne Boyer, Dr. Pascal Molli, Dr. Jean-Louis Pazat and Françoise Baude for taking some of their precious time to read my thesis manuscript and for the judicious comments and thoughtful questions during the Ph.D. viva. I would like to emphasize my gratitude to Dr. Pazat and Dr. Baude for following me as Ph.D. reviewers/reporters since the beginning of my Ph.D. research.

I would like to thank my team-mates from ASCOLA research group including Ph.D. students and faculties. In particular, Dr. Rémi Sharrock, Dr. Jean-Marc Menaud and Dr. Adrien Lèbre whom I had the pleasure to work with. More generally, I am really thankful to my fellows from the Computer Science Department at École des Mines de Nantes for making a huge homesickness to seem way less important than it actually is.

My sincere thanks to my French instructors Lynda and Veronique. Many thanks to Florence, Catherine and Diana, from Ecole des Mines de Nantes; Cecile and Hanane, from INRIA; Annie from Université de Nantes; and Marie, Soline and Caroline from Maison des Chercheurs Etrangers à Nantes. Their help was crucial to dealing with French administrative issues.

Last but not least, I would like to thank my family: my wife Sandra, for standing by my side and supporting me throughout these past three years and a half; my sisters Luciana and Ana; and my parents Frederico (*in memoriam*) and Evanilde, for the financial and spiritual support all through my life as well as for giving me the opportunity to have an outstanding education.

Contents

1	Introduction	11
1.1	Problem Statement	12
1.2	Contributions	12
1.3	Outline	14
1.4	Scientific Production	14
I	State of the Art	17
2	Background	19
2.1	Cloud Computing	20
2.1.1	Definition	20
2.1.2	Service and Deployment Models	20
2.1.3	Platform Virtualization	22
2.1.4	Advantages and Drawbacks	23
2.2	Autonomic Computing	24
2.2.1	Definition	24
2.2.2	Self-management	24
2.2.3	Autonomic Computing Reference Model	25
2.3	Component-based Software Engineering	26
2.3.1	Definitions	26
2.3.2	Architectural-based Dynamic Reconfiguration	28
2.3.3	Component-based Software Engineering for Service-Oriented Architectures	28
2.4	Constraint Programming	30
2.4.1	Definition	31
2.4.2	Modeling a CSP	31
2.4.3	Solving a CSP with CP	31
2.4.4	Drawbacks and Overcomes	31
2.5	Summary	32
3	Related Work	33
3.1	Energy/Resource/QoS Management at Application Level	33
3.1.1	QoS and Energy Management in Multi-tier Web Applications	34
3.1.2	QoS and Energy Management in Component/Service-Based Applications	35
3.2	Energy/Resource/QoS Management at the Infrastructure Level	37
3.2.1	Hardware Tuning	37
3.2.2	Virtualization	38
3.3	Cross-layered Energy/QoS-aware Management	40
3.3.1	Single Autonomic Management	41
3.3.2	Multiple Autonomic Management	44
3.4	Summary	50

II	Contributions	55
4	Cloud Computing and Component-based Software Engineering: Elasticity in Many Levels	57
4.1	Context Overview	58
4.1.1	Cloud Actors	58
4.1.2	Elasticity in Many Levels: From Infrastructure to Software Architecture	58
4.2	Energy-aware Elastic Infrastructure	59
4.2.1	Example Scenario: Virtual Machine Placement	59
4.3	QoS-aware and Energy-compliant Elastic Applications	60
4.3.1	Capacity Elasticity	60
4.3.2	Architectural Elasticity	61
4.3.3	Motivation Scenario: Advertisement Management Application	62
4.3.4	Conclusion	63
4.4	Autonomic Architecture to Optimize Energy Consumption	64
4.4.1	Architecture	64
4.4.2	Coordination and Synergy Issues	65
4.5	Summary	66
5	Synchronization and Coordination of Multiple Autonomic Managers to Improve Energy Efficiency	69
5.1	Synchronization and Coordination of Multiple Autonomic Managers	70
5.1.1	Multiple Autonomic Managers Architecture Model	70
5.1.2	Synchronization and Coordination	73
5.2	Synchronization and Coordination of Application and Infrastructure for Energy Efficiency in Cloud Computing	75
5.2.1	Architecture Overview	77
5.2.2	Infrastructure Manager	77
5.2.3	Application Manager	85
5.3	Summary	96
III	Validation	99
6	Research Prototype	101
6.1	Common Autonomic Framework	101
6.2	Application Manager	104
6.2.1	Application Meta-model	105
6.2.2	Managed System Technical Details	105
6.2.3	Autonomic Tasks	106
6.3	Infrastructure Manager	113
6.3.1	Infrastructure Meta-model	113
6.3.2	Managed System Technical Details	114
6.3.3	Autonomic Tasks	116
6.4	Autonomic Managers Lifetime	122
6.5	Summary	127
7	Experiments and Evaluation	131
7.1	Qualitative Evaluation	131
7.1.1	Experimental Testbed	132
7.1.2	Budget-driven Architectural Elasticity Scenario	134
7.1.3	Scale Down Scenario	137
7.1.4	Promotion Scenario - With Architectural Elasticity	139

7.1.5	Promotion Scenario - Without Architectural Elasticity	143
7.1.6	Discussion	147
7.2	Quantitative Evaluation	148
7.2.1	Synchronization and Coordination Protocols	149
7.2.2	Results	150
7.2.3	Optimization Problems	153
7.3	Summary	157
8	Conclusion	159
8.1	Problem Statement Revisited	159
8.2	Summary of Contributions	159
8.3	Perspectives	161
Appendix A	Resumé	167
A.1	Introduction	167
A.1.1	Problématique	168
A.1.2	Contributions	168
A.2	État de l'art	169
A.3	Un modèle multi-autonome pour l'informatique en nuage	171
A.3.1	Modèle architectural	171
A.3.2	Protocole de synchronisation pour l'accès à la base de connaissance publique	173
A.3.3	Protocole de coordination dirigé par des évènements	174
A.4	Une approche pour la gestion multi-autonome pour l'informatique en nuage	175
A.4.1	Vue architecturale globale	175
A.4.2	Gestionnaire d'infrastructure	177
A.4.3	Gestionnaire d'application	181
A.5	Évaluation	186
A.5.1	Banc d'essai	186
A.5.2	Protocole d'expérience	187
A.5.3	Résultats	187
A.6	Conclusion	188
A.6.1	Récapitulatif de la problématique	188
A.6.2	Sommaire des contributions	190
A.6.3	Perspectives	192

Appendices

List of Tables

3.1	Related Work Summary.	53
4.1	Resource Requirements for the Advertisement Management Application.	63
4.2	Possible Architectural Configurations for the Advertisement Management Application.	64
5.1	MAPE-K Control Loop Timing Notation.	75
7.1	Synthetic Application Resource Requirements.	133
7.2	Application Settings for the Budget-driven Architectural Elasticity Scenario.	135
7.3	Application Settings for the Scale Down Scenario.	138
7.4	Application Settings for the Promotion Scenario (with Architectural Elasticity).	141
7.5	Promotion Settings for the Promotion Scenario (with Architectural Elasticity).	142
7.6	Arrival Rates for Endogenous Events.	149
7.7	Energy Shortage Parameters.	154
7.8	Energy Shortage Constraint Margins.	154
A.1	Sommaire des travaux connexes.	172

List of Figures

2.1	Cloud Computing Stack.	21
2.2	Platform Virtualization.	23
2.3	Autonomic Computing Reference Architectural Model Proposed by IBM [KC03].	25
2.4	Service Component Architecture Concepts.	30
3.1	Multi-tiered Web Application [AB10].	34
3.2	The MoKa Web Application Model [AB10].	34
3.3	Autonomic Adaptation Design [PDPBG09].	36
3.4	Illustration of Three Composite SaaS [YT12].	37
3.5	Hierarchical Architecture of Virtual Machine Management System.	39
3.6	Coordinated Power Management Architecture for Datacenters [RRT ⁺ 08].	45
3.7	Semantic-less Coordination Algorithm for (a) Sytem/Infrastructure and (b) Application [KLS ⁺ 10].	46
3.8	Coordinated Power and Performance Control Architecture [WW11].	48
3.9	Intersection of the State of the Art Areas.	51
4.1	Energy-aware Placement Example Scenario.	60
4.2	Example of Component Scaling Up and Down.	61
4.3	Advertisement Manager Architecture Overview.	62
4.4	Multi-level Architecture for QoS and Energy Efficiency Optimization.	66
5.1	Autonomic Manager Types: (a) Independent Autonomic Managers; (b) Coordinated Autonomic Managers; and (c) Synchronized and Coordinated Autonomic Managers.	72
5.2	Autonomic Actions and Events Hierarchy.	73
5.3	Interloop Communication.	74
5.4	Autonomic Managers Coordination and Token Synchronization Protocols and Timings.	76
5.5	Multi-control loop Architecture for Cloud Computing.	78
5.6	Infrastructure Manager Events and Actions.	80
5.7	Performance Degradation Levels.	87
5.8	Application Manager Events and Actions.	89
6.1	Meta-model Arrows Semantics.	102
6.2	Common Autonomic Framework Meta-model.	103
6.3	The Plan Meta-model of the Common Autonomic Framework Meta-model.	104
6.4	Application Meta-model.	106
6.5	Managed Application Technical Details.	107
6.6	Application Manager Autonomic Meta-model.	108
6.7	Application Manager Monitoring Task Meta-model.	109
6.8	Application Manager Wildcat Resource Hierarchy.	109
6.9	Application Manager Analysis Meta-model.	111
6.10	Application Manager Planning Meta-model.	111

6.11	Application Manager Execution Meta-model (Interloop Actions).	112
6.12	Application Manager Execution Meta-model (Actions on the Managed System).	112
6.13	Infrastructure Meta-model.	114
6.14	Managed Infrastructure Technical Details.	115
6.15	Infrastructure Manager Autonomic Meta-model.	116
6.16	Infrastructure Manager Monitoring Meta-model.	117
6.17	Infrastructure WildCAT Resource Hierarchy.	118
6.18	Infrastructure Manager Analysis Meta-model.	119
6.19	Infrastructure Manager Planning Meta-model.	120
6.20	Infrastructure Manager Execution Meta-model for Interloop Actions.	121
6.21	Infrastructure Manager Execution Meta-model for Change Public Knowledge Actions.	121
6.22	Infrastructure Manager Execution Meta-model for Actions on the Managed System.	122
6.23	Autonomic Events Creation Over the Time.	123
6.24	Autonomic Actions creation over the time.	124
6.25	Token Acquisition / Release Over the Time.	125
6.26	Application Manager Initialization Procedure.	126
6.27	Application Manager Interactions Over the Time.	128
6.28	Infrastructure Manager Interactions Over the Time.	129
7.1	Synthetic Application Architectural Configurations: (a) k_1 and (b) k_2 .	132
7.2	Workload for the Budget-driven Architectural Elasticity Scenario.	135
7.3	Average Response Time for the Budget-driven Architectural Elasticity Scenario.	136
7.4	Switch Between Architectural Configurations.	136
7.5	Ratio of the Arithmetic Mean of Quality of Service to Cost.	137
7.6	Ratio of Application Weighted Average Quality of Service to Cost for (a) $\alpha = 0.4$ and $\beta = 0.6$; and (b) $\alpha = 0.3$ and $\beta = 0.7$.	138
7.7	Workloads for the Scale Down Scenario.	138
7.8	The Average Response Time upon a <i>Scale Down</i> event for: (a) 1 Physical Machines, (b) 3 Physical Machines and (c) 5 Physical Machines to Shutdown.	140
7.9	Power Consumption upon an <i>Energy Shortage</i> Event.	141
7.10	Workloads for the Promotion Scenario (with Architectural Elasticity).	142
7.11	Infrastructure Income Increase (%) with Respect to the Promotion-disabled Run (with Architectural Elasticity).	143
7.12	Total Infrastructure Power Consumption (with Architectural Elasticity).	143
7.13	Workloads for the Promotion Scenario (without Architectural Elasticity).	144
7.14	Infrastructure Income Decrease (%) with Respect to the Run without Promotion (Without Architectural Elasticity).	145
7.15	Infrastructure Power Consumption for the Promotion Scenario (without Architectural Elasticity).	146
7.16	Energy Consumption Difference between Promotion-disabled and Promotion-enabled Runs.	146
7.17	Ratio of the Normalized Income to the Power Consumption.	147
7.18	Average Token Waiting Time for (a) Low and (b) High Arrival Rates.	151
7.19	Per-Event Processing Time for (a) Low and (b) High Arrival Rates.	152
7.20	The Solving Time for (a) the Virtual Machine Packing Problem and (b) The Solving Time for the Virtual Machine Placement Problem.	155
7.21	The Solving Time for the Energy Shortage Problem.	156
7.22	The Solving Time for the Architecture and Capacity Planning Problem.	156
A.1	Intersection des travaux connexes.	169
A.2	Patrons de coordination de gestionnaires autonomes.	173

A.3	Modèle architectural proposé pour la coordination des multiples gestionnaires autonomes.	174
A.4	Hiérarchie d'actions et d'évènements autonomes.	175
A.5	Vue globale de l'architecture.	176
A.6	Actions et événements du gestionnaire IM.	178
A.7	Événements et actions du gestionnaire applicatif.	183
A.8	Consommation de puissance lors d'une pénurie d'énergie.	188
A.9	Le temps de réponse moyen lors d'un événement <i>Scale Down</i> pour l'arrêt de: (a) 1, (b) 3, et (c) 5 PMs.	189

Introduction

Over the last decade, the electricity use of Information Technology and Communication (ITC) has surprisingly increased. Koomey estimated that the electricity use due to data centers worldwide has doubled from 2000 to 2005 [Koo07]. In a report published by the the U.S. Environmental Protection Agency (EPA), it was estimated that only the U.S. data centers were responsible for the consumption of 61 billion kilowatt-hours (kWh) in 2006, which corresponds to 1.5% of the nation's total electricity use. In a recent report [Koo11], Kommey estimated that the electricity use of data centers increased at a rate of 54% worldwide and at 36% in the U.S. from 2006 to 2010. Apart from the environmental impacts deriving from that level of consumption, energy consumption becomes a major concern, since energy costs are becoming more and more predominant in data centers total costs [BH07].

There are many reasons for that growth. The popularization of Internet, started in late 1990's, has motivated the migration of local softwares to remote, online and 24/7 available services. More recently, the dissemination of cloud computing [AFG⁺09, BYV⁺09, Mon12] has also pushed users to consume services in the cloud in a pay-as-you-go fashion. As a consequence, we have seen an explosion in the number of new data centers worldwide.

Paradoxically, cloud computing models [Pet11] can be seen as a very powerful tool to face the issue of energy consumption. From the infrastructure provider point of view, techniques like virtualization [SN05] enable the mutualization of a physical machine's (PM) resources between several virtual machines (VMs). As a result, the workload of several applications can be concentrated in a minimum number of PMs, which allows the unused ones to be shutdown and hence reduce the energy consumption [HLM⁺09].

From the application provider point of view, cloud computing permits to precisely request/release resources on the fly. It means that the infrastructure provider can deliver computational and storage resources in a flexible and elastic manner, while charging the applications only for what they actually consume. This is particularly interesting for applications that need to cope with a highly variable workload, so they can dynamically adjust the right amount of resources needed to deliver the desired level of Quality of Service (QoS).

The necessity of cloud based systems to be more responsive and autonomous to environment changes is one of the main reasons for the great adoption of Autonomic Computing [KC03]. In point of fact, Autonomic Computing makes applications capable of reacting to a highly variable workload by dynamically adjusting the amount of resources needed to be executed while keeping its QoS [KCD⁺07, VTM10, APTZ12, CPMK12, YT12], whereas from the infrastructure provider point of view, it also makes the infrastructure capable of rapidly reacting to context changes (e.g. increase/decrease of physical resource usage) by optimizing the allocation of resources and

thereby reduce the costs related to energy consumption [HLM⁺09, LLH⁺09, BB10, FRM12].

1.1 Problem Statement

In spite of the recent efforts in providing more energy-proportional hardware [Bar05], nowadays PMs still consume a lot of power even when they are inactive [BH07]. Hence, the way applications are deployed in VMs may indirectly interfere on the infrastructure energy consumption. Indeed, application providers perform actions such as for requesting/releasing computing/storage resources with the purpose to optimize the trade-off QoS and cost due to resource usage. These actions may lead the infrastructure to a state that lowers the infrastructure utilization rate and by consequence the energy efficiency. In other words, QoS-driven decisions taken at the application level may indirectly interfere in the energy consumption of the underlying infrastructure. For example, following a decreasing demand, an application may decide to release part of the computing resources allocated to it. As a consequence, instead of decreasing the energy consumption in the same proportion (energy-proportional limitation), this action (release resources), taken at the application level, may lead the infrastructure to a less energy efficient state, since the number of free spots in the physical infrastructure increases for nearly the same amount of energy consumed. Similarly, the infrastructure provider may face constrained situations such as an energy shortage, which would consequently require the shutdown of part of the physical infrastructure. As a result, this event at the infrastructure level may drastically interfere in the guest applications' QoS. Therefore, the lack of synergy between cloud systems may lead both applications and infrastructures to undesired states.

As a further matter, in order for the synergy between applications and infrastructure to be effective, it is necessary that applications are flexible enough to react to events produced not only by the application itself (e.g. increasing/decreasing number of clients), but also by the infrastructure (e.g. renting fees changes and/or resource constraints). That way, a reconfigurable capability would make applications more sensitive to infrastructure changes and therefore more adaptable to a wider range of situations. For instance, in scenarios of resource constraints imposed by the infrastructure (e.g. energy shortage), a video-on-demand application would be able to seamlessly cope with this restriction by degrading a video-specific quality attribute (e.g. resolution, frame rate, etc.). In the other sense, the application can be encouraged by the infrastructure provider to occupy a certain part of the infrastructure so as to improve the infrastructure utilization rate (and consequently the energy efficiency) as well as to enable the application to re-increase the video-specific quality attribute.

1.2 Contributions

This thesis focuses on the autonomic management of applications and infrastructure to improve the QoS and the energy efficiency in cloud environments.

We believe that managing each application along with the infrastructure in a single autonomic system is a very hard task. Each application may have its own goals and requirements, which consequently impacts the way their manager should be conceived (e.g. with respect to the reactivity time scale). Moreover, it is common for cloud applications and infrastructure to belong to different organizations, which implies that applications details should not necessarily be visible at the infrastructure. In the same way, application providers should not neither be aware of the infrastructure details. For these reasons, we advocate a multiple autonomic management approach, in which each application is equipped with its own autonomic manager in charge of determining the minimum amount of computing resources necessary to deliver the best QoS possible. Another autonomic manager autonomously manages the physical resources at the infrastructure level in an energy aware manner.

In order to manage the lack of synergy between applications and infrastructure, we propose an approach for the synchronization and coordination of multiple autonomic managers. The objective

is to promote a synergy between applications and infrastructure so the information of one manager can more easily be taken into account by the others so as to avoid negative interferences.

Last but not least, we extend the usual resource elasticity capability (request/release of computing/data resources) by considering applications as *white boxes*. In other words, we assume that the way applications are composed (i.e. which components are used and how they are connected) may lead to different architectural configurations with different needs in terms of resources. As a result, applications become more sensitive and reactive to changes in the environment, no matter whether these changes occur in the application itself (e.g. a drastic workload increase with a limited budget) or in the infrastructure (e.g. resource restriction due to energy constraints).

This thesis' contributions are summarized as follows:

- **Architectural Elasticity Capability.** We propose a Component-based Software Engineering (CBSE) approach in which applications are deployed on the cloud as *white boxes*. Thus, the way applications internals are bound as well as their domain-specific QoS criteria are taken into consideration to configure them. Complementary to the resource elasticity property, in which the adaptation to the environment is performed only by adding or removing resources, the *architectural elasticity* promotes the management of applications that are more flexible, more reactive to environment changes and therefore more suitable for a wider range of runtime contexts. As a result, applications become more susceptible to energy constraints and optimization issues.
- **A Novel Autonomic Model.** In order to accommodate multiple autonomic managers, we propose a model for synchronization and coordination of autonomic managers, which consists of a coordination and a synchronization protocols. The former defines communication interfaces based on autonomic managers' actions and events. The latter defines a way to enable consistent interactions between managers. More precisely, we propose a shared knowledge-based synchronization pattern. Hence, decisions taken by one manager may take into consideration some information provided by the others.
- **Application and Infrastructure Managers for the Cloud.** Based on the model mentioned on the previous paragraph, we specified and implemented an *Application Manager* and an *Infrastructure Manager*. The former is an autonomic manager whose objective is to make applications self-manageable. It listens for events on the managed application (e.g. the workload) and from the *Infrastructure Manager*. According to these data, the *Application Manager* decides whether to reconfigure the managed application. For example, a decision should be made to find the amount of resources needed to meet the QoS requirements while keeping a certain architectural configuration or whether to degrade the architectural configuration (change the internal configuration) in order to cope with resource constraints imposed by the infrastructure provider. Finally, if a reconfiguration is applicable, a set of actions is dynamically performed on the managed application (e.g. deploy, start, stop, bind component) or onto other managers such as the infrastructure manager (e.g. request/release computing resource). The latter is also an autonomic manager whose objective is to make the infrastructure self-manageable. It listens for events on the managed infrastructure (e.g. PMs utilization rate) and from the *Application Manager* (e.g. requests on computing resources by means of VMs). According to the collected data, it decides whether or not to reconfigure. For instance, a decision should be made in order to find where new VMs requested by applications should be placed so that the infrastructure utilization rate is optimized. The result is a set of actions performed on the managed infrastructure (e.g. create/destroy VM) and/or onto other autonomic managers (e.g. notify VMs creation).
- **Synergy between Applications and Infrastructure.** We promote a synergy between *Infrastructure* and *Application Managers* with the purpose to mitigate negative interferences of one manager in another. Hence, we define communication interfaces based on the coordination protocol previously mentioned. They can be used by a manager to inform other

managers about important changes that may affect them. For example, in periods of energy shortage, the *Infrastructure Manager* may warn *Application Managers* that part of the infrastructure must be shutdown and that some of the allocated resources will be temporarily unavailable. It should be noticed that applications equipped with *architectural elasticity* are more likely to cope with that kind of situation. Apart from the communication interface based synergy, we also propose a more implicit synergy mechanism. It consists in implementing the renting fees due to computing resources in a variable manner, meaning that they can be dynamically adjusted by the *Infrastructure Manager* (e.g. by creating promotions) in order to encourage *Application Managers* to request more resources and thus optimize the infrastructure utilization rate. We rely on the autonomic model previously mentioned to define the renting fees as a shared knowledge so several autonomic managers can access it in a synchronized way.

- **Constraint Programming Model to Solve Optimization Problems.** We provide a set Constraints Satisfaction and Optimization Problems (CSOP) that models the decisions at both managers. The solutions for these problems are implemented with Constraint Programming (CP) [RVBW06] techniques.

1.3 Outline

This section describes how the rest of this document is organized. Chapter 2 is dedicated to the technical background necessary for the understanding of this thesis work. Chapter 3 discusses a selection of recent and relevant scientific work related to the issues addressed in this thesis. In Chapter 4, we provide some discussion about how the flexibility inherent to cloud computing along with the flexibility achieved by component-based software engineering can assist application and infrastructure providers to cope with dynamic environments. Based on these levels of flexibility, we present an overview of a multi-level self-management approach for QoS and energy optimization in cloud environments. Chapter 5 presents the autonomic model for coordination and synchronization of multiple autonomic managers and details the application and infrastructure managers. Chapter 6 describes the implementation details of the research prototype used to validate this thesis work. In Chapter 7, some results obtained from a set of experiments on the research prototype are presented and discussed. These experiments were performed on Grid'5000, a real physical infrastructure testbed. Finally, Chapter 8 summarizes the contributions of this thesis and presents the perspectives on the topic by outlining the on-going and future work.

1.4 Scientific Production

REFEREED ARTICLES

- 1 Frederico Alvares de Oliveira Jr., Thomas Ledoux, "Self-management of cloud applications and infrastructure for energy optimization", In SIGOPS Operating Systems Review, ACM, vol. 46, no. 2, New York, NY, USA, 2012.

CONFERENCE PAPERS

- 2 Frederico Alvares de Oliveira Jr., Rémi Sharrock, Thomas Ledoux, "A framework for the coordination of multiple autonomic managers in cloud environments", In Proceedings of the Seventh IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO'2013, Philadelphia, PA, USA. To Appear.
- 3 Frederico Alvares de Oliveira Jr., Rémi Sharrock, Thomas Ledoux, "Synchronization of Multiple Autonomic Control Loops: Application to Cloud Computing", In Proceedings of the 14th International Conference on Coordination Models and Languages, COORDINATION'12, Stockholm, Sweden, 2012.

WORKSHOP PAPERS

- 4 Frederico Alvares de Oliveira Jr., Thomas Ledoux, "Self-management of applications QoS for energy optimization in datacenters", In Proceedings of the 2nd International Workshop on Green Computing Middleware, ACM, New York, NY, USA, pp. 3:1-3:6, 2011.
- 5 Frederico Alvares de Oliveira Jr., Thomas Ledoux, "Self-optimisation of the energy footprint in service-oriented architectures", In Proceedings of the 1st Workshop on Green Computing Middleware, ACM, New York, NY, USA, pp. 4-9, 2010.

BOOK CHAPTERS

- 6 Frederico Alvares de Oliveira Jr., Adrien Lèbre, Thomas Ledoux, Jean-Marc Menaud, "Self-management of applications and systems to optimize energy in data centers", Chapter in Achieving Federated and Self-Manageable Cloud Infrastructures: Theory and Practice, IGI Global, 2012.



State of the Art

Background

The goal of this chapter is to present some concepts and technologies that are necessary for the good comprehension of the remainder of this thesis.

Firstly, we present some concepts related with cloud computing by providing a discussion about its definition, models and technologies used to implement it. In addition, we highlight the main advantages and drawbacks. Then, we give a short introduction on autonomic computing, by briefly defining it. Also, we describe the self-management properties and present the autonomic computing architectural reference models. Then, we present the main concepts related to component-based software engineering. We provide a discussion about the definitions of component as well as about the support for dynamic reconfiguration in some component models. We point out some limitations of component-based architectures and how they meet service-oriented architecture to overcome these limitation while supporting dynamic adaptation. Finally, we briefly describe constraint programming, which is used in this work to perform constraint satisfaction and optimization analysis. We provide a definition of constraint programming as well as with an introduction on how problems are modeled and solved. Lastly we provide a discussion about some drawbacks and outcomes of constraint programming.

Contents

2.1	Cloud Computing	20
2.1.1	Definition	20
2.1.2	Service and Deployment Models	20
2.1.3	Platform Virtualization	22
2.1.4	Advantages and Drawbacks	23
2.2	Autonomic Computing	24
2.2.1	Definition	24
2.2.2	Self-management	24
2.2.3	Autonomic Computing Reference Model	25
2.3	Component-based Software Engineering	26
2.3.1	Definitions	26
2.3.2	Architectural-based Dynamic Reconfiguration	28
2.3.3	Component-based Software Engineering for Service-Oriented Architectures	28
2.4	Constraint Programming	30
2.4.1	Definition	31

2.4.2	Modeling a CSP	31
2.4.3	Solving a CSP with CP	31
2.4.4	Drawbacks and Overcomes	31
2.5	Summary	32

2.1 Cloud Computing

This section presents the main concepts of cloud computing. First, we provide a definition of cloud computing. Then, we describe the service and deployment models of cloud computing. Finally, we give a brief introduction on virtualization, which is a technology widely adopted to implement cloud computing in a feasible and cost effective way.

2.1.1 Definition

According to the U.S. National Institute of Standards and Technology (NIST) [Pet11], cloud computing is a model for enabling on-demand network access to a shared, and often virtualized, pool of resources. Although cloud computing has received a lot of attention over the last years, the concepts involved are not entirely new [AFG⁺10, AFG⁺09, BYV⁺09]. Indeed, grid computing also aims at the data and computing resources provisioning in a geographically distributed, standardized, network-accessible way with the objective to meet a specific task or goal [FK04]. Utility computing, in turn, is about delivering computing as utility, just like electricity does for electrical power [BYV⁺09]. Hence, consumers of computing services are charged in a pay-per-use basis, that is, they pay only for what they actually use, instead of a flat and fixed price. Furthermore, software as utility is also a similar model to the one offered by the application service providers (ASPs) [Fac02], which are third-party organizations providing software applications that can be accessed 24/7 via Internet. In short, cloud computing can be thought as the convergence of lots of concepts (e.g. ASP, grid and utility computing) that have been proposed for a while, but thanks to the recent technological advance only now became viable [Mon12].

2.1.2 Service and Deployment Models

Service Models

Cloud computing is based on the service-consumer model, which means that resources (no matter if it is a software, hardware, infrastructure or anything else) are delivered to consumers as services. Generally, the acronym XaaS is used to designate anything-as-a-service or everything-as-a-service. The most emblematic service models of cloud computing are Software-as-a-Service, Platform-as-a-Service and Infrastructure-as-a-Service. These models are detailed as follows:

- **Software as a Service (SaaS):** the capability to deliver Internet accessible applications deployed on a cloud infrastructure to consumers. It means that SaaS consumers does not control the software application neither the underlying platform or infrastructure used to deploy it the application. There are many examples of SaaS provides such as Salesforce.com¹ which is a suit of business applications, Office 365², which is a suit of office tools (e.g. text processing and spreadsheet application) provided by Microsoft, and Google Apps³, which is a productivity suite application provided by Google.
- **Platform as a Service (PaaS):** the capability to deliver a means to facilitate the deployment of applications on the underlying cloud infrastructures to consumers. Like the SaaS

¹<http://www.salesforce.com>

²<http://www.microsoft.com/office365/>

³<http://www.google.com/Apps>

model, the PaaS consumer does not control the operating system (OS), the storage nor the underlying infrastructure. Examples of PaaS include Google AppEngine ⁴ and CloudBees ⁵.

- **Infrastructure as a Service (IaaS):** the capability to deliver processing (e.g. CPU and RAM), storage and network resources to consumers. The consumers do not control the physical infrastructure but it may control the OS, storage and deployed applications. Examples of IaaS include Amazon Elastic Compute Cloud (Amazon EC2 ⁶), Amazon Simple Storage Service (Amazon S3 ⁷), and Microsoft Azure ⁸.

Figure 2.1 depicts the cloud computing stack, in which the interaction among service providers and consumers for the SaaS, PaaS and IaaS service models are illustrated. As it can be seen, services are accessed via application programming interfaces (APIs) which are available over the Internet (e.g. Webservices). Although the cloud stack in Figure 2.1 is presented in a layered fashion, it might be possible that consumers situated on uppermost layers consume services offered by providers situated on lowermost layers. For example, a SaaS consumer may consume a SaaS service deployed on a platform/infrastructure leased to him/her (the SaaS consumer). Therefore, the SaaS consumer would have a relationship not only with the SaaS provider but also with the PaaS and/or IaaS provider.

The rights and obligation of both service consumer and provider involved in service purchase might be formalized by a Service Level Agreements (SLA) [WB11]. For example, at the SaaS level, the SLA established by Google Apps ⁹ and their consumers states that Google's obligation is to offer a service availability of 99.9% (i.e. the monthly uptime percentage). In case of contract violation, i.e. if Google does not meet its obligations (the monthly uptime percentage inferior to 99.9), the consumer will be eligible to receive service credits according to the level of violation. In the same sense, SLAs can also be defined at the PaaS (e.g. for Google AppEngine ¹⁰) and IaaS (e.g. for Amazon EC2 ¹¹) so as to give consumers compensations in case of contract violation.

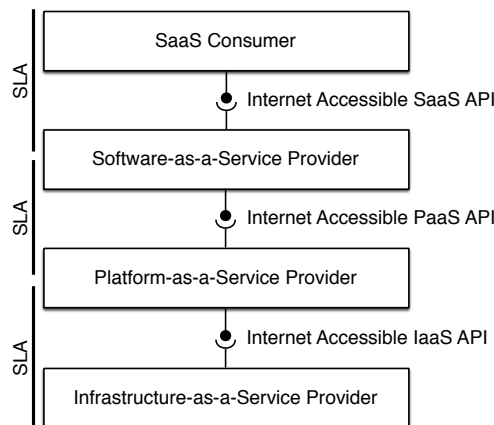


Figure 2.1: Cloud Computing Stack.

Although basic concepts of SLA is applicable to real world applications, there are several issues regarding the SLA management and cloud computing [BGC11, KL12]. Indeed, the dynamism of cloud environments along with the lack of access to metrics on the provider side make it hard to monitor and audit the QoS metrics and hence be able to detect violations.

⁴<http://cloud.google.com>

⁵<http://www.cloudbees.com/>

⁶<http://aws.amazon.com/ec2/>

⁷<http://aws.amazon.com/s3/>

⁸<http://www.windowsazure.com/>

⁹<http://www.google.com/apps/intl/en/terms/sla.html>

¹⁰<http://developers.google.com/appengine/sla>

¹¹<http://aws.amazon.com/ec2-sla/>

Deployment Models

The cloud computing deployment models are defined as follows:

- *Private Cloud*: the cloud infrastructure is dedicated exclusively to the needs of a single organization and it may be owned, managed or operated by the organization itself, a third-party organization or a combination of them.
- *Community Cloud*: the cloud infrastructure is dedicated to the needs of a community of organizations with common concerns. It may be owned, managed or operated by the organizations themselves, a third-party organizations or a combination of them.
- *Public Cloud*: the cloud infrastructure is owned, managed or operated by business, academic, government organizations, or a combination of them; and it is open to serve the open public.
- *Hybrid Cloud*: it is a combination of the three previous models.

2.1.3 Platform Virtualization

According to Smith and Nair [SN05], virtualization is a way of getting around constraints imposed by the interface specificities of each system component. The objective is to provide more flexibility for software systems by mapping “*its interface and visible resources onto the interface and resources of underlying, possibly different, real systems.*”. Singh [Sin04] defines virtualization as “*a framework or methodology of dividing the resources of a computer into multiple execution environments[...]*”, meaning that it enables the mutualization of resources by sharing them among several virtual machines (VMs) in an isolated manner. The most known form of virtualization is the platform virtualization, in which platform refers to the hardware platform and its physical resources such as CPU, memory, disk and network.

Virtualization (or platform virtualization) is particularly interesting for IaaS providers, since it brings more flexibility to the on-demand provision of resources. Indeed, in a single physical machine (PM), an IaaS can simultaneously serve several consumers in an isolated way, as if for each consumer was given a small single PM. A result of this physical resource mutualization is a better management of the resources utilization, which consequently may be more cost effective.

The virtualization is controlled by a system called virtual machine manager (VMM), also known as hypervisor (Figure 2.2). It is in charge of controlling several guest VMs that are hosted by the PM in which the hypervisor is installed. Each VM manages its own OS and applications. The hypervisor can be of two types: bare-metal hypervisor, which lays directly on the hardware and can serve as a host OS; and hosted hypervisor, which depends on a host OS.

A VM is nothing but a file, in which the modifications can be written (snapt-shotted), and which can be easily copied so as to create another VM instance (cloned) or transferred (migrated) to another PM. With this regard, most of hypervisors are equipped with VMs live migration capability, which allows migrating a running VM from one PM to another without having to switch it off before. This is particularly interesting to perform VM load balancing or server consolidation, that is, to concentrate the maximum number of VMs in fewest number of PMs so as to optimize the energy consumption of PMs [VAN08a, HLM⁺09]. However, if this feature is not used with moderation, due to its high overhead, it may cause serious problems of performance and thus affect the guest applications’ QoS [VKKS11, Str12].

There are several kinds of virtualization, such as OS-based virtualization [KW00]¹², application virtualization^{13 14 15} and desktop virtualization¹⁶. In this thesis, we make use of server

¹²<http://linux-vserver.org/>

¹³<http://www.vmware.com/products/thinapp/overview.html>

¹⁴<http://www.winehq.org/>

¹⁵<http://www.cygwin.com/>

¹⁶<http://www.realvnc.com/>

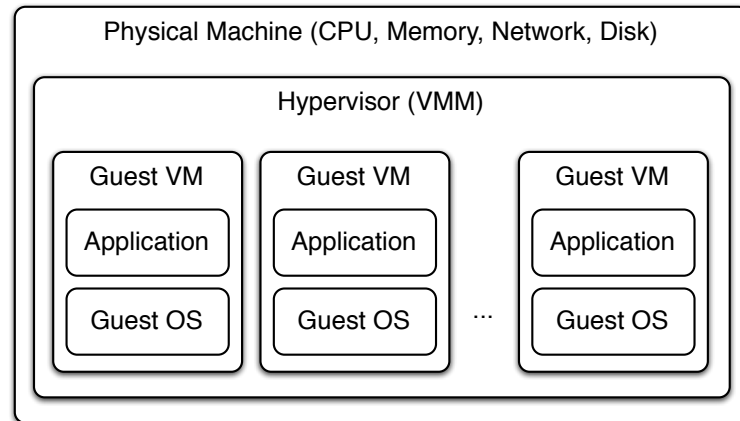


Figure 2.2: Platform Virtualization.

virtualization and thus we focus on the main virtualization models for this purpose, namely full virtualization and para-virtualization.

The *full virtualization* permits that a guest OS runs without any special adaptation for the hypervisor. Although it seems to be the ideal approach, mainly if the guest OS and applications are prepared without knowing in advance the target platform, it may be very costly in terms of performance. In fact, the full virtualization requires that every important characteristic of the hardware are emulated in details as if the guest OS were executed natively. Example of hypervisors that supports full virtualization include VMware¹⁷, Xen¹⁸, VirtualBox¹⁹ from Oracle, Hyper-V²⁰ from Microsoft and KVM²¹ from RedHat.

The *para-virtualization* overcomes the full virtualization performance issues by making VMs aware of the virtualization. The hypervisor instead of emulating all the main hardware devices, provides an API to access them. While it may turn the hypervisor more lightweight and thereby more performance effective, it comes at the expense of the needful for guest OS modification. There are implementations of para-virtualization by VMware, Hyper-V, Xen, KVM, among others.

2.1.4 Advantages and Drawbacks

The service utility model upon which cloud computing is based brings many advantages. Cloud computing may free consumers from the burden of maintenance at all levels, e.g. software, platform and infrastructure. This is particularly interesting when the business focus, although dependent on, is not related to IT, since the consumers become free to focus exclusively on their business rather than on other issues related to IT. Another advantage concerns the elasticity of cloud services. Because of its on-demand service model supported by techniques like virtualization, IaaS consumers are capable of easily scaling up and down so as to cope with very dynamic demands. Moreover, thanks to the pay-as-you-go philosophy of cloud service models, cloud consumers are given the chance to scale while paying only for what they actually consume, which can be more cost effective.

It is straightforward that all these advantages come at a price. Indeed, the lack of guarantees on data privacy and security [SK11] are examples of issues that arise in cloud systems. In fact, since consumers have no idea how their data are stored in cloud infrastructure or how safe the leased SaaS/PaaS/IaaS is, it becomes difficult to be assured about those attributes. The lack of

¹⁷<http://www.vmware.com>

¹⁸<http://xen.org/>

¹⁹<http://www.virtualbox.org/>

²⁰<http://www.microsoft.com/hyper-v>

²¹<http://www.linux-kvm.org/>

customization may also pose some problems. For instance, consumers' special requirements on hardware (e.g. a specific processor model) that may not be met by IaaS providers. Another example is the lack of customization in the context of multi-tenant SaaS applications (applications that are leased to multiple consumers at the same time). There might be a problem when some specific features that is desired by one consumer is not implemented on a SaaS [GTAB12].

2.2 Autonomic Computing

In order to cope with a variable environment which is intrinsic to cloud computing systems (cf. Section 2.2), it becomes imperative to rely on models that allows the system to react to the execution context so as to keep it running in an optimized way (in terms of QoS and cost). For this purpose this thesis relies on autonomic computing.

In this section, we describe the main concepts related to autonomic computing. Firstly, we introduce the term “autonomic computing”, then we provide some discussion about some properties inherent to autonomic systems. Finally, we describe the widely adopted autonomic computing reference model.

2.2.1 Definition

Autonomic Computing emerged from the necessity to autonomously manage complex systems, in which the manual human-like maintenance becomes infeasible. The term was first introduced in 2001 by IBM [Hor01] as a reference to the ability of the human autonomic nervous system to govern vital functions such as heart beat, body temperature, among many others, and hence discharging the human conscious brain of managing these tasks. The counterpart of the human autonomic nervous system in computing systems is therefore defined by IBM as systems that are able to manage themselves according to high-level goals provided by administrators [KC03].

2.2.2 Self-management

Self-management is the main characteristics of autonomic computing systems. It refers to the ability of systems to avoid human intervention by freeing administrators from details on the operation and maintenance to keep systems continuously running. The main properties of self-management evoked by IBM are self-configuration, self-optimization, self-healing and self-protection [KC03]. Following, we provide a brief definition of each one of them:

- **Self-configuration:** this property makes autonomic computing systems capable to configure themselves according to high-level goals based on the system and/or user needs. These high-level goals specifies which future states are desired, but not necessarily specifying how the transition from the current state to the desired one should be accomplished.
- **Self-optimization:** this property indicates that autonomic systems are able to constantly and/or pro-actively be in quest of optimizing the use of resources so as to improve the performance and QoS while minimizing costs.
- **Self-healing:** this property makes autonomic computing system capable of detecting, diagnosing and repairing localized problems. These problems may result from bugs in both software or hardware. An example of self-healing could be the replacement of a software component that does not work properly by another providing the same functionality but implemented differently.
- **Self-protection:** this property makes an autonomic system able to protect itself from malicious attack or cascading failures not repaired by the self-healing strategies. In addition, an autonomic system may be capable of anticipating problems in a proactive fashion by taking into consideration previous sensor data.

2.2.3 Autonomic Computing Reference Model

Figure 2.3 depicts the reference model proposed by IBM to realize autonomic computing [KC03]. Autonomic systems are defined as a collection of autonomic elements that communicate with each other. An autonomic element consists of a single autonomic manager that controls one or many managed elements.

A managed element is a software or hardware resource similar to its counterpart found in non-autonomic systems, except for the fact that it is adapted with sensors and effectors so as to be controllable by autonomic managers. Sensors gathers data from the managed elements, which allow the autonomic manager to monitor and perform changes on them via effectors.

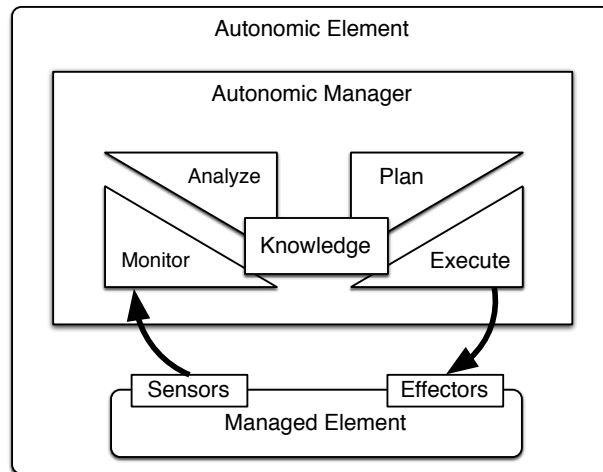


Figure 2.3: Autonomic Computing Reference Architectural Model Proposed by IBM [KC03].

According to [HM08], an autonomic manager is defined as a software component that, based on high-level goals, uses the monitoring data from sensors and the internal knowledge of the system to plan and execute actions on the managed element in order to achieve those goals. It is also known as a MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) loop, as a reference to the main architectural aspects of the IBM's reference model (Figure 2.3).

We now detail each task of the autonomic manager.

Monitoring

As previously stated, the monitoring task is in charge of observing the data collected by software or hardware sensors deployed in the managed element. The monitoring activity can be either *passive* or *active*. The passive monitoring gets information from the monitored system in a non-intrusive fashion. It means that there is no need for internal changes in the monitored system such as by adding or modifying the code (instrumenting it) to provide monitoring capabilities. On the contrary, the active monitoring refers to instrumenting the monitored system by for instance modifying the code to intercept function and/or system calls.

Analysis

The analysis task is in charge of finding a desired state for the managed element by taking into consideration the monitored data, the current state of the managed element, and adaptation policies. Policies are usually specified in terms of event-condition-action (ECA), goals and/or utility functions. ECA policies are defined by a set of tripled rules (when a given event occurs, and a given condition holds, then execute a certain action). This kind of policy does not require a planning task to effect the changes on the managed element. Goal based policies specify the characteristics

of desired states, but not how to perform the transition from the current state to them, which is done by the planning task. Finally, the utility function based policies allows one to quantify the level of usefulness of possible future states.

Planning

The planning task takes into consideration the current state and the desired state resulting from the analysis task to produce a set of changes to be taken effect on the managed elements. These changes may be organized in a ordered and constrained manner. For instance, there might be a set of changes that should be executed after the termination of other changes. Moreover, there might be changes that should be executed in a timely manner.

Execution

The execution task is responsible for effecting changes on the managed element, within the desired time and respecting the reconfiguration plan generated by the planning task. The execution is performed by relying on low-level actions which may impact several managed elements and may involve several machines.

Knowledge

The knowledge in an autonomic system assembles information about the autonomic element (e.g. system representation models, information on the managed system's states, adaptation policies, and so on) and can be accessed by the four tasks previously described. This information may come from diverse sources such as history of execution, observation on the system behaviour, log data, or even information added by human (domain experts, system administrators, etc).

2.3 Component-based Software Engineering

In environments that demand constant changes such as those of autonomic computing (cf. Section 2.2.3), it becomes imperative the programming language and architectural models support to perform changes on managed elements in a seamlessly way. Moreover, it is very important to have architectural support for dynamic reconfiguration, i.e., the reconfiguration performed at runtime. Hence, to this effect, this thesis presumes that SaaS applications are defined and constructed based on a component-based software engineering principles.

In this section we present the main concepts related to component-based software engineering. Then, we discuss how component-based architectures are appropriated for dynamic adaptation. Finally, we bridge the component-based software engineering and service-oriented architecture by presenting service component architecture, a component model whose objective is to support the construction, description, deployment and maintenance of service-oriented architecture based applications.

2.3.1 Definitions

Component-based software engineering (CBSE) is a branch of software engineering that seeks to provide means for the development of software as a composition of pre-fabricated and/or COTS (commercial off the shelf) software components [HC01].

Component Model

There are many definitions of *component* in the literature. In [SP97] a software component is defined as “*a unit of composition with contractually specified interfaces and explicitly context dependencies only.[...] A software component can be deployed independently and is subject to*

composition by third parties". A similar definition is given in [SGM02], where software component is defined as "a unit of independent deployment and third-party composition". Another definition by [HC01] states that component is "a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard". From the software reuse perspective [JGJ97a] defines components as "a type, class or any other workproduct that has been specifically engineered to be reused. (...) A component is cohesive and has stable interfaces".

All the definitions above converge towards the same characteristics. In order to be independently deployable, the component should be well separated from other components and encapsulate its implementation. In order to be third-party composable (or reusable), it should come with clear definitions (contracts) about what it provides and requires, that is, its interactions with the environment should be done via clear definitions of interfaces. The explicit context dependencies must specify the component's needs so that it can work properly. It refers to the context of composition and deployment, that is, the rules of composition and rules of deployment, installation, activation of components. In short, a component must conform to a component model, which specifies how the component should be constructed, how it should be deployed and how it should interact. COM/DCOM/COM+ (Component Object Model) ²² and .Net component model ²³ are examples of component models by Microsoft with the purpose to enable software components within Windows-Family OS to communicate. Sun has developed JavaBeans ²⁴ and Enterprise JavaBeans (EJB) ²⁵ to allow the componentization in both standalone and server-based Java programs. Finally, CORBA Component Model (CCM) ²⁶ is a component model maintained by the Object Management Group (OMG) ²⁷ that allows the implementation of distributed objects, in which services can be invoked regardless the object location, programming languages, operating system, communication protocols or hardware.

More recently, some component models with focus on the development of service-oriented architecture based applications have been proposed, such as open services gateway initiative (OSGi) ²⁸ and service component architecture (SCA) ²⁹.

Component-based Architecture

A software architecture defines the high-level structure of a software system, by describing how it is organized as a means of a composition of components [JGJ97a]. Generally an architecture description language (ADL) [MT00] is used to describe the architecture of a system. Although the diversity of ADLs, the architectural elements proposed in almost all of them follow the same conceptual basis [GMW00]. The main architectural elements are defined as follows:

- *Component*. A component is the most elementary processing or data storage unit. It may be simple (composed of only one component) or composite (composed of other components). It consists of two parts: the implementation and the interfaces. The implementation describes the internal behaviour of the actual component, whereas the interfaces defines how the component should interact with the environment.
- *Connector*. A connector corresponds to interactions among components. Roughly, it mediates an inter-component communication in diverse forms of interactions, such as procedure call or event broadcasting.

²²<http://www.microsoft.com/com>

²³<http://www.microsoft.com/net>

²⁴<http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/>

²⁵<http://www.oracle.com/technetwork/java/ejb-141389.html>

²⁶<http://www.omg.org/spec/CCM/>

²⁷<http://www.omg.org/>

²⁸<http://www.osgi.org>

²⁹<http://www.oasis-open.org/sca>

- *Configuration.* A configuration describes the structure and/or the behaviour of the component-based application. It corresponds to a directed graph of components and connectors describing the application's structure. The behavioural description describes how the interaction among components and properties evolve over the time.

It should be noticed that other elements like properties, constraints or architectural styles may also often appear in ADLs [GMW00]. For sake of scope limitation and brevity we omit further description on this regards.

2.3.2 Architectural-based Dynamic Reconfiguration

Dynamic architectures allows the modification of systems during their execution. It is particularly applicable in the domain of self-adaptive systems (cf. Section 2.2.3) in which systems should adapt to some context change while mitigating the interferences on their execution.

Reflection and Reflective Architectures

Reflective systems can be seen as good basis for dynamic architectures. Indeed, reflective systems are systems capable of reasoning on and modifying their own structure and/or behaviour [Smi84]. They are characterized by two capabilities: introspection and intercession. The first allows a system to observe itself, that is, to observe a representation of its structure and behaviour. The second capability corresponds to the possibility to change itself, that is, to modify its own structure and/or behaviour. According to [Mae87], a reflective system comprises two levels: (i) the base level, which corresponds to the implementation of what the system is actually indented to do; and (ii) the meta level, which reflects the base level implementation for parts of the system that are reified.

Reflective architectures act in a similar way to provide reflection capabilities (introspection and intersection) of software architectures [CSST98]. In a reflective component model, the base level consists of the actual components (with their provided and required interfaces). At the meta level, the components are equipped with control interfaces, which allow the introspection (e.g. observation of components, assembly, interfaces, connectors, and so forth) and intersection (e.g. creation/suppression of components, creation/suppression of connectors, etc).

Examples of reflective component models include FORMWare [MBC01, MBC03], OpenRec [WSKW06], Fractal [BCL⁺04], and its extensions Grid Component Model (GCM) [BCD⁺09] and JADE [BPG⁺09], for supporting distributed components in grid/cluster environments.

Dynamic Reconfiguration

Dynamic reconfigurations are reconfigurations in which it is not necessary to stop the system execution or to entirely redeploy it in order for the modification to take effect. As a consequence, the number interferences on the system execution is reduced and the availability is increased. Component-based architectures are very suitable for dynamic reconfigurations. Indeed, thanks to its native characteristics of modularity and encapsulation, it is possible to isolate the modifications so that the interference on the system execution is mitigated. However, in some cases, the dynamic evolution of system may not be achievable due to reliability reasons, meaning that dynamic reconfigurations may lead systems to inconsistent states and thus may not be able to deliver systems' expected functionality during all their life-cycle. In this regard, recent research work [LLC10] propose an transaction-based approach to address the reliability issue in dynamic and distributed component-based systems.

2.3.3 Component-based Software Engineering for Service-Oriented Architectures

This section discuss some limitations of CBSE when applied to nowadays software development and how it can be complemented by service-oriented architectures. Then we briefly describe Ser-

vice Component Architecture, a component model that focuses on the development of applications based on the service-oriented architecture.

Component-based Software Engineering versus Service-oriented Architecture

While CBSE is a very handy discipline that focus on the reuse, deployment and maintenance of software systems, it does not address a number of problems of nowadays software development such as interoperability, platform and protocol diversity, and distributed environments over the Internet [Has03]. Service-Oriented Architecture (SOA) aims at cope to these limitations. SOA is an architectural style in which services are the smallest functional/composition unit [Pap03], that is, SOA is an architectural style where applications are conceived as composition of loosely coupled, coarse grained, technology agnostic, remotely accessed, and interoperable (i.e. accessed via standardized protocols) services. In the other sense, SOA requires appropriate infrastructure to conceive, deliver and manage distributed applications based on the SOA principles [SMF⁺09]. Therefore, CBSE and SOA can be seen as complementary approaches.

Service Component Architecture

Service Component Architecture (SCA) is a component model for building applications based on the SOA principles. The first version of the SCA specification was published by a consortium of companies like BEA, IBM, IONA, Oracle, SAP, Sun and TIBCO. Since 2007, SCA has been maintained by the OASIS' Open Composite Services Architecture (CSA) section ³⁰.

SCA provides means for constructing, assembling and deploying software components regardless the programming language or protocol used to implement and make them communicate. To this end, the SCA specification consists of four main specifications: assembly model, component implementation, binding and policy framework.

The *component implementation specification* specifies how a component can be actually implemented in a certain programming language (e.g. another SCA component definition, Java, C, C++, Cobol, Spring, Business Process Execution Language).

The *assembly model specification* specifies how components should be defined and structurally composed. A component is specified in terms of implementation, services, references and properties (cf. Figure 2.4). The implementation points to the code that actually implements the component (e.g. Java Class). A service refers to a business function implemented by the component. A reference corresponds to a functional dependency on a service provided by another component. A reference and a service are wired together through a wire and they are both specified by either Java Interfaces or a WSDL description files. The component implementation may have properties whose values can be set within the component definition. A component can be defined as simple or composite, that is, composed of other components. In order for a service be accessible from outside the composite, it should be defined as a service promoted to composite service. Similarly, in order to access a service offered by a component located outside the composite boundaries, the reference in question should be promoted to composite reference.

The *binding specifications* specify how services are provided and/or how references are satisfied by describing which access methods or transport are allowed such as Java Connector Architecture (JCA) ³¹, Java Message Service (JMS) ³² and Simple Object Application Protocol (SOAP) ³³.

The *policies framework specifications* specify how components can deal with non-functional components in a transparent way. For example, in order for a service to be executed, it is required that the income arguments are first logged in the database. Instead of treating the logging service

³⁰<http://www.oasis-open.org/>

³¹http://download.oracle.com/otn-pub/jcp/connector_architecture-1.6-fr-oth-JSpec/connector-1_6-final-spec.pdf

³²<http://www.oracle.com/technetwork/java/jms/index.html>

³³<http://www.w3.org/TR/soap/>

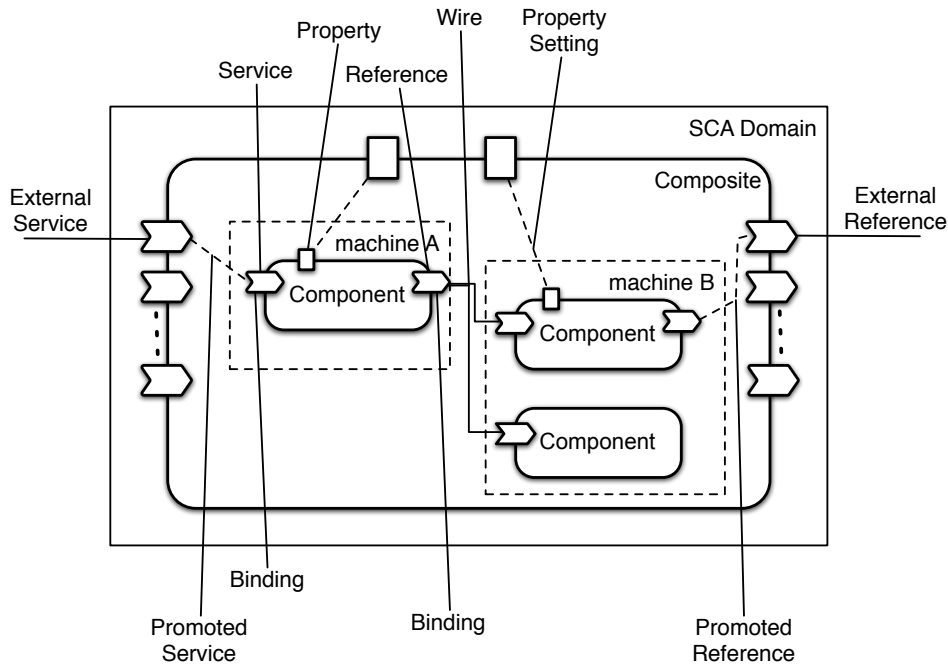


Figure 2.4: Service Component Architecture Concepts.

as a functional dependency (i.e. reference to a functional component), it is rather defined as a policy and therefore it does not change the way the component is implemented.

Deployment. Composites and components are deployed within a domain, which specifies an application provider boundary. That is, the details of composites defined in another domain are not visible. As a consequence, in order to guarantee interoperability, the communication between components located in different domains should be done in standardized ways such as Web Services. Nevertheless, two components of the same application may be executed in different SCA runtimes, or even in different machines, as long as they belong to the same domain.

SCA Dynamic Reconfiguration. There are a number of implementations of SCA such as Fabric3³⁴, Tuscany³⁵ and FraSCAti [SMF⁺09]. In this thesis, we rely on FraSCAti SCA runtime implementation, since it provides mechanisms for runtime reconfiguration. FraSCAti is developed on top of Fractal Component Model [BCL⁺04] and thus benefits of the reflexivity capability which allows dynamic reconfiguration of SCA application.

2.4 Constraint Programming

In the context of autonomic computing systems (cf. Section 2.1), in order to manage the dynamism of cloud systems (cf. Section 2.2) by self-optimizing their executions, it becomes necessary a method to concretely implement the ECA rules, goals or utility functions. ECA rules can be easily implemented with event listeners or well-known design patterns such as the observable pattern [GHJV95]. For goals and utility functions used to optimize the behaviour of the managed system, in terms of constrained (or limited) resource usage and performance, it is needful some times that the analysis or planning tasks is defined in terms of constraint satisfaction and/or optimization problems (CSPs/CSOPs). For this purpose, we rely on constraint programming (CP) for the modeling and solving of these kind of problems, which is a high-level, powerful, extensible

³⁴<http://www.fabric3.org/>

³⁵<http://tuscany.apache.org/>

and flexible paradigm for solving this kind of problems. This section presents the main concepts concerning CP by firstly providing a definition of CP. Then, we briefly describe how problems are modeled in CP and some search strategies used to solve the modeled problems. Finally, we provide some discussion about the drawbacks of CP and how they can be overcome.

2.4.1 Definition

Constraint Programming (CP) is a paradigm that aims to solve combinatorial search problems based on the definition of decision variables and constraints over them [RVBW06, RvBW07]. The basic idea is that the end-user states a set of variables and imposes a set of constraints and a general purpose solver tries to find a solution that meets these constraints. Examples of CP solvers include Choco [Tea10] and IBM CP Optimizer³⁶.

2.4.2 Modeling a CSP

In [CHO], a CSP is defined as a tuple (X, D, C) , where $X = \{X_1, X_2, \dots, X_n\}$ is the set of decision variables of the problem. D refers to a function that maps each variable $X_i \in X$ to the respective domain $D(X_i)$. A variable X_i can be assigned to integers values (i.e. $D(X_i) \subseteq \mathbb{Z}$), real values (i.g. $D(X_i) \subset \mathbb{R}$) or a set of discrete values (i.e. $D(X_i) \subseteq \mathcal{P}(\mathbb{Z})$). Finally, C corresponds to a set of constraints $\{C_1, C_2, \dots, C_m\}$ that restrain the possible values assigned to variables. So, let (v_1, v_2, \dots, v_n^j) be a tuple of possible values for subset $X^j = \{X_1^j, X_2^j, \dots, X_n^j\} \subseteq X$. A constraint C_j is defined as a relation on set X^j such that $(v_1, v_2, \dots, v_n^j) \in C_j \cap (D(X_1^j) \times D(X_2^j) \times \dots \times D(X_n^j))$.

For example, let us suppose that one wants to construct a rectangle whose perimeter is less than $16cm$. In order to model this as a CSP problem, we need to specify two variables $X = \{H, W\}$ representing respectively the rectangle's height and width. A priori, these variables can be assigned to any positive integers, that is, $D(H) = D(W) = \mathbb{N}^+$. Finally, in order to assure that the perimeter is lower than $16cm$, the following constraint should be defined $2H + 2W \leq 16$.

2.4.3 Solving a CSP with CP

In CP, the user provides a CSP problem and a CP solver takes care of solving it. Solving a CSP (X, D, C) is about finding a tuple of possible values (v_1, v_2, \dots, v_n) for each variable $X_i \in X$ such that all the constraints $C_j \in C$ are met. For example, $\{(1, 1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2), (2, 3), \dots, (4, 4)\}$ are possible solutions for the rectangle CSP presented in the previous section. In the case of a CSOP, that is, when an optimization criterion should be maximized or minimized, a solution is the one that maximizes or minimizes a given objective function $f : D(X) \mapsto \mathbb{R}$. An objective function for the rectangle CSP could be *maximize* $f(H, W) = H * W$, which means that the solution found must be the one that maximizes the rectangle's area.

CSP are usually solved with search algorithmic approaches, which can be either complete (systematic) or incomplete (non-systematic). The former performs systematic search (e.g. backtracking or branch-and-bound) that guarantees that a solution is found, if there exists at least one, whereas the latter performs a sort of local search and therefore should not be used to show that such a solution does not exist or to find an optimal solution. Nonetheless, non-systematic might take less time to find a solution (if there is any) and an approximation of the optimal solution.

2.4.4 Drawbacks and Overcomes

It is straightforward that the main advantage of CP is its declarative characteristics, i.e. users state constraints and an engine solves it. Hence, users are freed of the burden of thinking on and building solvers for the each CSPs. However, it may be a very hard task to correctly model a

³⁶<http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/>

CSP in a way that it can be solved in an efficient fashion, since CSPs are generally NP-Complete [RvBW07]. Moreover, CSP can be over-constrained.

Therefore, it becomes imperative to know how to take advantage of CP features so as to be able to solve problems in polynomial time. Symmetry prevention and soft constraints are example of such features. Symmetry may be introduced along with the model (e.g. allowing order permutation in a set of variables) and may make the search waste a lot of time visiting symmetric solutions. This kind of problem can be avoided in the model itself (e.g. by adding constraints to avoid symmetric solutions)[FFH⁺02] or internally in the search implementation [CGLR96]. Soft constraints [Sch92], in turn, allows one to specify preferences instead of hard constraints as well as ways for finding optimal solutions according to them.

Moreover, classical search approaches like backtracking or branch and bound might also be mixed with other techniques from operation research or artificial intelligence in order to decrease the solving time. For instance, one can design a heuristics that helps the solver ordering variables so as to outperform the backtracking search [GMP⁺96]. Finally, CP solvers can also benefit of inference, by propagating information from one constraint to others [RVBW06]. As a result, the search tree can be pruned and consequently the search time can be reduced.

2.5 Summary

This chapter presents the main concepts and technologies related to this thesis work. We first discourse shortly on cloud computing, which can be thought as a on-demand provisioning model for computing resources that is convergence of old concepts like grid and utility computing. Also, the main cloud service models (Infrastructure-as-a-Service, Platform-as-a-Service and Software-as-a-Service) and the way they interact were described. In addition, we described the deployment models of cloud computing, namely private, community, public and hybrid followed by an introduction on virtualization, an important technology that enables the implementation of flexible cloud infrastructure. Lastly, we provided some discussion about advantages (e.g. the flexibility in terms of scalability), and drawbacks (lack of guarantees of privacy and security).

Secondly, we gave an introduction on autonomic computing, which provides a reference model to enable software systems to be self-manageable. In other words, it fits very well the dynamic systems such as those based on cloud computing. We presented the four properties of a self-manageable system, namely self-configuration, self-optimization, self-healing and self-protection. Finally, we described the autonomic reference model which is based on a MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) loop.

Then, we briefly described component-based software engineering, which refers to a sub-discipline of software engineering with a focus on the construction of application as a composition of pre-fabricated components. We provided a short introduction on dynamic reconfiguration based on software architectures, which is particularly interesting for systems that needs to constantly adapt to environment changes, such as self-manageable cloud systems. At last, we pointed out some limitations of component-based models for some kinds of scenarios and how components meet service-oriented architectures to overcome such limitations.

Finally, we gave a very short introduction on constraint programming, which is a tool for solving constraint satisfaction and optimization problems based on variables and constraints. In this work, it is used to perform the analysis task of the autonomic reference model so as to try to find out the optimal state for a managed system's runtime context. We presented which are the elements of a constraint programming problem model as well as some discussion about the solving strategies. Lastly, we highlighted some disadvantages of constraint programming while providing some overcome elements.

Related Work

The objective of this thesis work is to deal with the problem of energy consumption in cloud infrastructures by taking into consideration not only the way the infrastructures are configured but also the application-level architectures and how they impact the applications' Quality of Service (QoS) and functionalities. This chapter presents a selection of publications which are related to the energy or resource management at infrastructure and application layers in isolation or as single cross-layered management unit. In addition, we provide some discussion about some approaches that consider the coordination of autonomic managers distributed across layers. Finally, we summarize the presented approaches by comparing them with regards to a set of qualitative attributes.

Contents

3.1	Energy/Resource/QoS Management at Application Level	33
3.1.1	QoS and Energy Management in Multi-tier Web Applications	34
3.1.2	QoS and Energy Management in Component/Service-Based Applications	35
3.2	Energy/Resource/QoS Management at the Infrastructure Level	37
3.2.1	Hardware Tuning	37
3.2.2	Virtualization	38
3.3	Cross-layered Energy/QoS-aware Management	40
3.3.1	Single Autonomic Management	41
3.3.2	Multiple Autonomic Management	44
3.4	Summary	50

3.1 Energy/Resource/QoS Management at Application Level

In this section we present a body of work related to the QoS or energy/resource management in distributed applications. More precisely, in this section, we discuss about some work dealing with energy consumption from the application point of view. We consider those work in which either the energy consumption itself or the incurred resource consumption (and by transitivity the energy) is taking into consideration as a counterbalance to the QoS. First, we focus on some work in the context of multi-tier web applications, which are applications following a client-server architectural style segregated into several tiers. That is, components from one tier communicate with components in subsequent tier in a client-server manner. Then we provide some discussion about some work in the context of component or service based applications.

3.1.1 QoS and Energy Management in Multi-tier Web Applications

Adaptive internet services through performance and availability control [AB10]

Description: The work propose MoKa, an approach to improve the performance in multi-tiered applications (cf. Figure 3.1.1). It takes into account two levels of configuration (architectural and local) in order to improve the system performance under charge. The architectural configuration means the cost in terms of physical machines (PMs) and the local configuration means the Multi-programming Level to define the maximum number of concurrent clients the servers can admit. The approach makes use of Mean Value Analysis (MVA) [RL80] queuing model to model the performance (i.e. the latency and abandon rate) in each tier. Based on those parameters and on the configuration cost, an objective function is provided (cf. Figure 3.2). Two algorithms are therefore provided: (i) one to implement the model which is used to predict the latency, cost and abandon rate based on a given configuration and workload; (ii) and other to find the optimal solution based on the objective function. The adaptation may occur in two levels: (i) degrade the application QoS by increasing the abandon rate (admission control); (ii) or by adding and suppressing PMs, resulting in energy savings.

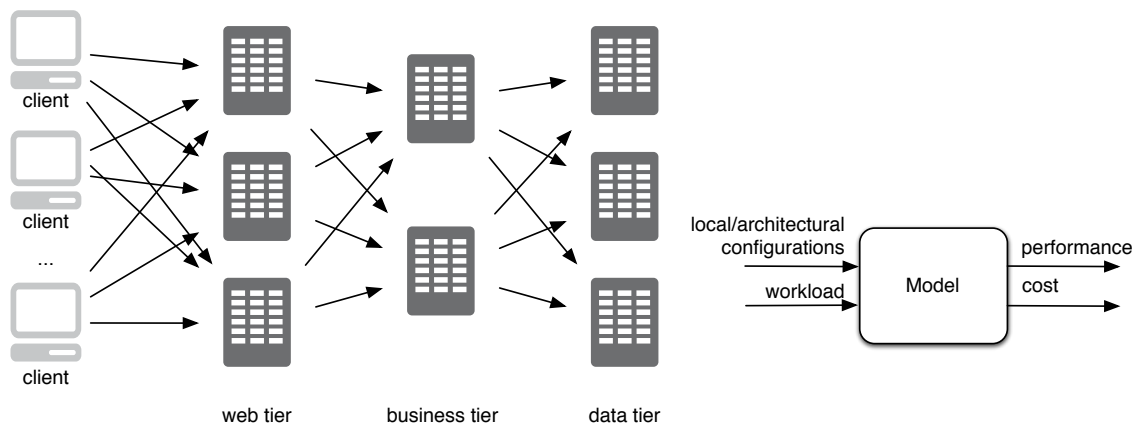


Figure 3.1: Multi-tiered Web Application [AB10].

Figure 3.2: The MoKa Web Application Model [AB10].

Discussion: By suppressing useless PMs, it is indeed possible to save energy. However, in resource sharing environment such as in cloud computing, that is, when several applications share the same computational infrastructure, it becomes hard to estimate the exact amount of resources necessary to host each application. An extension of this work for cloud environments was proposed in [KL12], in which case, a virtualized environment is considered to enable the resource mutualization in single PMs. There are a number of approaches for dynamic resource scaling based on Service Level Agreements (SLAs) [WB11, CMKS09]. The problem still remains due to the lack of visibility of the underlying infrastructure. In fact, one does not know how VM instances are placed on the infrastructure and because of that it is difficult to know how the decisions at the application level will impact the underlying infrastructure.

Energy-aware scheduling of service requests in multi-tier environment: Analysis and evaluation [Fer11]

Description: The work proposes an energy and QoS-aware scheduling algorithm for multi-tiered web applications. It also relies on MVA to model the scheduling problem. To this end, it is proposed a number of Key Performance Indicators (KPI), such as response time, and Green Performance Indicators (GPI), such as service energy efficiency. The indicators' threshold values are defined within a SLA. In order to define the KPI and GPI, a server power model is proposed,

in which there are three server classes: slow, medium and fast. For each class, it is considered five server power states: idle, low normal, normal, high normal and heavy. Hence, in order to estimate the service response time, the resource capacity as well as the workload are taken into consideration, whereas the service energy efficiency is given by the service time and the current server power state. Finally, the algorithm aims at maximizing the service response time while maximizing the service energy efficiency.

Discussion: The approach is similarly to MoKa. What distinguishes them is that in here the focus is on the energy consumption and QoS whereas in MoKa is on the minimization of resource usage and maximization of QoS. Apart from that, the limitations remain the same, that is, to transpose the approach in a cloud-based environment (virtualized), in which VMs can be created, destroyed at any time. As a consequence, the energy consumption issue should be tackled also at the infrastructure layer, since it might depend on how the applications are deployed on the PMs.

3.1.2 QoS and Energy Management in Component/Service-Based Applications

Energy-Aware Design of Service-Based Applications [MFKP09]

Description: The authors proposed an approach for developing service-based applications in an energy-aware manner. First, a characterization of the service energy consumption takes place along with other classical QoS dimensions such as response time and price (which is calculated based on the values of response time and energy efficiency). This characterization is based on a previous classification of the server in which the services will be executed. Specifically, a server can be slow, average and high with respect to their power of processing. In addition, servers can be running on idle (0-12%), normal (13-67%) or burst (68-100%) states at a given period of time. For each combination server class/state, there is a different power consumption value associated. The energy efficiency dimension is then proposed based on the ratio of the energy consumption of the service executing on normal state and the total energy consumed by the service. The objective is to choose the most appropriate service for each process activity in a business process that satisfies a set of local (task-level) and global (process-level) constraints. The model and the solver used were extended from those proposed in [AP07].

Discussion: Although the approach proposed in that work is able to dynamically choose a service implementation that is at the same time QoS and energy compliant, it is not clear how the choices made at selection time may impact the underlying infrastructure over the time. In fact, because a service execution time may vary according to the number of clients, it is hard to ensure a static response time for any workload. Moreover, the approach provides a way of choosing services' implementation during runtime, however, it does not tackle the dynamic deployment at application level nor resource management at the infrastructure which is imperative for highly dynamic environments such as in cloud-based systems.

Self-adapting Service Level in Java Enterprise Edition [PDPBG09]

Description: The work proposes a QoS/resource-aware self-managing approach for component-based Java Enterprise Edition applications. It basically consists of an implementation of an autonomic manager, which is composed of two thresholds (under-load and overload) for each resource. The adaptation system works in two modes: regulation and calibration mode (cf. Figure 3.3). In the regulation mode, the autonomic system reacts to overload situations by choosing the most effective adaptation not yet applied. Upon under-load situations, it selects among the already adaptations applied the most effective one to unapply. After each regulation, the system goes to the calibration mode, which measures the impacts on resource usage of the just-applied adaptation and then returns again to the regulation mode or re-applies an adaptation in the case of severe overload. The adaptations are chosen based on the efficiency of applying the adaptation, which

is given based on QoS level values provided by designers and based on observed values of past regulations, also called adaptation gains. The adaptation gains are estimated during the calibration mode and calculated based on the ratio of the resource usage before the calibration mode to the usage during the calibration mode. In order to isolate only the tasks whose execution are involved in the adaptation, the approach relies on a profiling technique called statistical sampling [GKM82]. It consists in periodically capturing stack calls in active threads in the system. In this case, component stacks are captured from call stacks and an execution pattern is established along with the corresponding resource consumption, for each resource. That way, it is possible to have more refined estimations on the resource consumption.

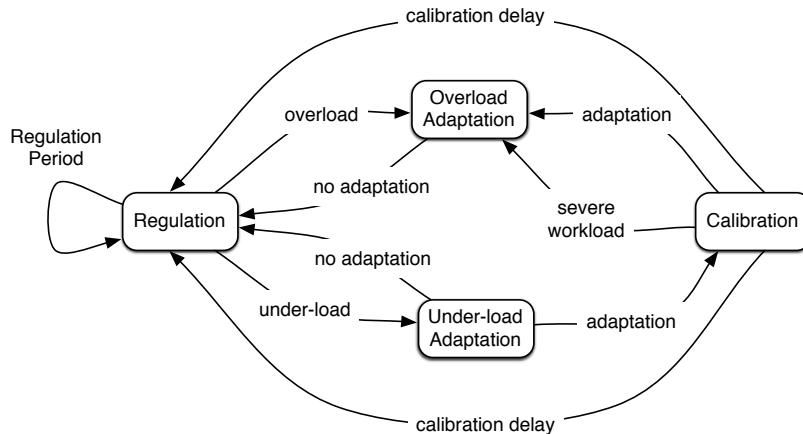


Figure 3.3: Autonomic Adaptation Design [PDPBG09].

Discussion: Although it is possible to estimate the impact on the resource consumption due to application execution, the approach relies on a static resource allocation. Indeed, the objective is to cope with underloaded and overloaded situations by respectively upgrading or degrading the application service level. It would be interesting to see how such a work could work in a resource elastic environment such as a virtualized one, in which both the service-level and the resource capacity could be self-adapted.

Enhancing a QoS-based Self-adaptive Framework with Energy Management Capabilities [PPMM11]

Description: This paper proposes a self-adaptive framework based on the trade-off QoS and energy consumption of service-based applications. The adaptation plans correspond to degrading/upgrading the servers frequencies or switching on/off servers to accommodate the workload or to reduce the energy waste. For this purpose, an off-line Stochastic Petri Nets (SPN) [Haa02] model on servers transient states was proposed. The objective is to analyze some variables such as the servers frequencies and power consumption for each phase (startup, shutdown, standby, maximum) and to produce as output the number of requests the server is able to respond as well as the calculated energy consumption for the corresponding number of servers and frequencies. Another SPN models the adaptation plan behavior by showing each possible configurations previously generated and the transitions allowed from the current configuration to another.

Discussion: Although the model seems to be complete, the work does not present any validation. Furthermore, the approach does not take into consideration a multiple application environment, which would require multiple autonomic managers and the appropriate management of them.

Composite SaaS Placement and Resource Optimization in Cloud Computing using Evolutionary Algorithms [YT12]

Description: The work proposes an approach that copes with the context variability at the application level, by considering the Software-as-a-Service (SaaS) application as an aggregation of computing and data components that can be spread out in different hosting units (VMs) or consolidated so as to minimize the number of VMs, as illustrated in Figure 3.4. In fact, it makes an analogy between the VM placement problem, in which VMs are placed onto PMs, and the component placement problem, in which computational and data components are placed onto VMs. The objective is two-fold: (i) to determine the initial component placement based on the estimated execution time; and (ii) to optimize the resource usage. Both problems are instances of well-know combinatorial problems in the context of resource management and automation and thus a solution based genetic algorithms is presented. The solution outperforms in comparison to a classical heuristic based approach (FFD).

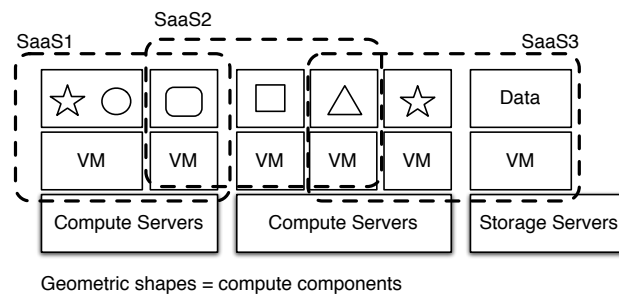


Figure 3.4: Illustration of Three Composite SaaS [YT12].

Discussion: Although the proposed work is capable of dealing with applications based on components, it does not mention anything about functional/non-functional adaptation in order to cope with resource restrictions or to fit in an energy compliant context. Moreover, despite the adaptation regarding the placement of components takes into consideration the resource usage, the impact on resource usage and consequently on the energy consumption at the infrastructure level may depend on how components/VMs are hosted on PMs.

3.2 Energy/Resource/QoS Management at the Infrastructure Level

This section presents some discussion about a selection of publications related to the management of QoS, energy or resource at cloud infrastructures. As the hardware management is out of scope of this thesis, we briefly cover some performance/energy management approaches that can be carried on at the hardware level. Then, we present some work that leverages virtualization to achieve energy efficiency in cloud data centers.

3.2.1 Hardware Tuning

One of the main challenges in making software systems more energy efficient is to face the lack energy proportionality inherent to nowadays hardware [BH07]. Modern hardware mitigates this problem by providing multiple configurable execution states (often known as c-states). For instance, power savings are achieved by applying techniques like Dynamic Voltage/Frequency Scaling (DVFS)[LWYH09]. The basic idea is to establish a trade-off between the processor performance and the current processor frequency and activated cores. That means that the lowest the processor speed (frequency) the lowest is the power consumption and performance. In DVFS-enabled machines, it is possible to dynamically change frequency (or power) states so as to adapt

to the current workload. Similarly, in [GWBB07] some power gains are obtained by turning-off components of hard disks.

3.2.2 Virtualization

EnaCloud: An Energy-saving Application Live Placement Approach for Cloud Computing Environments [LLH⁺09]

Description: EnaCloud is a framework to manage VM workloads with a focus on the energy efficiency by performing dynamic VM placement in cloud environments. The approach deals with three kinds of events: (i) a workload arrival, that is, a new VM creation; (ii) workload departure, i.e. a VM suppression; and (iii) workload resizing, which corresponds to a VM's resource resizing. Upon the occurrence of any of the above mentioned events, an heuristic-based algorithm generates a placement scheme containing a set of workload insert/pop (VM creation/suppression) and VM migration actions in a way the number of migrations are minimized and total number of PMs necessary to host the VMs are minimized, which may consequently improve the energy efficiency. The problem is an instance of a bin packing problem and is solved by an heuristic-based algorithm that relies on a composition of first-fit decreasing (FFD) and best fit decreasing (BFD) bin packing algorithms [JGJ97b]. The work shows that the heuristics outperforms these two algorithms when used in isolation with regards to number of migrations, number of PMs and energy consumption.

Discussion: Dynamically replacing VMs any time an event of the type VM arrival, departure or resizing occurs is certainly in theory the most energy efficient approach. However, technically, it may increase a lot the management time when several VM arrivals events occurs or if a number of migrations should be performed to accommodate the new incomer VMs.

Entropy: a Consolidation Manager for Clusters [HLM⁺09]

Description: Entropy is a virtualization-based resource manager for homogeneous clusters. More precisely, it relies on the consolidation technique to reduce the number of active PMs. The manager consists of a MAPE-K loop to autonomously and dynamically react to changes in the context (e.g. the resource usage of VMs). So, the monitoring task periodically gathers information about VM resource consumption such as the percentage of CPU utilization. The analysis task is in charge of solving the virtual machine placement problem, whereas the planning task is responsible for the virtual machine replacement problem. The first problem is an instance of the multi-dimensional bin-packing problem [JGJ97b] and it takes the running VMs with their respective resource usages and tries to pack them in the fewest number of active PMs so that the inactive ones can be afterwards switched off. The second problem tries to find a replacement plan for those VMs that should be migrated from one PM to another. This plan takes into consideration the dependencies among VMs and their respective migration cost, which is basically in function of the amount of memory allocated to the VM. Finally, the execution task executes the plan produced by the planning task. Both problems are modeled and solved with constraint programming [RVBW06].

Discussion: Although Entropy does not focus directly on the energy consumption of PMs, it is straightforward that by dynamically placing the VMs in a way the number of PMs can be minimized, it is possible to turn off the unused PMs and therefore save energy. However, optimizing the infrastructure by dynamically migrating (performing live migration) can be really costly in terms of performance [VKKS11, Str12] and hence can seriously degrade the QoS of applications hosted on the infrastructure.

Energy Efficient Resource Management in Virtualized Cloud Data Centers [BB10]

Description: The work proposes an approach whose objective is to optimize the trade-off between energy consumption and performance. The basic idea is to provide mechanisms of energy

efficiency by continuously optimizing the VM placement, while keeping a certain level of QoS and therefore honor the SLAs previously established. The approach relies on an hierarchical architecture (cf. Figure 3.5), which is composed of three kinds of components: dispatcher, global managers and local managers. The local manager reside on each PM along with the hypervisor. Based on monitored data on the resource consumption, it chooses VMs that should be migrated (6). It sends information about the PM's resource utilization and the VMs chosen to be migrated to the global manager (4). In addition, it performs commands to resize VMs, apply DVFS or turn on/off the PM. The global manager, in turn, it manages a set of PMs and processes their respective local managers' messages (5). Also, it applies a distributed version of an heuristic-based multi-dimensional bin packing algorithm (3). Finally, the dispatcher is responsible for dispatching the new clients requests (1) on VMs among the global managers (2), which shares the information about the VMs to be allocated. The VM placement optimization takes place in three stages: reallocation of VM considering the computational resources (e.g. CPU and RAM), the virtual network topology optimization, and the thermal optimization (based on PMs temperatures). These optimizations are computed separately and combined for a global solution. For the first stage, the choice for VMs to be migrated is based on upper and lower CPU utilization thresholds on the hosting PMs. If the current utilization exceeds the upper threshold, a VM is chosen to be migrated, whereas if the current utilization is lower than the lower threshold, all the VMs are chosen to be migrated. This information is sent to the global manager which issues the migration commands.

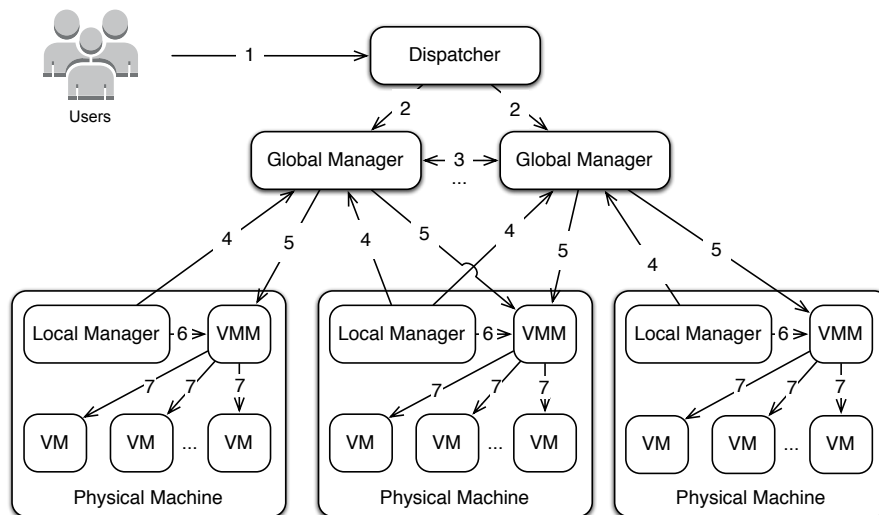


Figure 3.5: Hierarchical Architecture of Virtual Machine Management System.

Discussion: The approach shows important gains in energy consumption due to the server consolidation. However, similarly to [HLM⁺09], the live migration may significantly affect the system performance, which may result in QoS degradation. Although one of the objectives presented is to minimize QoS degradation and consequently SLA violations due to excessive number of migrations, the approach considers SLA violation failures in the requests for VMs (an Infrastructure-as-a-Service consumer that does not succeed in getting a new VM), instead of performance degradation on the running VMs. Moreover, no details about the strategies for minimizing migration operations is provided.

Snooze: A Scalable and Autonomic Virtual Machine Management Framework for Private Clouds [FRM12]

Description: The work presents Snooze, a framework for VM management whose focus is on the scalability, fault-tolerance and energy efficiency. Scalability because the authors claim to be capable of managing a huge number of VMs and PMs. Fault-tolerance because it is built based on a decentralized and hierarchical model and thus avoiding single points of failure. And energy efficiency because it considers the energy footprint of each possible configuration instead of only the amount of inactive PMs. The placement is also modeled as a multi-dimension bin-packing problem and is implemented by an algorithm based on ant colony optimization [FRM11] and outperforms classical greedy algorithms such as FFD and has potential to be solved in a distributed manner. The energy efficiency is achieved by packing the VMs in the fewest number of PMs and decreasing the power states of underload PMs. Similarly to [BB10], the framework architecture is composed of three main distributed and hierarchically-organized components: (i) local controllers, which are presented at the physical layer, i.e. each one is deployed in the PM which it is in charge of controlling. (ii) group managers, which are presented at the hierarchical layer, i.e. it is in charge of guaranteeing the fault-tolerance of a group of local controllers. It is also in charge of scheduling the new incoming VMs via diverse policies (e.g. round robin) as well as optimizing and planning periodically the VM placement; and (iii) group leaders which encompass a subset of group managers by keeping a summary of their information. It is also in charge of dispatching client requests on VMs. All the components are equipped with failure recovery based on heartbeat messages exchanged in a hierarchical way so as to achieve fault-tolerance.

Discussion: Unlike Entropy, Snooze is able to manage the existing VMs as well as to schedule the incoming ones, while guaranteeing fault-tolerance of management components, which is particularly interesting. However, like Entropy, the energy optimization relies on the consolidation, which may result in several VM migrations actions, followed by PMs shutdowns. As previously remarked, migrations may impact the hosted applications QoS and thus should be performed in a sparse fashion.

Other related work

There are countless work on the energy consumption in data centers by leveraging virtualization technologies. However, for brevity reasons we discussed only about some of them. Other work on the dynamic placement of VMs were proposed in [BKB07, VAN08b, SKZ08, RJQ⁺10, SZwL⁺11, DSV⁺11, GBF⁺12]. In [LBMN09], it was proposed a energy-aware cross-data center approach for load distribution. In [QLS12], the authors proposed a collaborative and decentralized extension of Entropy [HLM⁺09] to allow to support the management of tens of thousands of VMs. Likewise, [BDNDM10] proposed an algorithm nature-inspired to provide scalability and energy efficiency in large-sized data centers.

3.3 Cross-layered Energy/QoS-aware Management

In this section we discuss some work related to the management of energy/QoS in both levels of the cloud stack: infrastructure and application. We consider works whose self-management decisions are taken from several perspectives and impacts several layers of the software/infrastructure stack. We first discuss some work in which the cross-layered management is performed by a single management unit. Then we discuss some work that utilizes multiple management units that need to be synchronized and coordinated in order to achieve their goals.

3.3.1 Single Autonomic Management

Energy-Aware Autonomic Resource Allocation in Multi-Tier Virtualized Environments [APTZ12]

Description: The work proposes an energy-aware resource allocation approach for multi-tier web systems. The objective is to provide web services that achieve their SLA-based QoS requirements while tackling at the same time a varying workload and the energy consumption due to the infrastructure. To this end, the authors propose an autonomic framework that deals with several energy-related sub-problems across multiple layers of the IT infrastructure. In fact the work extends the cross-layered idea from a previous work [ACL⁺08] by dealing with several problems in a unified way. The approach assumes that the application follows a multi-tiered architectural style, in which each tier component communicates with a subsequent tier component in a client-server way (cf. Figure 3.1.1). A partition of physical resources (CPU and RAM) capacities, distributed over several VMs, can be allocated to components (capacity allocation) so they can be executed in parallel for load balancing and scalability purposes (load balancing problem). VMs instances are placed in PMs so as to reduce the number of PMs necessary to host all VMs (placement problem). Finally PMs can be switched on/off or have their CPU frequencies adjusted so as to save energy (i.e. frequency scaling or server switching problems). These problems are solved in two different time scales: short-term and long-terms. Short-terms refers to low-overhead reconfigurations and thus can be performed quite often (e.g. every 5 minutes). Load balancing, capacity allocation and frequency scaling are example of low-overhead problems and thus are considered as a short-term planning. Long-term planning, instead, deals with problems whose impact on the system is very high, such as switching on/off PMs. All the variables involved in those problems are considered together as they were the same optimization problem. In order to solve it in an efficient manner, a heuristic is proposed.

Discussion: Although the authors mention the possible utilization of a hierarchical model for controlling several managers, the proposed framework is a centralized controller in which variables of problems across several layers are shared. For that reason, it becomes difficult to apply this solution in the multi-vendor context such as in cloud computing environments, where each SaaS vendor may have its own autonomic manager which, in turn, may interact with Platform-as-a-Service (PaaS) or Infrastructure-as-a-Service (IaaS) autonomic manager in a seamlessly way. Moreover, as the number of different application providers (in a multi-application scenario) increases, it may become difficult to deal with all applications specificities within a single autonomic manager in a feasible manner.

Performance and Power Management for Cloud Infrastructures [VTM10]

Description: The authors propose a multi-application approach for the performance and power optimization in cloud infrastructures. The objective is to determine how much resource should be allocated to each application and how these applications should be placed in the infrastructure so as to optimize the energy consumption. To this end, a two-level framework is proposed, in which at the application level, for each application, a performance model along with an utility function are defined to express the degree of usefulness of a given response time. In addition, an application scaler is used to distribute the application requests over several VMs instances. At the infrastructure level, the global resource manager is basically composed of a VM provisioning manager and a VM placement manager, which are both modeled with constraint programming (CP) techniques. The former is in charge of determining the right amount of resources (in terms of VMs) for each application, whereas the latter takes as input the new provisioned VMs resulting from the first manager to optimize the placement of all VMs so as to minimize the number of required PMs and thus the power consumption. These two managers are periodically executed in a sequential way, i.e. the VM placement manager is executed only if there is a difference between the current number of VMs and the number of VMs resulting from the provisioning

manager. Hence, the result of the execution of the VM provisioning manager is a set of VMs to be requested/release. The VM placement manager, in turn, produces a set of create/destroy/migrate VMs and power on/off PM.

Discussion: The approach is considered multi-layer since it considers details about each application (performance models/utility functions) as well as about the infrastructure (current placement VM/PM) to take a global decision that involves both the application (in terms of amount of resources allocated) and the infrastructure (the new placement VM/PM). In this case, applications and infrastructure definitions are tightly-coupled, which is not very applicable to (public/hybrid) cloud infrastructures, in which applications does not have any details about the infrastructure and vice-versa. Another issue is due to the fact that it deals with all applications and infrastructure specifications to solve the same optimization problem may be not be feasible in as the number of applications increases. In that regard, the validation shows only a small experiment with three applications over an infrastructure with three PMs. Finally, migration operations may be performed any time the provisioning manager requests more VMs. We believe that this kind of operation should be performed in higher time scales, on pain of affecting applications' QoS.

SAFDIS: A Framework to Bring Self-Adaptability to Service-Based Distributed Applications [GDA10]

Description: SAFDIS (Self-Adaptation for DIstributed Services) is a framework for self-adaptation in multi-level distributed applications. The problem of adaptation is dealt in a traversal way, from the application to the infrastructure, in the sense that applications and infrastructure are monitored and thereafter the adaptation takes place at the upper level (applications) considering the current state of lower levels (infrastructure). It relies on a MAPE-K control loop which is implemented in a component-based architectural way. The monitoring gathers information provided by probes that are deployed along with the distributed system components. The analysis task is able to take short-term and long-term decisions as well as to distribute the analysis function, which may improve the performance of the analysis task. Given an adaptation strategy produced by the analyzer, the planning task may utilize several planning algorithms to generate the schedule of actions to be executed in a cross-layered manner. The actions are defined in an abstract way, that is, not considering the target technologies. The execution engine matches concrete actions that corresponds to the abstract ones, the execution context and imposed constraints. In [DPM12], the framework was applied to perform adaptations in the context of SOA-based applications hosted by SOA-based operating systems. The authors also showed how the framework can be used in the context of cloud computing [DAB11]. In that case, the adaptations may take place in the three main layers of the cloud stack, namely software, platform and infrastructure (cf. Section 2.1.2). For instance, at the SaaS level, a sensor may detect that an application hosted by a given resource is overloaded. The framework then tries to migrate the application from one resource to a more powerful one, which may involve monitored data and actions at all levels.

Discussion: The work proposes a cross-layered approach to adapt applications and infrastructure. However, dealing with all adaptation problems in a single autonomic manager may lead to some issues regarding the inter-layer isolation and loose-coupling characteristic, which is so common in cloud systems. For instance, details about the infrastructure is often hidden to the IaaS consumers in the case of a public/hybrid cloud. The same happens at PaaS and SaaS levels. Furthermore, as the number of managed applications/platforms increases, it becomes more and more difficult (in terms of performance) to manage applications, platforms and infrastructure via a unique cross-layered autonomic manager.

Optimized Management of Power and Performance for Virtualized Heterogeneous Server Clusters [PCL⁺11]

Description: The authors proposed an approach to optimize the energy consumption in a multi-application heterogeneous cluster environment. They rely on both DVFS and server consolidation techniques to reduce the power consumption in a cluster. The objective is to determine the number of VMs needed for each application under a given workload in a way the overall energy consumption is minimized. For this purpose, the performance of an application hosted by a given PM is modeled as a function of the application workload, the PM's CPU frequency and utilization. The workload of each application is indirectly inferred by the runtime CPU usage and frequency in all PM in which the application is deployed. The power consumption of a PM, in turn, is modeled as a function of the CPU frequency and the CPU utilization. Applications are deployed in several VMs, which, in turn are deployed in PM. The performance and power models are used by a Mixed-Integer Programming (MIP) [WN99] based optimization model to determine whether a PM hosts an application (encompassed by a VM) at a given CPU frequency while minimizing the cost due to power consumption of all PMs in the cluster. The optimization model is used by an autonomic manager to periodically optimize the performance and power consumption in the cluster.

Discussion: The work is not really multi-layer since the information about the applications (workload) is inferred rather than obtained directly from applications. Even though, it takes into consideration information about application so as to globally optimize not only the power consumption but also the performance. With respect to the loose-coupling characteristic of cloud environment, the work would fit perfectly in this context (e.g. in a IaaS manager), since it does not explicitly require any information of other layers. However, regarding the scalability, the validation was performed only over three applications and five PMs, which may be not enough to show the feasibility for a great number of PMs and applications/VMs.

CloudPack: Exploiting Workload Flexibility Through Rational Pricing [ISBA12]

Description: CloudPack is a framework based on game theory that allows rational pricing. It is composed of two main group of services: backend and frontend services. Backend services correspond to services utilized by IaaS providers to optimize the infrastructure. It comprises three components: (i) migration, which permits to eliminate hotspots and thus optimize the infrastructure utilization rate; (ii) allocation, which manages the placement of VMs from several customers in the same pool of physical resources; and (iii) profiling or monitoring, which provides customers raw data that allows them to adjust their resource reservation as well as visibility to resource utilization and performance. Frontend services correspond to services exposed to IaaS customers and consist of two components: (i) workload specification, where a language is provided to customers so that they can define how workloads are defined in terms of virtual resource requirements and dependencies among them over the time; and (ii) pricing, which attributes rational cost across customers in a game-theoretic way. For the workload specification, CloudPack proposes a XML-based language which enables customers to define their workload in term of a Direct Acyclic Graph (DAG). A node corresponds to a task (i.e. a VM) to be mapped to a resource (PM) and edges correspond to temporal dependencies in time between tasks. Each node is specified with the resource dimensions required (CPU, RAM, disk, etc.) and the amount of timeslots necessary. According to the authors, this language is able to express any kind of workload requirements. The DAG specified by the customers are used by the allocator component to instantiate the VMs accordingly so as to minimize the hosting costs (i.e. the energy consumption). To this end, a MIP model and a heuristic-based greedy solving algorithm are proposed. The pricing component is in charge of rationally distributing the total costs across customers according to the contribution of each customer. To this end, CloudPack relies on the concept of Shapley [NRTV07], which is a concept of game theory that allows fair cost sharing among players (in this case customers).

Discussion: The main contributions of CloudPack is a language for workload description as well as the pricing characterization approach based on game theory that allows a fair repartition of the

total cost amongst clients. However, the approach does not deal with the lack of synergy between the application and infrastructure levels, which may cause negative interferences from one level to another. Indeed, the CloudPack is focused at the IaaS provider. More particularly, in the definition of jobs (workloads) and how they are fairly priced according to their hosting costs.

3.3.2 Multiple Autonomic Management

Coordinating Multiple Autonomic Managers to Achieve Specified Power-Performance Trade-offs [KCD⁺07]

Description: The work proposes a framework for coordinating the management of power and performance in a data center. The approach relies on a joint utility function, in which one can express the degree of usefulness of both performance and power consumption together. This is particularly done by a performance-based utility subtracted by a linear power cost. This framework is composed of two separate agents that respectively manage performance and power. After receiving state information from the performance manager, which has its own power-unaware policy, the power manager tries to optimize the joint utility by applying its power policy to manage the trade-off between performance and power. The performance manager does not need to be modified except for the fact that it needs to be informed about the dynamic changes in the CPU frequency. Indeed, the power manager performs adjustments in the CPU frequency settings to cope with power caps. That is, whenever the PM uses less energy than the power cap it runs at maximum speed, whereas when it wants to use more energy than the power caps, the power manager reduces the CPU frequency so as to meet the power constraint. The performance manager, in turn, relies on monitoring data to adapt the resource allocation for each application. Based on an off-line estimation on the number of CPU cycles/seconds a given request requires, the performance manager is able to determine the number of concurrent servers (PMs) aiming at having 90% of utilization for each server. Eventual conflicts are avoided by informing the performance manager the correct CPU frequencies which are managed by the power manager.

Discussion: The approach allows that two autonomic managers developed independently, eventually by different vendors, work in collaboration to manage at the same time power and performance. However, some details about one manager (in this case the performance manager), must be provided in order for the power manager to take it into consideration in the joint utility function. In the other sense, the power manager should also provide details about the CPU frequency, which requires modification on the performance manager. Therefore, this tight-coupling characteristic makes the approach not straightforward to be applied to multiple-vendors or service-consumer architectures such as in cloud computing.

No 'Power' Struggles: Coordinated Multi-level Power Management for the Data Center [RRT⁺08]

Description: The authors proposed a coordination approach to deal with the power autonomic management in several levels of the data center architecture (e.g. VM, PM, rack). The idea is that the output provided by one manager may be useful to other managers to produce a result which is globally more appropriate. The coordination architecture consists of five autonomic managers with different objectives which are coordinated via an application protocol interface (API), as it can be seen in Figure 3.6. The efficiency controller (EC) is a per-server power consumption optimizer. It is responsible for adapting the actual resource utilization to a reference value of utilization (provided as output from a higher level manager) by adjusting the power mode states (P-states) on the server. The server manager (SM) is in charge of regulating the thermal power of a given server by monitoring the actual power consumption and comparing it with a reference value (provided as output from a higher level manager or limited to a server level budget). As a result, the controller updates the utilization reference value used as input for the EC. The enclosure manager (EM)

implements the power capping at the enclosure servers (or rack) level also by comparing the actual power consumption across a set of servers with a reference value (the minimal value between an enclosure level power budget reference value and a reference value provided as output from a higher level manager). The result of this comparison is used to re-provision the power by updating the power budget for each server individually. The group manager (GM) is responsible for the power capping at the rack/data center level. Similarly, it monitors the actual power consumption and compares it with a reference group power budget. The result is used to assign the power budget of the enclosure of server. Finally, the VM controller (VMC) is in charge of reducing the average power consumption of a set of servers by consolidating VMs so that it is possible to turn off unused servers. Basically, it monitors the VMs' resource utilization and proposes a new mapping VMs/servers that optimizes the power consumption. It takes into account the current power state of servers in order to have the real server resource utilization which may influence the VMC decisions. Moreover, the VMC is aware of power budget caps and power budget violations at several levels so as to avoid conflicts among controllers. For example, with no constraints on the VMC optimization, the VMC may aggressively pack workloads, which may lead to an excess of local power consumption, which in turn can lead to performance throttling. The VMC may interpret this as more free room for consolidation, which results in a vicious cycle.

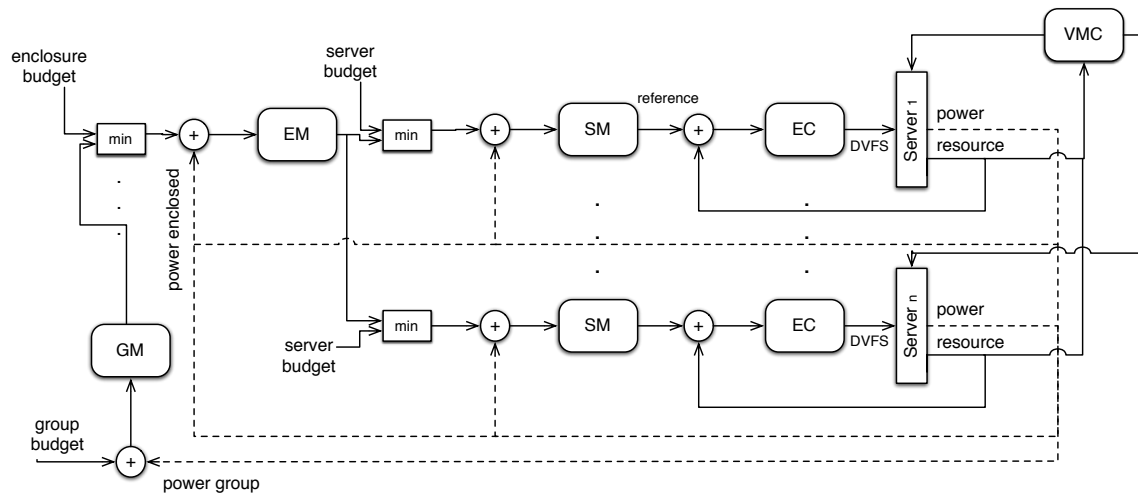


Figure 3.6: Coordinated Power Management Architecture for Datacenters [RRT⁺08].

Discussion: The work proposes an approach for the coordination of several autonomic managers, each one responsible for a specific resource in each level of the data center architecture. Although the approach has potential to be extended to other scenarios, it is focused on and restrain to the data center domain. That is, it does not cross the organization boundaries as it would be if a SaaS or PaaS providers were considered. In this case a more loose-coupled solution would be required to manage the lack of visibility between parts/layers.

Semantic-less coordination of power management and application performance [KLS⁺10]

Description: The authors propose an interface-based coordinated method for multiple applications and system layer. The idea is that only semantic-less numbers (i.e. integers that can be increased or decreased) representing performance and power are shared among applications and system so that shared values cannot be compared to other values, and a management module does not know whether a higher or lower values is better. Hence, the application has no information about the system power management modules, and similarly, the system has no details about the application performance levels. The coordination interface is composed of performance levels

(QoS levels), power levels, a signal variable (C) and a lock. Each application publishes its QoS levels ($QoS(i)$) (also called A-states) in terms of semantic-less integers (an integer set), on which only the concerning application writes. Similarly, each power management module publishes its power settings ($P(j)$) in terms of semantic-less integers, which are writable only by the concerning module. These settings may internally signify P-states, sleep modes, throughput cap levels, and so on. For the rest of the system it means only integer values. The system also publishes a signal value ($C = \{-1, 0, 1\}$) indicating that the energy should be reduced (-1), or that energy is available (1) or should it remain constant (0). This signal is only writable by the system module that understand the constraints on energy consumption. Any system module or application that wants to change the settings should acquire the lock first so as to avoid simultaneous uncoordinated actions that lead to undesirable states. The challenge is to choose a power management settings (e.g. P-states) such that the QoS of all applications approximates the optimal one. A coordination approach is provided to combine all applications needs in terms of QoS and energy requirements. This approach consists of two algorithms (cf. Figure 3.7): one for each system management module (a) and (b) another for each application. The algorithm for application management checks for a signal to reduce energy and if it is not yet in the lowest level of the A-states, it tries to get the lock, switches to the next lower A-state (if it is necessary), publishes it and releases the lock. The system management modules checks if the resource controlled by it is under-utilized, acquires the lock, changes the power settings and publishes it. It then monitors the QoS level of all applications in order to see if the power settings change produced any changes in the QoS levels. If that is the case, it undoes the power settings change and releases the lock, otherwise, it releases the lock and starts the process over again. That way, power levels will only be lowered if QoS levels in all applications remain the same.

- | | |
|--|---|
| <ol style="list-style-type: none"> 1. Set state $M = ACTIVE$. 2. Acquire lock. 3. Change power knob to next lower setting. 4. IF this causes any $QoS(i)$ to change <ol style="list-style-type: none"> (a) Revert setting to previous value (b) Set $M = INACTIVE$ and release lock. 5. ELSE <ol style="list-style-type: none"> (a) publish new $P(j)$, release lock, and goto Step 2. 6. Monitor shared interface. If any $QoS(i)$ or $P(j)$ values change, go to step 1. <p style="text-align: center;">(a)</p> | <ol style="list-style-type: none"> 1. Continue to check if $C = -1$ or user requested to save power. If need to reduce power, and not already in deepest A-state: <ol style="list-style-type: none"> (a) Acquire lock. (b) Switch to next lower A-state. (c) Publish new $QoS(i)$ and release lock. <p style="text-align: center;">(b)</p> |
|--|---|

Figure 3.7: Semantic-less Coordination Algorithm for (a) System/Infrastructure and (b) Application [KLS⁺10].

Discussion: The authors propose a signal-based coordination approach for several managers that aim to establish a trade-off between performance at the application level and power consumption at the infrastructure level while avoiding undesired changes. The approach can be easily scaled due to the fact that it relies on power management modules performing low-overhead operations (such as DVFS). While those power management modules enable a PM to improve its energy efficiency, it is far from a good solution for a more global scenario, say, a data center or a IaaS provider in which the average utilization rate is very low [Vog08]. In fact, more sophisticated techniques such as virtualization could have been used to improve the flexibility of the management of hosted applications in a way to mutualize physical resources to enable PMs to be turned off when they are needless rather than just adjusting their CPU frequency.

An Economic Approach for Application QoS Management in Clouds [CPMK12]

Description: The authors propose an economic model to manage the resource provisioning in a multi-application scenario so as to meet applications QoS goals while reducing costs. The architecture consists of several per-application managers which request resources to the resource manager by submitting bids that express their intention in paying for resources. The resource manager relies on a proportional-share rule [LRA⁺05] to provision resources according to the applications bid. The idea of having a fluctuating price policy is to provide differentiating services according to the bids, while the resource proportion is to optimize the resource utilization. More precisely, application managers specify the virtual cluster size (group of VMs), the minimum resource allocation that should be ensured by the resource manager and the spending rate. In order to scale up/down applications so as to meet their QoS goals, application managers should adjust their bids, by either changing the spending rate or the virtual cluster size. The resource manager periodically evaluates the new requests as well as the current allocation and slices the total infrastructure capacity proportionally among applications. This is done by occupying first the physical nodes with less utilization rate with the group of VMs with highest bid. As a consequence, allocations failures are minimized and the resource utilization is maximized. Similarly, in [NB10], a market-based QoS-based approach for web service composition was proposed. The idea is similar, but in that case the marketplace is for software services instead of computing physical resources.

Discussion: Although this approach does not deal directly with the problem of energy consumption, its results may indirectly impact positively the energy consumption on the underlying infrastructure by optimizing the resource utilization. Moreover, it provides an interesting multi-level approach in which the infrastructure details are not visible at the application level, meaning that the willingness and expectations at each level are expressed by the economic model. The approach seems also to be attainable in terms of scalability, since it performs local searches instead of global ones. However, it does not guarantee optimal allocations i.e. optimal placement VM/PM. In addition, it is not clear what happens when a virtualized cluster of a request fails in fitting in all existing nodes. More precisely, it is not clear if a cluster can be split into several groups and how it would impact on the rest of the allocation process. Furthermore, the approach does not take into consideration the fact that PMs may remain a long time without being used, which depends on applications' activity. In which case, there are no policies of resource utilization optimization such as migration or other based on the synergy with applications.

Coordinating Power Control and Performance Management for Virtualized Server Clusters [WW11]

Description: The work presents Co-Con: an approach for the coordination of multiple managers in order to manage power and application-level performance in data centers. More precisely, it considers a multiple application scenario, in which each application corresponds to one VM (cf. Figure 3.8). The data center is composed of several PMs and each PM can host several VMs. There is a cluster-level power controller, which is responsible for dynamically adjusting PMs' CPU frequency according to a reference value. To this end, the cluster-level power monitor periodically observes the actual power consumption of each PM and sends it to the power controller. The power controller, based on the difference between the actual power consumption and desired power consumption, computes the new frequency levels for each one of the PMs in the cluster. These levels are sent to the frequency modulators which are placed within the hypervisor of each PM in the cluster. For each VM there is an autonomic manager which controls the VM performance by dynamically adjusting a fraction of the CPU where the VM is hosted. This per-VM manager is composed of a performance monitor, which measures the actual VM average response time. Based on that value, the performance control calculates the amount of CPU necessary to improve the performance and sends it to the CPU resource allocator. The CPU resource allocator tries to (re)allocate the desired fraction of the CPU where the VM is hosted. If the requested resource

is more than the available, low-priority VMs will be given less resources than required and if the situation continues for a certain period of time, the resource allocator requests the cluster-level controller to migrate these VMs. In order to deal with interferences of one manager over the others, they claim that a higher level manager, i.e. the primary manager has to have a control period longer than the settling time of the other manager, i.e. the secondary one. That way, the secondary manager can always achieve its steady state within the control period of the primary one. The impact of the primary manager on the secondary one is considered as variation in the system model, whereas the interference of the primary manager on the secondary one is considered as system noises. So, the power manager is designed as the primary one and the performance manager as the second one. This is because the DVFS has a more significant impact on the performance than the impact of the CPU resource allocation on the power consumption. So, it is preferable to model it as model variation than a system noise.

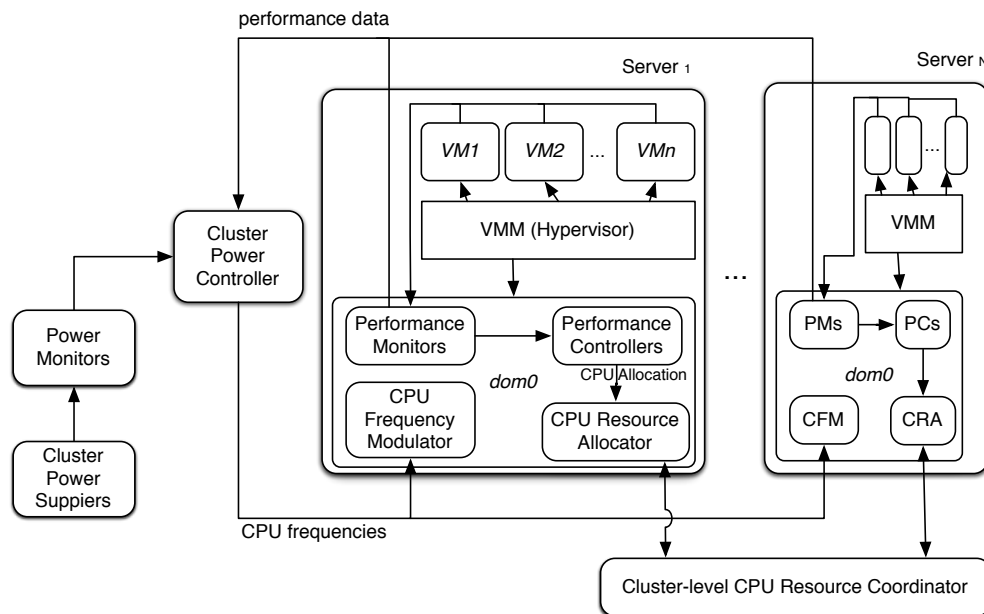


Figure 3.8: Coordinated Power and Performance Control Architecture [WW11].

Discussion: Co-Con presents a theory control based coordination approach of autonomic managers, in which managers can be designed independently, that is, there is no need for previous knowledge on the other managers. Although Co-Con has shown to be effective in terms of system stability there are some drawbacks that should be pointed out. The performance manager does not consider VM replication so as to make applications more elastic and thus cope with a varying workload. Contrary to a simple VM sizing (changing in the CPU allocation), a VM replication may significantly impact the power consumption, which may affect the way it should be modeled (model variation instead of system noise). Moreover, although CPU DVFS may have lower overhead than turning on/off PMs, its impact on the energy efficiency of an idle PM is also much less than the one of a turned off PM.

Business Process Co-Design for Energy-Aware Adaptation [CFF⁺11]

Description: The work proposes E-BP, an energy-aware business process. E-BP is an extension of regular BP model that takes into consideration the energy consumption incurred by the execution of business tasks. Services implementing the constituent tasks in the BP are hosted by VMs which in turn are hosted by PMs. The energy consumption along with other information about resource consumption are constantly monitored at all levels, that is, at the infrastructure (physical

devices), middleware (VMs) and application (services implementing business tasks). These measurements are associated with Green Performance Indicators (GPIs) and QoS dimension, based upon which a set of constraints are imposed. GPIs include measures on the CPU load or number of transactions per energy unit, for instance [FPP12a]. Given that, a set of adaptation strategies are proposed according to the following classification: *less quality*, *less functionality* and *resource allocation* strategies. *less quality* corresponds to non-functional requirements reduction or degradation (turn off a service of a task if there is more than one service available for the same task). The *less functionality* strategy refers to a computational or data degradation (e.g. skip a task or task functionality). The *resource allocation* strategy changes how or which the resources are used by the process (e.g. VM configuration or replication, enable the possibility to turn-off servers and disks). Also, it may happen that a triggered adaptation can be enacted along with other adaptations at the three layers (application, middleware and infrastructure). In that case, the approach relies on an inter-dependency model among indicators so as to avoid cascading negative effects upon an adaptation strategy.

Discussion: One of the limitations of the work is about the energy metering, since it is not easy to measure the energy consumption in several layers while isolating the impact of concurrent processes. For instance, the energy footprint of a service in isolation may be different than when it executes along with another service in a virtual environment. Another limitation of the approach is regarding the single application restriction, since the approach deals with a single process running several instances. So, it does not take into consideration multiple applications/processes sharing the same resources, as it would be the case in a public/hybrid cloud infrastructure. In addition, this limitation may result in serious problems with respect to the constraint management resolution. As the number of applications increases, the number of constraints and dependencies among GPIs may proportionally increase which may lead to an over-constrained rule definitions.

Coordinating Energy-aware Administration Loops Using Discrete Control [MKGdPR12]

Description: The work proposes a synchronization model of autonomic managers where a global coordination manager in a upper layer orchestrates other autonomic managers in a underlying layer. The model is applied to a scenario of energy efficiency in data centers whose objective is to coordinate two controllers: (i) optimization controller, which dynamically adapt the degree of server replication for load balancing purposes so as to cope with the application load; and (ii) CPU-frequency controller, which controls servers power consumption by dynamically adjusting a PM frequency (by relying on DVFS) according to the PM resource utilization. The optimization controller observes the average CPU utilization of the whole cluster and if it exceeds a certain upper/lower threshold, a PM is turned-on/off. Similarly, the CPU-frequency controllers increases/decreases the CPU frequency of a given PM if it exceeds an upper/lower resource utilization threshold and if it is not in the lowest/highest CPU frequency. The idea of having a coordination manager aware of states and decision of each controller is to prevent that actions performed by one controller impact negatively the management of the other one. For instance, an overload situation may trigger actions from both controllers such as turning on PMs and increasing the frequency of PMs. However, maybe only increasing the PMs frequency would be enough to cope with the increasing load. In order to tackle this synchronization issue, the approach relies on synchronous programming and discrete controller synthesis. For this purpose, the authors rely on BZR [DMR10], a language that allows the definition of reactive systems by means of generalized Moore machines. Thus, both the optimization and CPU-frequency controllers are defined in terms of automata machines along with a coordination controller. A set of control variables are also defined to help the synchronization of the two controllers. For instance, a variable that forbids the optimization controller to turn on a PM until all PMs frequencies are not set as the highest ones.

Discussion: The approach allows the coordination of autonomic managers by relying on the synchronization model-based technique. However, the scenario to which the approach was applied

considers a single application management that allows or not the instantiation of new servers for load balancing purposes with respect to the power management and the application load. In addition, controllers are defined at the same level within a same organization boundaries. Hence, it is not clear how it could eventually be applied in the context of cloud computing, for instance, where multiple SaaS applications may be managed independently of the IaaS management, without any hierarchical relationship between SaaS and IaaS providers.

Other Related Work

There are other work about multiple autonomic management that go far from the scope of this work. [Lem13] identifies five different patterns of interacting autonomic managers in self-adaptive systems where each pattern can be considered as a particular way to orchestrate the managers. [FDD12] goes further and proposes a collection of architectural design patterns addressing different classes of integration problems focusing on the possibly conflicting goals. [VWMA11] extends autonomic managers with support for two types of coordination: intra-loop (intra-manager) and inter-loop (inter-manager) coordinations, but restrained to a self-healing use case.

3.4 Summary

This chapter presented a selection of relevant work related to the topic of this thesis work. The work were grouped by application and infrastructure layers and cross-layered approaches. Approaches at the application layers take into consideration the applications characteristics (Quality of Service or functionality) to build strategies to reduce the energy consumption (or the resource usage and by transitivity the energy) at the underlying infrastructure. Approaches at the infrastructure layer consider the flexibility provided by hardware tuning like Dynamic Voltage Frequency Scaling (DVFS) or virtualization techniques at the infrastructure (virtual machine placement, consolidation, etc.) so as to improve the resource consumption and consequently the energy efficiency. Cross-layered approaches take into account both the infrastructure and application aspects to provide a global solution in terms of Quality of Service (QoS) (at the application level) and energy efficiency at the infrastructure level. Finally, multi-autonomic management approaches rely on several autonomic managers that may (or not) be coordinated to achieve their goals. In our case, the goals are to improve the QoS at the application level while optimizing the energy efficiency at infrastructure level. All those previously mentioned related work areas are summarized in Figure 3.9.

In order to concisely evaluate the work previously described, we define some qualitative comparison attributes:

- **Functional Adaptation:** this attribute indicates that at the application level, applications are capable of changing their architecture so as to adapt to changes in the environment. This architectural changes may impact their functionality. As a consequence, the degree of desirability of each possible functionality should be taken into account.
- **Non-functional Adaptation (QoS):** there might be changes (architectural or not) at the application level that may impact QoS attributes such as response time, throughput, etc. For example, decisions taken at application level to increase the resources allocated to it may lead to performance improvements. Conversely, the QoS may be degraded if resources shrinks.
- **Multi-application:** this attribute refers to the possibility to deal with several Software-as-a-Service (SaaS) applications sharing the same public infrastructure Infrastructure-as-a-Service (IaaS). Since our focus is on cloud environments, it is imperative that the solution takes into consideration the management of multiple applications.

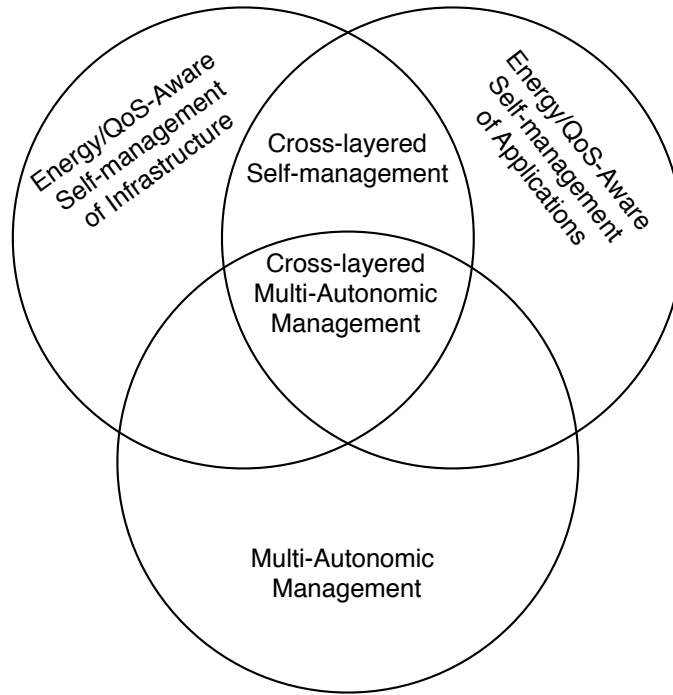


Figure 3.9: Intersection of the State of the Art Areas.

- **Cross-layered:** this attribute indicates that the approach takes into consideration aspects of both applications and infrastructure layers to deal with the problem of QoS and resource/energy consumption. For instance, an approach that performs QoS-based resource allocation (at application level) and server consolidation (at infrastructure level) is considered as being cross-layered.
- **DVFS/Switch on-off PM:** for approaches at the infrastructure-level, energy or resource saving strategies can be materialized by either DVFS, switching Off PMs or a combination of them. On the one hand, DVFS offers the possibility to reduce the energy consumption with a low system overhead by adjusting the CPU voltage and frequency. On the other hand, due to the lack of proportionality of hardware systems [BH07], this approach still remains limited when no work at all is required for a given set of PMs, which is the case in many data centers [Vog08]. Therefore, although the high overhead (in comparison to DVFS) of switching off / on a PM, the gains in energy can be much higher if well done. The ideal solution would be to combine the two options.
- **Multi-Autonomic Management:** this attribute refers to the ability of systems to be managed by relying on several autonomic managers, which may require some kind of coordination between them to achieve the desired result.
- **Loose-coupling:** sometimes, in a cross-layered approach, details of one layer is considered to solve problems of another layer, no matter if it is a single or multi-autonomic management approach. Since this thesis focus on cloud environments in which actors interact in a loose-coupling manner via well defined interfaces, we consider the loose-coupling as an important characteristic for comparison.

Table 3.1 summarizes the related work with regards to the above mentioned comparison attributes, where X , $-$ and n/a correspond respectively to a feature implemented, not implemented, and not applicable to/by the approach in question.

From the work that takes into consideration only the application aspects, all of them are able to adapt the non-functional aspects (QoS) according to the environment. However, only [PDPBG09]

and [YT12] present a multi-application approach in which applications are capable of changing their functionality with alternative configurations to cope with environment changes. From the work that perform energy optimization only at the infrastructure level, only [BB10] performs both DVFS and PM switching, the rest [LLH⁺09, HLM⁺09, HLM⁺09] relies on consolidation and therefore PM switching off. From the body of work that consider both application and infrastructure aspects to optimize applications' QoS and infrastructure energy consumption, all of them are able to adapt the QoS (non-functional aspects) so as to utilize less resources and consequently less energy. Only [GDA10, DAB11] provide means to adapt the applications so as to provide alternative configurations. Regarding the energy strategies, only [PCL⁺11] and [ISBA12] allow both DVFS and PM switching off, whereas the others rely only on PM switching. Finally, from the work that are cross-layered and multi-autonomic manager, [RRT⁺08] is considered a multi-autonomic approach within the data center boundaries (e.g. VM, PM or rack level). Meaning that the approach does not consider explicitly any aspect of applications in order to optimize the energy consumption in the data center. [KLS⁺10] and [CFF⁺11] provide cross-layered and multi-autonomic management solutions in which applications are capable of adapting both their functionalities and their non-functional aspects. While [KLS⁺10] is also able to deal with multiple applications, [CFF⁺11] considers a single application scenario. However, the energy saving strategies in [KLS⁺10] as well as in other approaches [MKGdPR12, WW11, KCD⁺07] is based only on DVFS, which can be relatively limited. Finally, concerning the loosely-coupling characteristics, only [KLS⁺10],[CPMK12] and [WW11] provide solutions in which autonomic managers work in a loosely-coupled fashion. This is particularly interesting because in the context of cloud computing, the details of a manager running at a given layer might not be visible to some other manager located at a different layer.

In order to overcome the previously mentioned limitations this thesis presents a cross-layered optimization approach for applications' QoS and infrastructure energy efficiency in the context of cloud computing. The optimization is achieved thanks to the coordination of a multiple autonomic managers that communicate in a loosely-coupled manner. At the application level, each application is equipped with its own manager which is able to adapt not only the application non-functional aspects (QoS) but also its functionality so as to be more flexible to changes in the environment. At the infrastructure level, another manager tries to reduce the number of PMs (and consequently their implied energy consumption) necessary to attend all applications' requests.

	Work	Functional Adaptation	Non-Functional Adaptation	DVFS / Switch On/Off PM	Multi-application	Loosely-Coupled Management	Year
Application	[MFKP09]	-	X	n/a	-	n/a	2009
	[PDPBG09]	X	X	n/a	-	n/a	2009
	MoKa [AB10]	-	X	n/a	-	n/a	2010
	[PPMM11]	-	X	n/a	-	n/a	2011
	[Fer11]	-	X	n/a	-	n/a	2011
	[YT12]	X	X	n/a	X	n/a	2012
Infra	EnaCloud [LLH ⁺ 09]	n/a	n/a	Switch	X	n/a	2009
	Entropy [HLM ⁺ 09]	n/a	n/a	Switch	X	n/a	2009
	[BB10]	n/a	n/a	Both	X	n/a	2010
	Snooze [FRM12]	n/a	n/a	Switch	X	n/a	2012
Cross-Layered	[VTM10]	-	X	Switch	X	-	2010
	SAFDIS [GDA10, DAB11]	X	X	-	X	-	2010/2011
	[PCL ⁺ 11]	-	X	Both	X	-	2011
	[APTZ12]	-	X	Both	-	-	2012
	CloudPack [ISBA12]	-	X	Switch	X	-	2012
	[KCD ⁺ 07]	-	X	DVFS	-	-	2007
	[RRT ⁺ 08]	-	-	Both	-	-	2008
	[KLS ⁺ 10]	X	X	DVFS	X	X	2010
	[CPMK12]	-	X	-	X	X	2011
	Co-Con [WW11]	-	X	DVFS	X	X	2011
	[CFF ⁺ 11]	X	X	Both	-	-	2011
	[MKGdPR12]	-	X	DVFS	-	-	2012

Table 3.1: Related Work Summary.



Contributions

Cloud Computing and Component-based Software Engineering: Elasticity in Many Levels

This chapter discusses about how the flexibility of the on-demand provisioning model of cloud computing along with the flexibility made by component-based applications may contribute to cope with dynamic environments in many levels.

Firstly, we present an overview of the context in which this thesis takes place, by presenting the main actors involved and by briefly providing some discussion on how applications and infrastructure flexibilities can help the optimization of the Quality of Service (QoS) and the resource/energy usage. Then, we detail, through an example, how state-of-the-art solutions benefit of that flexibility to reduce the energy consumption at the infrastructure level. We then detail, also through an example, how current solutions benefit of application-level flexibility to improve the application QoS so as to face with changing environments and how it may impact the resource usage on the underlying infrastructure. In addition, we provide some discussion about how applications can become even more flexible and able to be more energy compliant. Finally, we introduce an multi-level autonomic approach that aims to make both applications and infrastructure self-manageable with respect to the infrastructure energy efficiency and applications' QoS.

Contents

4.1	Context Overview	58
4.1.1	Cloud Actors	58
4.1.2	Elasticity in Many Levels: From Infrastructure to Software Architecture	58
4.2	Energy-aware Elastic Infrastructure	59
4.2.1	Example Scenario: Virtual Machine Placement	59
4.3	QoS-aware and Energy-compliant Elastic Applications	60
4.3.1	Capacity Elasticity	60
4.3.2	Architectural Elasticity	61
4.3.3	Motivation Scenario: Advertisement Management Application	62
4.3.4	Conclusion	63
4.4	Autonomic Architecture to Optimize Energy Consumption	64
4.4.1	Architecture	64
4.4.2	Coordination and Synergy Issues	65

4.1 Context Overview

This section provides an overview of the context in which this work takes place. First we present the cloud actors involved in our solution. Then we revisit some state-of-the-art solutions for energy/resource usage and QoS optimization at both infrastructure and application levels. Finally, we overview how to improve the level of flexibility at the application level, which can be interesting to make applications more energy compliant.

4.1.1 Cloud Actors

We leverage cloud computing software stack (cf. Figure 2.1) to define the actors considered in this thesis work. Hence, there are three main actors who interact with each other in a producer-consumer fashion, that is, actors at lower layers offer services that are consumed by the actors at upper layers. At the lowest layer, the *infrastructure provider* offers computing resources to *application providers* in a Infrastructure-as-a-Service (IaaS) way. Thus, there is a Service-Level Agreement (SLA) stating that the *application provider* pays a financial cost to the *infrastructure provider* corresponding to the amount of usage time and resources. The *infrastructure provider*, in turn, must ensure a certain Quality of Service (QoS) such as the service availability. At the middle layer, the *application providers* offer application functionalities to *clients* in a Software-as-a-Service (SaaS) manner. There are SLAs stating the financial cost *clients* have to pay for the offered services and the QoS guaranteed by the *application providers*. Every time a SLA is violated, the service provider (*application or infrastructure*) provider is penalized. In which case, the service client does not pay the full price or does not pay at all for the service. Therefore, it is important that service providers avoid as much as possible service degradation which may cause SLA violations and thereby a drop on their incomes.

Although the management of SLA is extremely important to formalize the commercial relationship between providers and consumers, it is out of the scope of this thesis work. Hence, the SLAs are not explicitly nor formally taken into account, which means that there is no formal statement describing the parts, roles, rights, obligations and penalties. On the other hand, even if we do not take into consideration a formal statement or special treatment for the eventual violations, the SLAs may help in the definition of service level objectives that should guide the management of each service (e.g. SaaS or IaaS) in the whole system.

4.1.2 Elasticity in Many Levels: From Infrastructure to Software Architecture

As previously mentioned in Chapters 2 and 3, the issue of energy/resource consumption and QoS optimization can be directly or indirectly addressed at several levels. At a lowermost level, the flexibility of virtualized infrastructure inherent to cloud computing systems along with consolidation or optimal placement/scheduling techniques enable a better utilization of the physical resources, which consequently allows a part of the physical infrastructure to be shutdown. This flexibility in the virtual resource provisioning allows application providers to dynamically scale up and down, which results in a more reasonable resource (and consequently energy usage) for the maintenance of applications' QoS.

However, dealing with applications as *black-boxes* (i.e. as if the internal application composition was unknown) may not be enough to cope with highly dynamic environments, in which constrained situations as for instance to meet energy requirements may emerge. Therefore, we claim that in order to increase their flexibility, applications could be considered as *white boxes* in the sense that their internals are visible from the outside so that it is possible to configure them to fit a certain situation. In this context, component-based software engineering (CBSE) [HC01] along with reflection capabilities [Smi84] play an important role, since they provide means for

modular and dynamic (runtime) adaptation. As a result, it enables the management of even more flexible applications which are able to enable/disable functionalities or non-functional aspects in order to face resource/energy issues or other kinds of constrained situations.

To sum up, we propose a multi-level elastic approach in which adaptation can be seamlessly performed thanks to the flexible nature of applications and infrastructure. The *infrastructure elasticity* refers to the capability of the *infrastructure provider* to readily provision computing resources by allocating them in a way to optimize the infrastructure utilization. The *application elasticity* can be divided into *capacity elasticity* and *architectural elasticity*. The former refers to the capability of *application providers* to dynamically adjust their applications' resource capacity so as to cope with a varying environment (e.g. workload) while maintaining the QoS. The latter refers to the capability of *application providers* to internally adapt their corresponding applications (by enabling/disabling functionalities or non-functional aspects) in order to be even more able to fit non-anticipated environment changes (e.g. more energy compliant).

The next sections details these three levels of elasticity through a set of example scenarios.

4.2 Energy-aware Elastic Infrastructure

This section aims at describing the infrastructure and how it can be elastic with regards to the energy consumption.

The infrastructure corresponds to a pool of physical resources expressed by means of physical machines (PM), which are defined in terms of computing resource capacity, namely CPU and memory (RAM). The infrastructure provider relies on virtualization [SN05] to be able to concentrate several workloads in a single PM and thereby optimize the resource usage. Hence, the physical resources are offered to application providers in terms of virtual machines (VMs). Likewise, VMs are defined in terms of computing resource capacity (CPU and RAM) and can be dynamically created, destroyed and migrated from one PM to another. Lastly, each PM is able to simultaneously host one or several VMs according to their resource capacity (CPU and RAM).

The infrastructure provider may benefit of the virtualization to improve the resource utilization rate and thereby the energy efficiency [HLM⁺09, LLH⁺09, FRM12]. Actually, it is possible to place new VMs in a way the number of PMs necessary to hosts all the VMs is minimized. The following section details through an example how this can be done.

4.2.1 Example Scenario: Virtual Machine Placement

This scenario illustrates how the placement can be performed so as to use less PMs and by consequence less energy.

In this scenario, the infrastructure is composed of four PMs (PM_1, \dots, PM_4), as it is illustrated in Figure 4.1. Each PM has a resource capacity of four units of CPU (y-axis) and RAM (x-axis). VMs are offered in two types: small (1 unit of CPU and 1 unit of RAM) and large (2 units of CPU and 2 units of RAM).

At time t_1 all the PMs are unoccupied and PM_3 and PM_4 are powered-off. At time t_2 upon requests of new VMs (namely VM_1, \dots, VM_5), the infrastructure provider creates these VMs by placing them in a way to use as less PMs (turned-on) as possible. It is important to mention that, due to resource constraints, if new request for VMs, it would be necessary to turn-on either PM_3 or PM_4 to place the requested VMs.

Besides, the VM placement, the infrastructure provider might rely on mechanisms like VM migration to dynamically to pack VMs by reposition them in a minimum number of PMs [HLM⁺09]. Although this kind of technique might lead to higher utilization rates and therefore to high efficiency in terms of energy consumption, we claim that it should be performed carefully and in moderation. Migration operations performed within a short period of time may cause interruption in the VM services, which may interfere the application QoS [VKKS11, Str12].

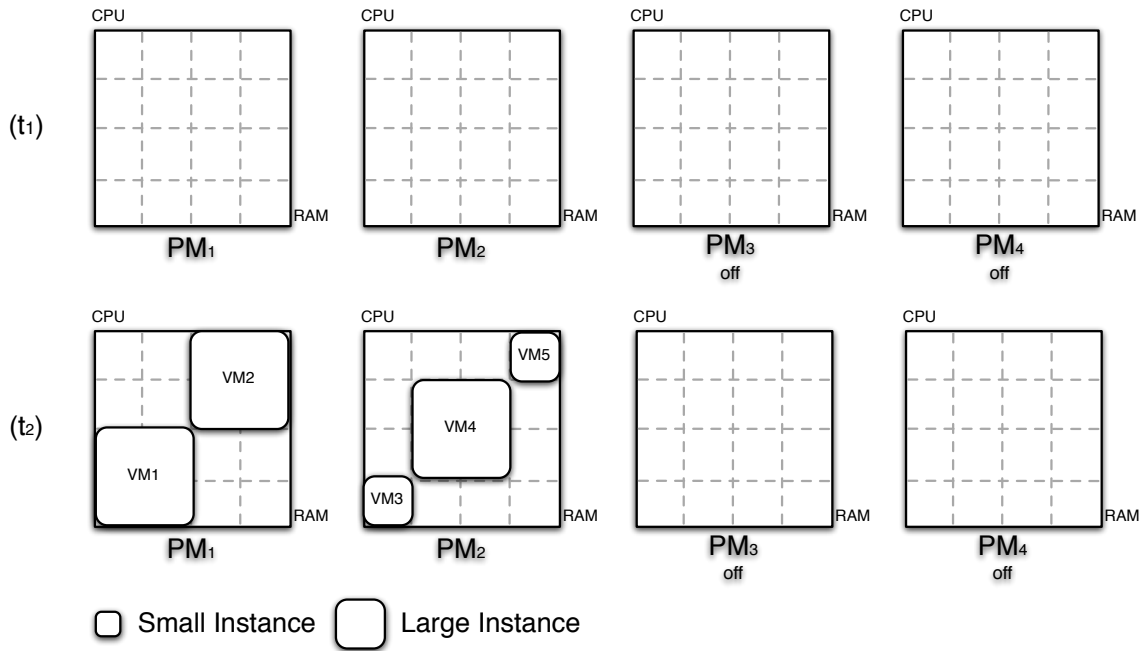


Figure 4.1: Energy-aware Placement Example Scenario.

4.3 QoS-aware and Energy-compliant Elastic Applications

In this section, we describe how applications are defined, how they are deployed on the underlying infrastructure and how they can scale up/down by dynamically stretching or shrinking the amount of resources (capacity elasticity). Then we describe how applications have been architecturally modified (architectural elasticity) and thus be more energy compliant. We then present a motivation scenario to ease the understanding of these resource and architectural elasticities. We conclude this section by providing some discussion about how these two elasticity mechanisms can turn the application more flexible and consequently more energy compliant.

Before providing the details on the application-level types of elasticity, it is noteworthy that we relied on the service component architecture (SCA)¹ (cf. Section 2.3.3) as component model to support the applications' definitions.

4.3.1 Capacity Elasticity

As presented in Section 4.2, the flexibility provided by virtualization allows application to seamlessly request and release computing resources in an on-demand manner. As result, *application providers* can estimate the amount of resources necessary to host the application so as to face the current workload (i.e. the number of simultaneous clients).

In this context, we rely on virtualization to provide computing resources to applications component. Each component has one or several *component instances* which are deployed each one in a different VM. It means that a VM must host at most one *component instance*. This can be interesting to scale up/down a particular component, since only instances for a particular component have to be created or destroyed. Figure 4.2 shows an example on how *component instances* of a given component *c* are deployed in VMs. By relying on techniques of load balancing [LL05], the applications' load can be balanced so as to able to deal with the all incoming requests and thereby meet the requirements in terms of QoS.

It is straightforward that *application providers* can benefit of this elasticity to reduce its energy footprint. In fact, this approach allows them to finely adjust the amount of resource necessary to

¹<http://www.oasis-open.org/>

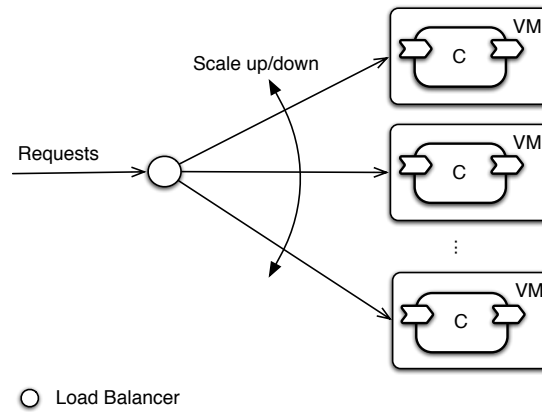


Figure 4.2: Example of Component Scaling Up and Down.

execute the application in a given context, which avoids resource over allocation.

This kind of scaling presented in Figure 4.2 is also known as *scale out* (or horizontal scaling). It distinguishes from the *scale in* (or vertical scaling) by the fact the former allows to increase/decrease the resource capacity by adding or removing VM instances, whereas the latter allows to increase/decrease by re-dimensioning the physical resources (e.g. CPU and RAM) allocated to a single VM instance [CRMDM12]. Currently, in virtualized infrastructures, the *scale in* technique is done by destroying a VM (or at least shutting it down) and re-creating (or re-launching) it with a bigger capacity. Consequently, there is a non-neglected and undesired service interruption time. For this reason, this work does not take into consideration the *scale in* scaling.

4.3.2 Architectural Elasticity

As previously said, we assume that applications are developed in a modular component-based manner and that they can be introspected and interceded to allow dynamic reconfigurations.

Applications can therefore operate in different *configurations* depending on which components are running, their properties, and how these components are bound to each other. Moreover, applications can switch from one configuration to another at runtime. Two or more *configurations* may provide the same functionalities while providing different levels of QoS. For instance, there could be a *configuration* that comprises a non-functional component (e.g. a logger), which is disabled in another *configuration*. The application *configurations* may also provide different functionalities that might impact the QoS. For instance, an application could have a *configuration* that contains a component implementing one functionality (e.g. a 2D map viewer). In another configuration the component is replaced by another component that implements another functionality (e.g. 3D map viewer). Supposing that 3D map viewer is a more desired functionality than the 2D one, this component change may end up by affecting application overall QoS. So, the QoS for the *configuration* comprising the 2D map viewer is functionally degraded in comparison to the other *configuration*.

Since components might have different requirements in terms of resources, applications may impact differently the resource usage and therefore the energy consumption, according to the configurations. Thus, applications are able to lower their energy footprint or cope with a given restriction by degrading their QoS. And the other way around, they can upgrade their QoS and thus increasing their energy footprint on the underlying infrastructure.

In short, this architectural elasticity characteristics promoted by a runtime CBSE approach are very appropriated to highly dynamic environment such as in cloud computing. The next section illustrates an example that eases the understanding of this kind of elasticity.

4.3.3 Motivation Scenario: Advertisement Management Application

The motivation scenario application consists of a 2-tier web-application in the domain of internet advertisement. The application behaves in an autonomous manner by choosing the appropriate advertisement type (video, image or text) according to the runtime context, namely the current workload, the advertisement quality, the performance and the energy compliance. As shown in Figure 4.3, the application is composed of four components:

- Frontend (FE). It is the presentation tier of the application which is implemented by a web container. Upon a client HTTP request, the web container encapsulates the advertisement media retrieved either from components Video Ad Server, the Image Ad Server or the Text Ad Server, and returns it.
- Video Ad Server (VAS). It chooses randomly one advertisement video file and returns it to the FE.
- Image Ad Server (IAS). This component also randomly picks an advertisement image file and returns it to the the FE.
- Text Ad Server (TAS). Similarly, it chooses randomly an advertisement text and returns it to the FE.

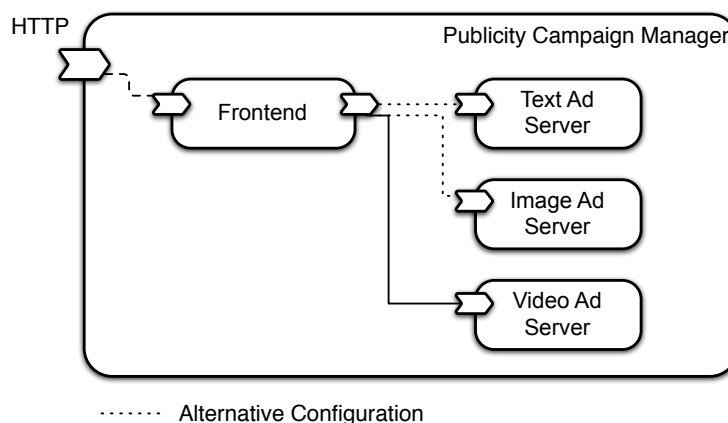


Figure 4.3: Advertisement Manager Architecture Overview.

The *application provider* establishes a SLA with his/her clients stating that they are charged for each advertisement exhibited on time, that is the advertisement should not exceed a certain amount of time to be exhibited. Thus, the type of advertisement along with the response time constitute the application’s QoS criteria for which clients pay more the higher they are. Similarly, the *application provider* also establish a contract with the *infrastructure provider* stating that the *application provider* must pay a certain amount of money according to the amount of resources required.

Table 4.1 presents the resource requirements (in terms of CPU and RAM) for each component. There are two levels of workload: high and low. For each one it is specified the resource requirements for a given performance level (namely bad or good). Components *FE* and *TAS* are the less resource consuming components. In fact, only one unit of CPU and RAM is enough to guarantee a “good” service under a heavy workload. With regards to components *IAS*, that amount of resources guarantees a “bad” service under a light workload. Two units of CPU and RAM are required to guarantee a “good” service under a light workload or a “bad” service under a heavy workload. In order to guarantee a “good” service under a heavy workload, it requires three units of CPU and RAM. Finally, component *VAS* is the most resource consuming component. It

requires two units of CPU and RAM to guarantee a “bad” service under a light workload, whereas it requires three units of CPU and RAM for a “good” service under a light workload or a “bad” service under a heavy workload. For a “good” service under a heavy workload it requires four units of CPU and RAM.

Hence, the *application provider* can benefit of the flexibility provided by the *infrastructure provider* to request the exact amount of resources needed to execute the application under a certain workload.

Component	Workload	Performance	CPU Units	RAM Units
<i>FE</i>	heavy	good	•	•
<i>TAS</i>	heavy	good	•	•
<i>IAS</i>	light	bad	•	•
		good	••	••
	heavy	bad	••	••
		good	•••	•••
<i>VAS</i>	light	bad	••	••
		good	•••	•••
	heavy	bad	•••	•••
		good	••••	••••

Table 4.1: Resource Requirements for the Advertisement Management Application.

Apart from the resource elasticity, the *application provider* also benefits of the architectural elasticity provided by the CBSE approach. To this end, Table 4.2 shows three possible configurations for the application and their respective QoS (in terms of responsiveness and advertisement quality) as well as the energy compliance (based on the resources usage). These criteria are expressed in terms of stars, that is, the more stars, the better is the criterion. It should be mentioned that in order to facilitate the comparison of configurations, the performance and energy compliance criteria are for a fixed amount of allocated resources under a constant workload.

Configuration 1 (k_1) is the configuration with best performance and worst advertisement quality, because it is composed of one *FE*, like the others, and one *TAS* whose requirements in terms of resources are lower than those of components *IAS* and *VAS*. As a consequence, it also has a best energy compliance. On the other hand, the advertisement quality is the worst in comparison to the quality of the other advertisement server components (*IAS* and *VAS*).

Configuration 2 (k_2) is an intermediate configuration in which the advertisement quality and performance criteria are of average quality. The component *IAS* is more resource consuming than component *TAS* and less resource consuming than component *VAS*, meaning that this configuration is able to handle more/less requests for the same amount of resources in comparison to components *TAS* and *VAS*, respectively. That way, its the level of energy compliance is average in comparison to these two components.

In Configuration 3 (k_3), the advertisement quality is better than the other configurations, since it is composed by components *FE* and *VAS*. As the latter is not able to manage the as many number of client requests as components *TAS* and *IAS*, the performance and energy compliance are lower than those of the other configurations.

4.3.4 Conclusion

By considering the application as a *white box*, it is possible to harness the possible configurations of components which allows the application to be more elastic in an architectural fashion. Hence, it is possible to take into consideration the domain-specific criteria (e.g. advertisement quality) and under which situations it is pertinent to degrade/upgrade (i.e. switch from one configuration to another) by considering the impact in terms of resource consumption. In short, it increases

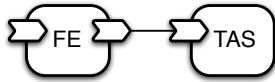
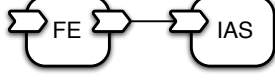
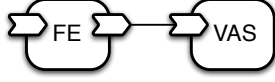
	Configuration	Trade-off	
k_1		Ad Quality	★
		Performance	★ ★ ★ ★ ★
		Energy Compliance	★ ★ ★ ★ ★
k_2		Ad Quality	★ ★ ★
		Performance	★ ★ ★
		Energy Compliance	★ ★ ★
k_3		Ad Quality	★ ★ ★ ★ ★
		Performance	★
		Energy Compliance	★

Table 4.2: Possible Architectural Configurations for the Advertisement Management Application.

applications flexibility and allows *application providers* to find the best trade-off between QoS and the resources needed to execute the application by adapting the application to changes in the context, be it regarding the application incoming workload or related to changes on the underlying infrastructure (e.g. energy shortages).

4.4 Autonomic Architecture to Optimize Energy Consumption

This section describes firstly a two-level autonomic architecture whose objective is to make applications and infrastructure self-adaptable with regards to QoS and energy consumption. Then, we discuss some issues regarding the lack of coordination and synergy presented in this kind of system.

4.4.1 Architecture

It is important to recall that cloud environments are highly dynamic. Indeed, the workload of this kind of cloud applications may fluctuate substantially within the same business day and the flexible resource provisioning of *infrastructure providers* may also make the infrastructure very instable. As a consequence, it becomes infeasible the manual intervention to rapidly react to changes so as to: (i) maintain the desired level of applications' QoS while internally configuring applications and adjusting the amount resources necessary; and (ii) arrange VMs in PM so as reduce as much as possible the energy consumption at the infrastructure level.

For this purpose, we rely on autonomic computing [KC03] to conceive a system consisting of two kinds of autonomic managers: the *application manager* (AM) and the *infrastructure manager* (IM). As it is depicted in Figure 4.4, each application and infrastructure is managed individually, which means that there is one AM per application and one IM for the infrastructure. The choice for several autonomic managers (one per application) instead of only one is justified by many reasons. As the number of applications increases, it becomes infeasible to manage several applications, at the same time, with different objectives parameters. In addition, applications may have different characteristics and because this the autonomic tasks (monitoring, analysis, planning and execution) may differ significantly from one application to another. For instance, the reaction time (i.e. the time interval between the end of one loop round and the start of another) of an application that performs scientific calculation might be different than for a web application. Furthermore, in cloud environments, for organizational boundaries reasons, it might not be desirable to have applications and infrastructure belonging to different vendors being managed together by the same entity. That is, the management details on a given organization may not be shared with other organizations. For those reasons, we advocate a multiple autonomic manager architecture in which applications and infrastructure are independently managed.

It is also important to remember that violations on the SLAs established between providers and consumers may lead to additional costs and consequently to a reduction of the providers' revenues. That way, *application providers* should always try to increase as much as possible their applications QoS while reducing as much as possible the costs due to resource allocation. The *infrastructure provider*, in turn, should try to sell as much resource as possible while minimizing the costs due to the infrastructure energy consumption.

So, the AM's main objective is to optimize the application QoS while adjusting the amount of resources necessary to host the application. It takes into consideration the incoming workload to decide which architectural configuration is more appropriate for the current context. For example, in order to cope with an increasing workload, the AM may decide to request more resources and maintain the same level of QoS. Alternatively, the AM may decide to switch from one configuration to another by degrading its QoS (e.g. switching from video to image advertisement) instead of requesting more resources. As a consequence, the *application provider* does not need to pay more for resource renting fees. This choice should be driven by the QoS degradation specification.

At the lower level, the IM is in charge of autonomously responding to AMs' requests on computing resources (VMs) while optimizing the placement of just-created VM. To put it another way, the IM tries to place the new incoming VMs in way the number of used PMs are minimized. As a result, the unused PMs may be turned off, leading to an improvement of the energy efficiency. Eventually, the IM may also try to pack the running VMs so as to minimize the number of running PMs. However, this operation should be done in moderation, since this is a very costly and susceptible to service interruptions [VKKS11, Str12], which may impact the QoS at the application level.

4.4.2 Coordination and Synergy Issues

It is noteworthy that due to the strong dependency among the component instances and the hosting VMs, the executions at both levels must be coordinated. For example, the VM that hosts one component instance cannot be stopped before the component has been removed from the rest of the application. Also, a component instance cannot be deployed until the VM that will host it has not been created.

Furthermore, autonomic decisions are taken in an independent way, that is, autonomic managers are not aware of the current state of the other autonomic managers' managed systems. Thus, decisions taken by one autonomic manager may affect (negatively or positively) a system managed by another autonomic manager. For example, let us suppose that the AM for the advertisement management application decides to request a large VM to the IM in order to face an increasing workload while keeping the best architectural configuration (video). Supposing that the infrastructure has only two PMs turned on and two PMs turned off, and that there is only one spot left to accommodate a small VM instance. That is, the IM should turn on a new PM in order to place a large VM, which globally would not be very optimal solution in terms of energy efficiency. Alternatively, the AM could decide to degrade the QoS by switching from configuration k_1 to k_2 , which would require one small VM instead of a large one. Hence, the impact on the underlying infrastructure would be more positive, since utilization rate would be increased without needing to turn on any PM. Nonetheless, as the AM is not aware of the current state of the managed infrastructure, it would be impossible for the AM to take such a decision in an independent manner. In short, the lack of synergy between autonomic managers may lead the global system to undesired states.

Likewise, let us suppose that the infrastructure provider faces a constrained situation such as an energy shortage, which force him/her to shutdown part of the physical infrastructure. If this is done without any coordination with the guest applications, they will likely have their QoS drastically affected. Instead, the infrastructure provider could warn guest applications about the situation so applications have the chance to cope with before the shortage actually takes place. Moreover, it is preferable that this is done in a way that mitigates the interference on applications. Still, the infrastructure is not aware of the details at the application level in order to be able to take

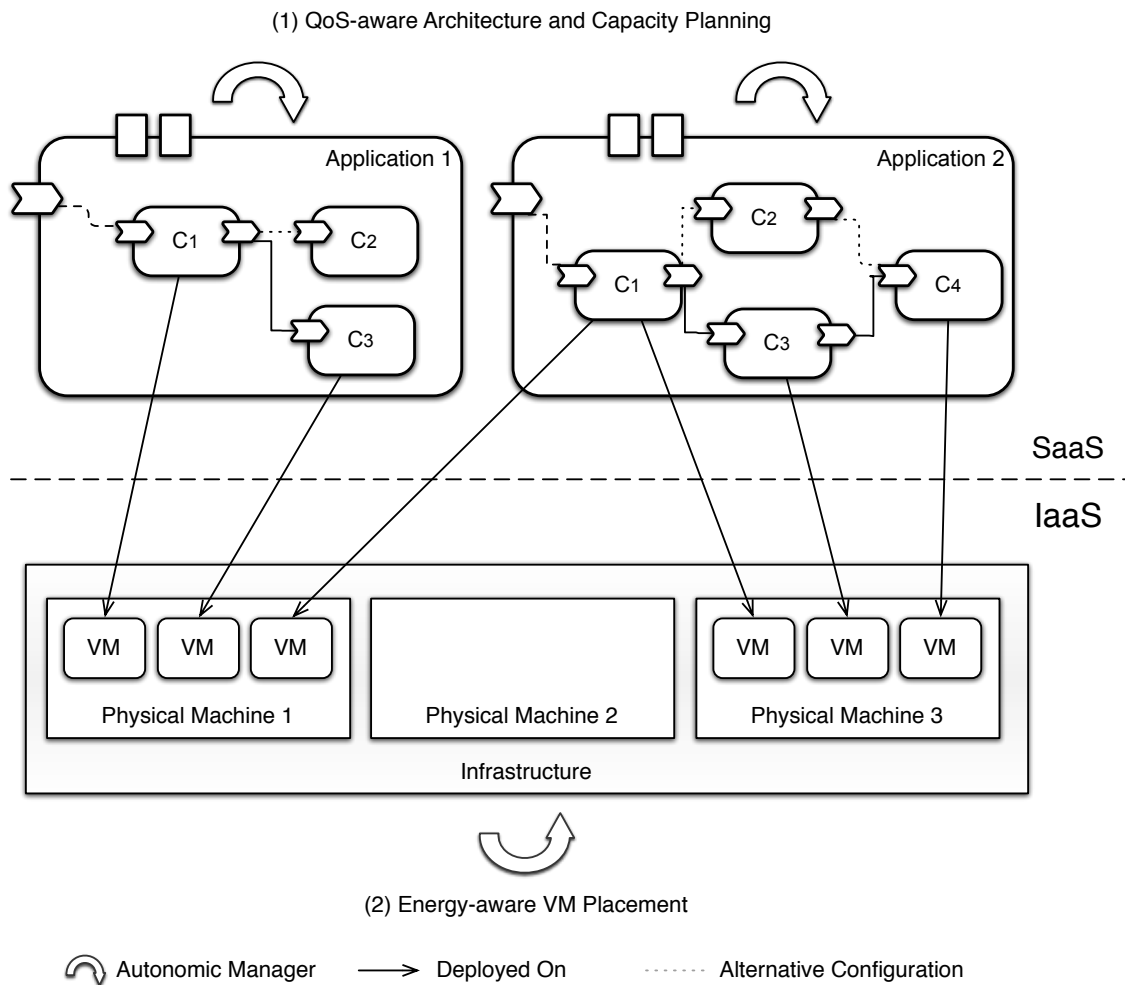


Figure 4.4: Multi-level Architecture for QoS and Energy Efficiency Optimization.

such an application QoS-aware decision. Therefore, the lack of synergy between the infrastructure and application may encumber the seamlessly adaptation to such a kind of constraint.

4.5 Summary

This chapter presented an architecture overview of a multi-level self-management approach for cloud applications and infrastructure. It relies on autonomic computing to make applications and infrastructure self-adaptable and able to optimize the Quality of Service (QoS) and the energy efficiency, respectively.

We claim that elasticity can be achieved at many levels and may significantly contribute to cope with dynamic environments while meeting QoS requirements and optimizing the energy efficiency.

Firstly, we defined an elastic infrastructure by means of virtual machines (VMs) deployed on physical machines (PMs). That way, it is possible to mutualize the computing physical resources by deploying several VMs in a single PM. Through an illustrative example, we showed how to optimize the placement of new virtual machines and also how virtual machines can be rearranged so as to minimize the number of physical machines needed.

Secondly, we relied on component-based software engineering to define elastic applications. Along with the resource capacity elasticity property of cloud applications, we are able to conceive applications that are capable of reconfiguring to suit the runtime context. We presented a

motivation scenario to show the importance of such a flexibility.

At last, we presented an autonomic approach that considers both the applications and the infrastructure. At the application level, an autonomic manager tries to increase the application's QoS while minimizing the costs due to resource allocation. At the infrastructure level, an autonomic manager seeks for increasing the infrastructure utilization rate and by transitivity improving the energy efficiency.

Due to the absence of information from the infrastructure at the application level and from the application to the infrastructure level, there is a lack of synergy between layers which may lead the system to undesired (or non-optimal) states. This issue is addressed in the following chapter.

Synchronization and Coordination of Multiple Autonomic Managers to Improve Energy Efficiency

When dealing with multiple autonomic managers, it might happen that actions performed by one manager interfere in the managed system of the other one. These interferences exist because of a lack of synergy between autonomic managers, which results in completely independent behaviour although with side-effect consequences. This is an issue in the domain of cloud computing, in which the energy consumption is a concern reserved to the infrastructure provider. In fact application providers are not aware about how application components are deployed on the underlying infrastructure. For example, application providers may request a large piece of resource (virtual machine), which may require a physical machine to be switched on, instead of requesting several small fractions of resources so as to better occupy the infrastructure. This synergy-less behaviour may lead to an energy consumption increase or to a less energy efficient state, which is not desirable for the infrastructure provider. On the other way around, infrastructure providers may be obliged to shutdown part of the infrastructure to cope with energy constraints. Doing that without warning the guest applications could cause serious problems of Quality of Service (QoS) at the application level.

This chapter describes an approach for the synchronization and coordination of multiple autonomic managers to improve the synergy between applications and infrastructure. Firstly, we present a model for synchronization and coordination of multiple autonomic managers. This model helps to understand how autonomic managers communicate and how decisions taken by one manager may interfere in a system managed by another manager. Then, we apply this autonomic model to the context of cloud computing. More precisely, we define an autonomic manager in charge of managing one application (one per-application) and another autonomic manager that is in charge of managing the infrastructure. We describe these two autonomic managers by providing details about their objectives, their autonomic tasks, how they communicate, how they are synchronized and coordinated so as to optimize at the same time the applications QoS and the infrastructure energy efficiency, while mitigating the negative interferences.

Contents

5.1 Synchronization and Coordination of Multiple Autonomic Managers . . .	70
5.1.1 Multiple Autonomic Managers Architecture Model	70
5.1.2 Synchronization and Coordination	73

5.2 Synchronization and Coordination of Application and Infrastructure for Energy Efficiency in Cloud Computing	75
5.2.1 Architecture Overview	77
5.2.2 Infrastructure Manager	77
5.2.3 Application Manager	85
5.3 Summary	96

5.1 Synchronization and Coordination of Multiple Autonomic Managers

Autonomic computing [Hor01] aims at providing self-management capabilities to systems. The managed system is monitored through sensors, and an analysis of this information is used, in combination with knowledge about the system, to plan and execute reconfigurations through actuators (or effectors). Classically, an autonomic manager internal structure is implemented by a MAPE-K (monitoring, analysis, planning and execution tasks over a knowledge base) control loop [KC03]. Bottom-up interactions between the managed system and the autonomic manager are realized via *events* whereas top-down interactions via *actions*.

Our approach aims to provide synchronized and coordinated autonomic managers by introducing a synchronization of the access to a knowledge shared among the autonomic managers and a coordination protocol.

5.1.1 Multiple Autonomic Managers Architecture Model

According to state-of-the-art work on self-adaptive architectures [Lem13], interacting autonomic managers can follow a set of patterns:

- **Hierarchical:** in this pattern, complete MAPE-K control loops (with monitoring, analysis, planning, execution and knowledge) are organized in a hierarchy, in which autonomic managers at lower levels operate at short time scales to solve local problems, whereas the ones at higher levels work at long time scales and have a more global vision of the problems, which is obtained from messages passed from the lower level managers. Examples of this kind of architecture has been used in [MKGdPR12] and [APTZ12].
- **Master/slave or Regional:** in the master/slave pattern, autonomic managers are also organized in a hierarchy, in which, at a lower level, many slaves control loops are responsible only for the monitoring and execution tasks of their respective managed systems, whereas a single master centralizes the analysis and planning tasks of all managed systems. Regarding the regional pattern, only the planning task is centralized at the master. One example of the use of this kind of architectural pattern can be found in [GDA10].
- **Decentralized:** in this pattern, each autonomic task (monitoring, analysis, planning, execution) is coordinated with its counterpart in the peer autonomic manager. It requires a flexible sharing of information among managers so as to allow the coordination in every single task.
- **Information Sharing:** in this pattern, each system is managed by a complete MAPE-K control loop, but they communicate only via the monitoring and execution tasks. The information gathered on the managed system is shared among all the monitors, whereas the analysis and planning is performed without any coordination between peers.

In the context of this thesis work, we are interested in architectures that fulfill the requirements imposed by the cloud service/consumer architecture. The hierarchical pattern does not fit in this kind of architecture, since autonomic managers at higher levels detain a global view (information)

of the whole system. This is not desirable in cloud environments since autonomic managers may belong to different vendors (e.g. in hybrid/public clouds).

The master/slave and regional patterns requires a unique decision module (analysis and planning) for all the autonomic managers, i.e. for all applications and for the infrastructure. It implies in a lack of flexibility in the autonomy of each autonomic manager (e.g. applications that are managed at different time scales) as it requires that all autonomic managers are created by the same vendor, which is not very suitable for cloud environments. Moreover, the concentration of analysis and planning in only one autonomic manager may lead to serious problems of scalability. That is, dealing with several problems (for each one of the managed systems) with a large number of variables at the same time may not be feasible.

The decentralized pattern requires that every autonomic task from one autonomic manager coordinates with its counterpart in the peer manager. It leads to a high level of information sharing in all autonomic tasks (monitor, analysis, planning and execution). While it may be very interesting to cope with conflicting behaviours, it fails in guaranteeing information hiding, which is so important in multi-vendor environments and more specifically in cloud environments.

The information sharing pattern improves the lack of information hiding observed in the decentralized pattern, since it allows communication between autonomic managers only via the monitors and executors. However, it allows a sharing of monitoring data among all monitors, which may not be applicable to cloud environments. For example, it not envisaged that applications are aware about the details of placement of virtual machines (VMs) in physical machines (PMs).

Considering those limitations, we claim that it is necessary a novel autonomic architecture model for cloud systems or systems with similar requirements. Firstly, as in the information sharing pattern, the communication should be done only through the monitoring and execution tasks, but the monitoring data is not shared among all monitors. Instead, only a small fraction of an autonomic manager's knowledge may be shared with others and this sharing may require special synchronization treatment to avoid inconsistent data.

Given these requirements, we define three types of autonomic managers, as follows:

- **Independent:** this type of autonomic manager is completely independent from the others. The source of the received events is always the managed system and the actions are executed only on the concerned system. There is no communication between other managers and the knowledge is entirely internal (private) to the manager in question. Figure 5.1(a) illustrates this kind of autonomic manager.
- **Coordinated:** this type of autonomic manager communicates with the others, as it can be seen in Figure 5.1(b). Detected events may be originated from other managers and actions may trigger notifications on other managers. To this end, a message-based protocol is used to perform the communication among multiple managers. The kinds of events and actions used for that is domain-specific and therefore depends on the managed systems.
- **Synchronized:** sometimes, it is needful to share some information for a collective activity. That is to say that all the autonomic managers may take into consideration that information so as to promote a synergy among them and therefore avoid negative interferences. As the access to shared data may lead to concurrency and consistency problems, managers should be able to access the information in a synchronized manner. Figure 5.1(c) illustrates three synchronized autonomic managers which share some information through a public knowledge.

It is important to remember that our objective is to improve the QoS and the energy efficiency in cloud systems by creating a synergy between them (i.e. among autonomic managers). Therefore, we focus on coordinated and synchronized autonomic managers.

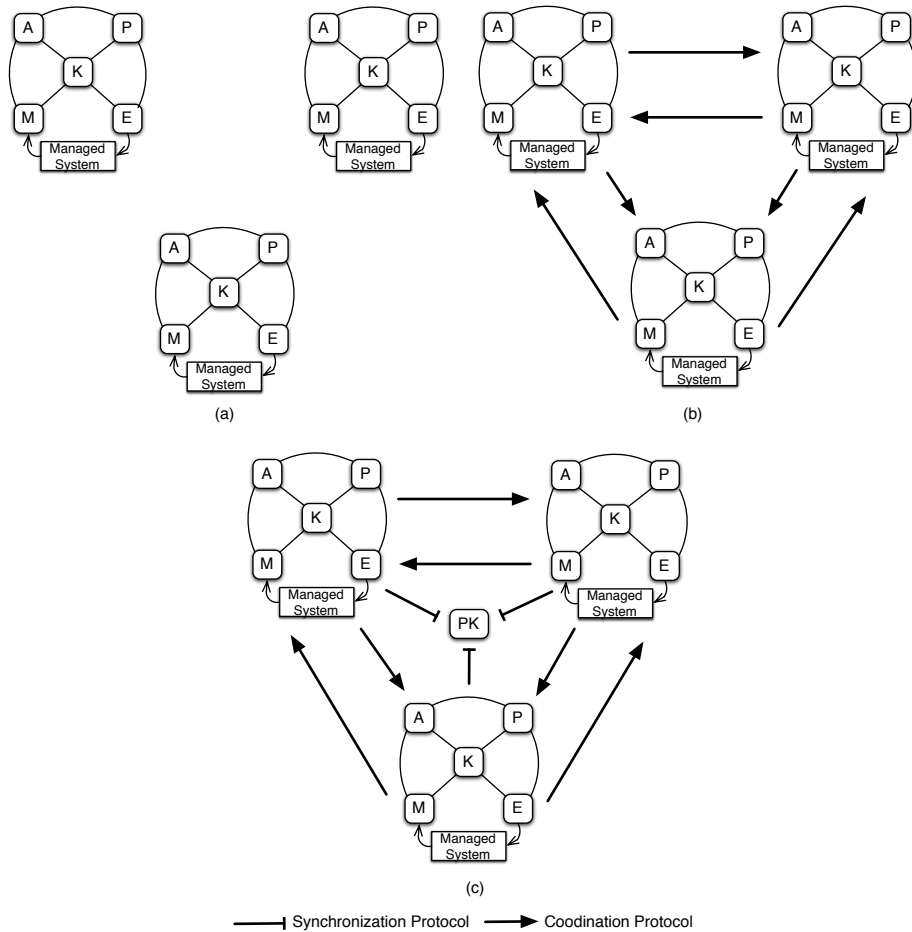


Figure 5.1: Autonomic Manager Types: (a) Independent Autonomic Managers; (b) Coordinated Autonomic Managers; and (c) Synchronized and Coordinated Autonomic Managers.

Public and Private Knowledge Model.

Regarding the information sharing we separate the knowledge base of each autonomic manager into two parts: the **private knowledge**, which stores the internal information needed by the autonomic tasks, and the **public knowledge**, which is shared among other managers. The public knowledge base may have to be synchronized if the actions executed by the autonomic managers require to modify the shared information (directly or indirectly). Indeed, the simultaneous actions of multiple autonomic managers may lead to changes at the public knowledge at the same time (concurrency problem) and may lead to non-logical global results (consistency problem). In our approach, we consider that the autonomic manager that makes the knowledge public is the only one eligible to modify it directly, which restrain the concurrency problem.

Autonomic Actions

In order to clarify the interactions between several autonomic managers, it is important to differentiate actions and events which are part of the managed system and those which are part of the multi-autonomic managers system model. Figure 5.2 depicts a hierarchy of the different types of events and actions.

Actions can be executed on the actual managed system, within the MAPE-K control loop (*intra*loop) or over another MAPE-K control loop (*inter*loop).

The actions on the managed system and the *interloop actions* are always executed by the execution task of the MAPE-K control loop. An *interloop action* may notify another autonomic

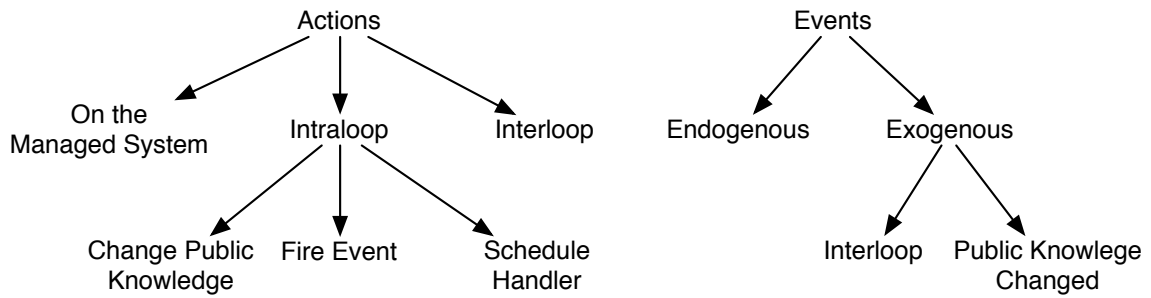


Figure 5.2: Autonomic Actions and Events Hierarchy.

manager as if it was asking for a service and waiting for the response. In this case, the planning task creates and schedules a handler that contains all the other actions that have to be executed in response to this *interloop* action. This *interloop* action is therefore a notifying action, which is taken as an *interloop* event at the target control loop.

Finally, *intraloop* actions can be *Change Public Knowledge*, *Schedule Handler* or *Fire Event* actions. The *Change Public Knowledge* is an action that modifies the content of a public knowledge. As previously said, the *Schedule Handler* action schedules a handler containing a set of actions that depend on a future object to be executed. Finally, the *Fire Event* action is an action that triggers an endogenous event on the same autonomic manager. It may be instantaneous or timed (i.e. triggered after a given time interval).

Autonomic Events

As it is shown in Figure 5.2, an event can be either endogenous or exogenous. The source of endogenous events is always the managed system. The source of exogenous events is another control loop. For the exogenous events, there are the *interloop* events - created by the *interloop* action - and the *Public Knowledge Changed* - created by the *Change Public Knowledge* action.

5.1.2 Synchronization and Coordination

Coordination Protocol

Considering the coordinated case with multiple autonomic managers, we take into consideration what we call the collaboration problem, where two control loops have to communicate in order to accomplish a global activity together. Indeed, the execution task of one MAPE-K control loop may ask another autonomic manager a service and waits its completion. However, in order to avoid blocking states, autonomic managers must communicate in an asynchronous manner. So, the action performed by the first autonomic manager will trigger an *interloop* event at the second manager. At the same time, the first manager creates a handler containing all the actions (as future objects [EFGK03]) that have to be executed after the service has terminated. The *interloop* event is detected by the monitoring task of the second MAPE-K control loop. Once the event is processed, another *interloop* event is sent back to the caller manager.

Figure 5.3 illustrates this *interloop* communication. We rely on a message-oriented middleware (MOM) [Mah04], in which messages are exchanged through a message broker. So, the *interloop* communication can be either peer-to-peer, in which one autonomic manager sends a message to another manager; or one-to-many, where one autonomic manager sends a message to one or several. More concretely, the “sends” and “published” arrows in Figure 5.3 corresponds to an *interloop* action, whereas the “receives” arrow represents an *interloop* event.

Hence, each autonomic manager has a queue registered in the message broker. Any manager willing to send a message to another manager should use the broker and the respective queue to mediate the communication. In the case of message broadcasting, we rely on a publish/subscribe

(PubSub) model [EFGK03], where a manager may create a topic and publish on it, whereas others managers can subscribe to topics so as to be notified every time another manager publishes on it.

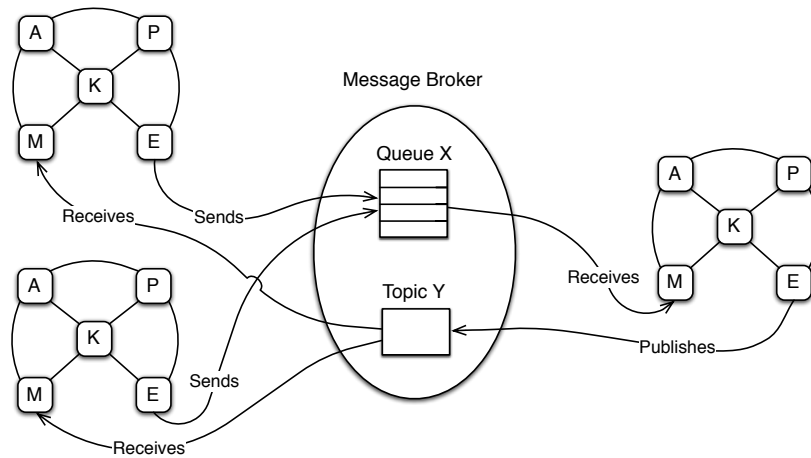


Figure 5.3: Interloop Communication.

Token Protocol for Synchronizing the Public Knowledge

The public knowledge is composed of critical and non-critical sections. The first corresponds to fractions of information that are rarely changed and do not compromise the consistency of the system, whereas the other one refers to pieces of information that may be frequently changed and therefore may interfere in the system consistency. Thus, while the synchronization is dispensable for accessing non-critical sections, it is mandatory for accessing critical ones.

For that reason, we introduce a simple token protocol in order to synchronize the access to the critical sections among multiple autonomic managers. As for transactions in databases, this synchronization protocol ensures that only one autonomic manager can access a critical section. Each critical section is associated with a token, meaning that in order to access a given critical section, managers must request the corresponding token. A manager can get the token by either asking for it explicitly with a *TOKEN REQUEST* message (active token request) or it can receive the token from another manager through a *TOKEN TRANSFER* message (passive token reception). Whenever a manager does not need the token anymore it releases it with a *TOKEN RELEASE* message. Whenever a token is requested, the requester has to wait for a *TOKEN ACQUIRED* message. Each autonomic manager having a public knowledge with critical sections implements a token manager which is in charge of managing the token protocol.

Timing MAPE-K Control Loops

All the MAKE-K control loops evolve in a dynamic environment where multiple events may occur simultaneously. The arrival rate of these events may vary according to the autonomic manager and are stored in a waiting queue. In order to manage the arrival of these events, the monitoring task of each MAPE-K control loop has a scheduling process. This scheduler may implement different policies, some of them may take into account the events priorities. In our approach and for the sake of simplicity, for endogenous events, we consider a FIFO (First-In First-Out) scheduling policy without priorities. However, *interloop events* that result in handlers execution, i.e. events that are response for *interloop actions*, have higher priority. Indeed, handlers contain actions that have to be executed in response to a service request and need to be treated in priority in order for the source event to be considered as treated as soon as possible. Therefore, one event is considered to be treated only if the entire MAPE-K control loop is finished, including the eventual handlers. In

order to formalize the timings, we introduce some notations in Equations 5.1, 5.2 and 5.3. These notations are explained in Table 5.1.

$$T_i^j = T_{lock} + \mu_i^j \quad (5.1)$$

$$\mu_i^j = T_i^{jA} + \rho_i(modif) * (T_i^{jP} + T_i^{jE}) \quad (5.2)$$

$$T_i^{jE} = T_i^{jEactions} + \rho_i(interloop) * (T_{i'}^{j'} + T_i^{jEhandler}) \quad (5.3)$$

Symbol	Description
T_i^j	Time needed to treat event i for control loop j
T_{lock}	Waiting time for the token to be acquired
μ_i^j	Service time for control loop j for event i
$\rho_i(modif)$	Probability to start a planning and execution task (modification of the managed system required)
T_i^{jA}	Analysis task duration for event i and control loop j
T_i^{jP}	Planning task duration for event i and control loop j
T_i^{jE}	Execution task duration for event i and control loop j
$T_i^{jEactions}$	Duration of the first part of the execution task for event i and control loop j
$\rho_i(interloop)$	Probability to ask another control loop a service with an <i>interloop event</i>
$T_{i'}^{j'}$	The duration for another control loop j' to treat the <i>interloop event</i> i'
$T_i^{jEhandlers}$	Duration to execute the handlers for event i in control loop j

Table 5.1: MAPE-K Control Loop Timing Notation.

Figure 5.4 presents a sequence diagram to illustrate how two autonomic managers would use the token synchronization protocol and the coordination protocol. The M-1 to E-1 vertical lines are the tasks of the first MAPE-K control loop, whereas M-2 corresponds to the monitoring task of the second MAPE-K control loop. The TOKEN vertical line shows how managers obtain the token to access one critical section of the public knowledge of the second manager.

The monitoring tasks are continuously listening for events. As we can see, a first endogenous event arrives for MAPE-K control loops 1 and 2. The control loop 1 acquires the token (TReq and TA), launches the analysis (INTRA) and releases the token straight after the event is treated (TRel). The same goes for control loop 2 which acquires the token as soon as it is released by control loop 1. A second endogenous event is treated by control loop 1 which requires a coordination between control loop 1 and control loop 2. First, the loop 1 acquires the token and launches the analysis A-1, the planning P-1 and the execution task E-1. As we can see, control loop 1 sends an exogenous event to control loop 2 (first INTER, exogenous event 2) along with the token (TT). This allows control loop 2 to eventually modify the public knowledge, which is managed by it. As soon as control loop 2 finishes to treat this event, it sends back an exogenous event to control loop 1 (second INTER, exogenous event 4), which allows control loop 1 to execute the handler (INTRA invoke handler) and to finish treating the event 2.

It is noteworthy that the coordination and synchronization approach presented in this chapter has been quantitatively validated through an experimental evaluation (cf. Section 7.2.1).

5.2 Synchronization and Coordination of Application and Infrastructure for Energy Efficiency in Cloud Computing

The cloud computing architecture is typically defined as a stack of several inter-dependent systems, in which systems on lowermost layers are service providers to systems on uppermost lay-

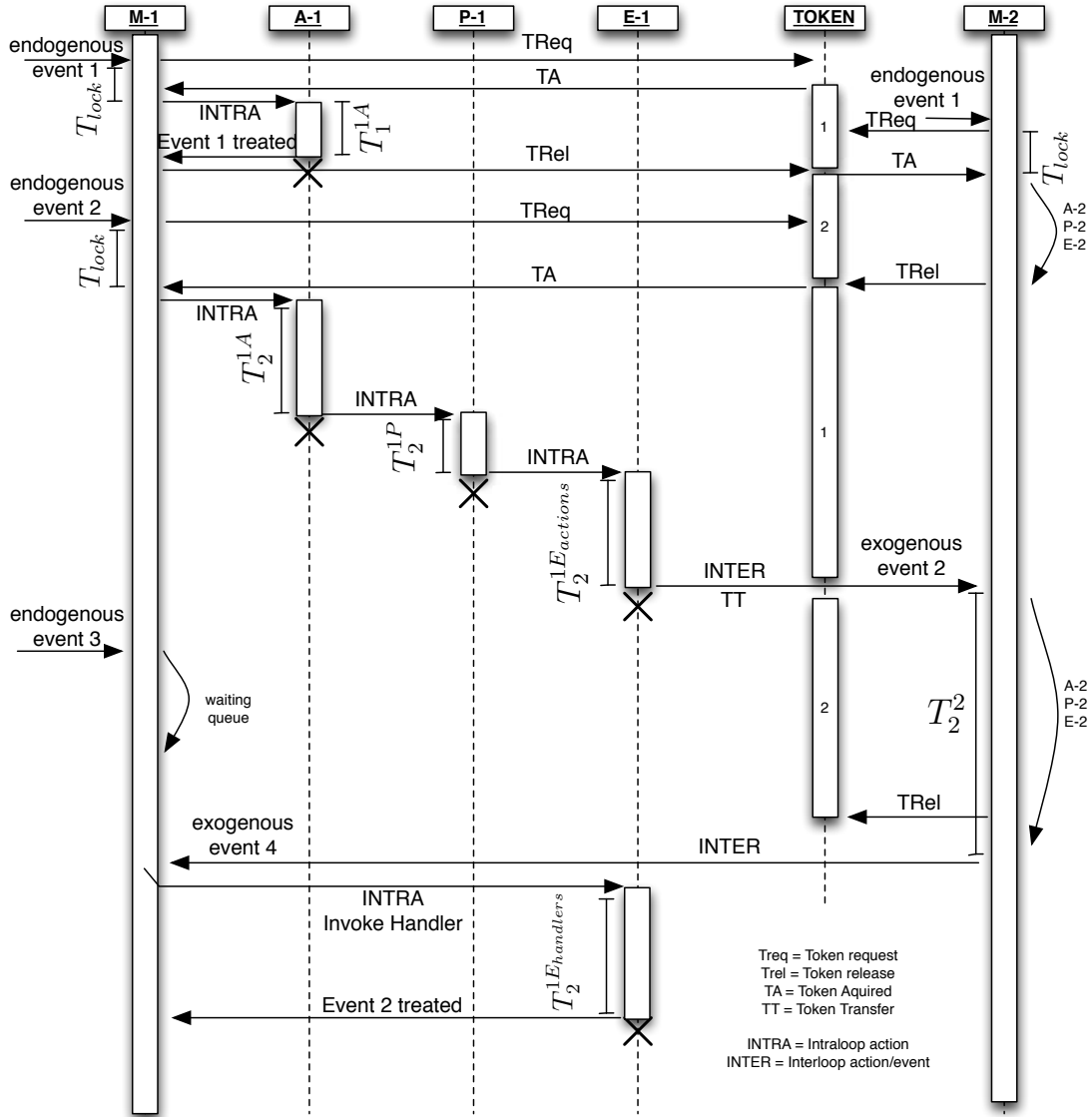


Figure 5.4: Autonomic Managers Coordination and Token Synchronization Protocols and Timings.

ers. Our autonomic architecture in the context of cloud computing consists of two types of inter-dependent managed systems: applications and infrastructure.

The infrastructure consists of a set of physical machines (PMs), whose computing resources (CPU and RAM) are made available by means of virtual machines (VMs). There might be one or several classes of VM, each one with a different CPU or RAM capacity. The use of each VM instance is chargeable according to the amount of resources allocated to it and the usage amount of time. Finally, the infrastructure provider may give a limited number of discounts on VMs in order to attract application providers to occupy portions of resources that are not being utilized and therefore improve the infrastructure utilization rate.

An application is defined as a set of components. Each component offers a set of services, which, in turn, might depend on a set of references to services offered by other components. Services are bound to references through bindings. The application may thus operate in different architectural configurations (architectural elasticity, cf. Section 4.3.2), which are determined by the subset of components used and how they are linked to each other. In addition, for scalability purposes, application providers may deploy the same application component on one or more VMs,

that is, for each component there might be one or several instances which are used in a load balancing manner (capacity elasticity, cf. Section 4.3.1) to scale in. Finally, applications' QoS are evaluated in terms of performance, i.e. the application responsiveness when dealing with a given number of simultaneous requests; and architecturally, i.e. a quality degree referred to the current architectural configuration.

The rest of this section is organized as follows: first, we present an overview of our cloud computing multi-autonomic management architecture. Then, we provide details about each autonomic manager that takes part of this architecture, namely the application manager and the infrastructure manager.

5.2.1 Architecture Overview

This section presents an architecture overview of our approach. The approach is composed of two type of autonomic managers: one dealing with the infrastructure, namely infrastructure manager (IM); and another dealing with the applications (per-application), namely application manager (AM), as it is shown in Figure 5.5.

AMs aim at minimizing the amount of VMs needed to keep the level of QoS as high as possible (capacity elasticity). Furthermore, AMs are able to adapt their application's architecture (architectural elasticity) in order to cope with resource restriction imposed by the IM or other restriction concerning the application itself. More precisely, AMs monitor/listen for events that come either from the application itself or from the IM; analyze whether or not it is necessary to reconfigure the application by considering the runtime context (e.g. the workload, the current architectural configuration, the current mapping of components to VMs, etc.); elaborate a reconfiguration plan; and execute actions corresponding to the reconfiguration plan.

Regarding the IM, apart from dealing with requests sent by AMs on VMs, its objective is to optimize the placement of these incoming VMs in the PMs so that it is possible to reduce the number of PMs switched on and consequently reduce the energy consumption. To this end, the IM monitors/listens for events that come either from the infrastructure itself (e.g. PMs utilization rate) or from the AMs; analyzes whether or not it is necessary to replace or to change its current configuration by considering the runtime context (e.g. the current mapping VMs to PMs); plans and executes the reconfigurations.

As previously mentioned, the lack of synergy among multiple autonomic managers may result in negative interferences or side-effect actions. In this context, in order to improve the synergy, we can apply the coordination and synchronization protocols presented in Section 5.1. The coordination protocol defines a set of messages exchanged by autonomic managers which are transformed into actions and events, and used for instance by the IM to inform AMs about energy shortage at the infrastructure level. The synchronization protocol defines a set of public knowledge (critical) sections that are shared with autonomic managers. For instance, the IM can change the VMs renting fees by putting some VMs in promotion. The shared knowledge is used by AMs to take into consideration those changes in order to globally take better decisions.

5.2.2 Infrastructure Manager

This section describes the IM by detailing each one of its autonomic tasks. Firstly, we present the knowledge model which is divided into a public and a private parts. Then, we present the set of events and actions used by the IM. We present how the analysis task is modeled by detailing each analyzer used by the IM. Finally, we show how reconfigurations take place by detailing the planning task.

Knowledge Model

Public Knowledge. The public knowledge is composed of critical and non-critical sections. In the case of the IM, there is one non-critical section corresponding to the rental fees. It is expressed

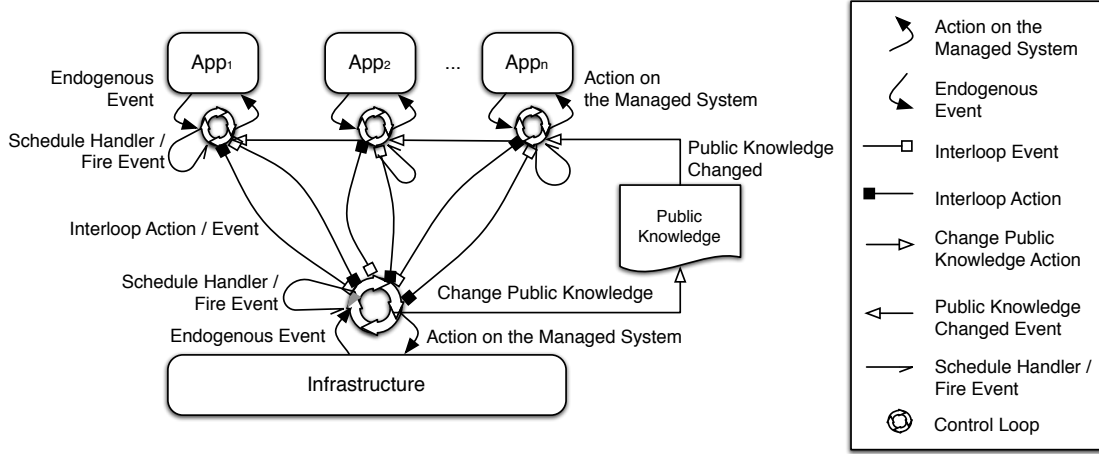


Figure 5.5: Multi-control loop Architecture for Cloud Computing.

by the set of VM classes $M = \{m_i \mid i \in [1, q]\}$, where each element $m_i \in M$ is defined in terms of CPU capacity (m_i^{cpu}), RAM capacity (m_i^{ram}) and renting cost (m_i^{cost}).

The IM may decide to create promotions in PMs whose utilization rate is lower than a given threshold so as to stimulate applications to occupy this piece of under-utilized resource. The objective is to establish a synergy between applications and infrastructure so as to increase the infrastructure utilization rate, i.e., the ratio of the amount of resources used to the amount of resources available. To this end, we define the set $Pr = \{pr_i \mid i \in [1, \xi]\}$ as the set of available promotions, in which each promotion $pr_i \in Pr$ is defined as a tuple $(pr_i^{pack}, pr_i^{vms}, pr_i^{price}, pr_i^{token})$. pr_i^{pack} corresponds to a pack of offered VMs $pr_i^{pack} = \{pack_{ij} \mid j \in [1, q]\}$, where $pack_{ij} \in \mathbb{N}^*$ corresponds to the number of VMs of class m_j in the pack. $pr_i^{vms} \subset V$ is a set of the actual VMs taking part of promotion pr_i , that is, these VMs are instantiated in the model, but not actually created in the infrastructure. This is done in order to make the model aware that the PMs in which the promotions are created are actually occupied with those (not yet launched VMs). pr_i^{price} corresponds to the global discount price for the promotion pr_i and is determined according to a fixed discount rate (δ) of the regular price, as it is defined in Equation 5.4.

$$pr_i^{price} = (1 - \delta) * \sum_{j=1}^q m_j^{cost} * pack_{ij} : \forall i \in [1, \xi] \quad (5.4)$$

For example, let us suppose that one PM with four CPU units and four RAM units capacity is half occupied, that is, it has 2 CPU units and 2 RAM units available. The IM may decide to create a promotion in this PM so as to fill it up and thus increase the utilization rate. Let us also suppose that VM class m_1 is available and that $m_1^{cpu} = 1, m_1^{ram} = 1$ and $m_1^{cost} = 0.18$. So, supposing a discount rate $\delta = 0.3$ (30% percent), a promotion pr would be composed of a pack pr^{pack} containing two VMs of class m_1 , a unique and randomly created token pr^{token} and a reduced price $pr^{price} = (1 - 0.3) * 2 * 0.18 = 0.252$, instead of $pr^{price} = 0.36$ (regular price).

The IM might manage several critical sections, each one corresponding to a different promotion $pr_i \in Pr$. In fact, as each promotion can be given to only one client, the access to promotions should be synchronized. So the token manager generates a token pr_i^{token} every time an AM wants to access a promotion pr_i (TOKEN REQUEST message). Once the promotion is available (in an exclusive manner), the token manager responds back with a TOKEN ACQUIRED message containing the generated token pr_i^{token} . The AM can either release the token (TOKEN RELEASE message) if the promotion is no longer useful for the concerned application, or transfer the token (TOKEN TRANSFER message) if the AM wants to get the promotion.

Private Knowledge. The private knowledge is composed of an infrastructure model and the runtime context, which are used internally by the analysis and planning tasks.

The infrastructure model consists of a set of PMs $P = \{pm_i \mid i \in [1, p]\}$. Each element $pm_i \in P$ is a tuple $(pm_i^{cpu}, pm_i^{ram}, pm_i^{on})$, where pm_i^{cpu} and pm_i^{ram} corresponds to the CPU and RAM capacities of PM pm_i , and $pm_i^{on} \in \{true, false\}$ indicates whether the PM is powered on.

Runtime Context. The infrastructure runtime context is define as a tuple $\mathcal{C} = (P, V, A, H, Pr)$, where:

- P is the set of the infrastructure PMs.
- $V = \{vm_i \mid i \in [1, v]\}$ represents the set of VMs currently executing in the infrastructure. Each element $vm_i \in V$ corresponds to a tuple $(vm_i^{cpu}, vm_i^{ram}, vm_i^{app})$, where vm_i^{cpu} and vm_i^{ram} corresponds respectively to the CPU and RAM capacities of vm_i , and vm_i^{app} corresponds to the application owner $vm_i^{app} \in A$ of the VM.
- $A = \{a_i \mid i \in [1, \psi]\}$ represents the set of guest applications. Each element $a_i \in A$ corresponds to an unary tuple (a_i^{rr}) , where a_i^{rr} corresponds to the resource reduction rate application a_i agrees to be subject to. This information is useful when the infrastructure is facing shortage situations such as energy brownouts or programmatic energy efficiency policies. Hence, the IM can benefit of that information to choose the right parts of the infrastructure to shutdown and meet while meeting the resource/energy constraints.
- $H_{p \times v}$ is an incidence matrix where

$$H_{ij} = \begin{cases} 1 & \text{if } vm_j \text{ is hosted by } pm_i \\ 0 & \text{otherwise} \end{cases} \quad (5.5)$$

- Pr is the set of available promotions.

Events and Actions

This section presents the possible events and actions for the IM according to the hierarchy illustrated in Figure 5.2. We present also the possible scenarios that maps events to actions. That is, upon the detection of an event, which actions can be performed.

VMs Requested. It is an *interloop event* that happens when some AM performs a *Request VMs interloop action* (cf. Section 5.2.3). This event comprises a set of requested VMs $R^{vm} = \{rvm_i \mid i \in [1, q]\}$, where each element rvm_i corresponds to the number of requested VMs of class m_i ; and a set of requested promotions $R^{pr} \subseteq Pr$. For every requested promotion $pr \in R^{pr}$, there must be a corresponding token that is transferred (TOKEN TRANSFER) within the message. So, it might happen that one of these sets is empty, i.e., if there is only request for promotions ($R^{vm} = \emptyset$) or if there is no promotions in the request ($R^{pr} = \emptyset$). This event triggers the analysis task that evaluates the placement of the requested VMs on the PMs so as to minimize the number of PMs needed. The planning task takes the result of the analysis and translates it into a set of *Switch On PM*, *Create VM* and *Activate Promotion* actions on the infrastructure. Finally, it notifies the AM that requested the VMs by executing a *Notify VMs Created interloop action* (Figure 5.6 (a)).

VMs Released. It is also an *interloop event* that happens when some AM performs a *Release VMs interloop action* (cf. Section 5.2.3). It contains a list of VMs that should be destroyed $R^{vms} \subseteq V$. As there is no special analysis for this kind of event, the planning task is directly triggered from the monitoring. The planning task translates the list of VMs to be released into a set of *Destroy VM* actions on the infrastructure (Figure 5.6 (b)).

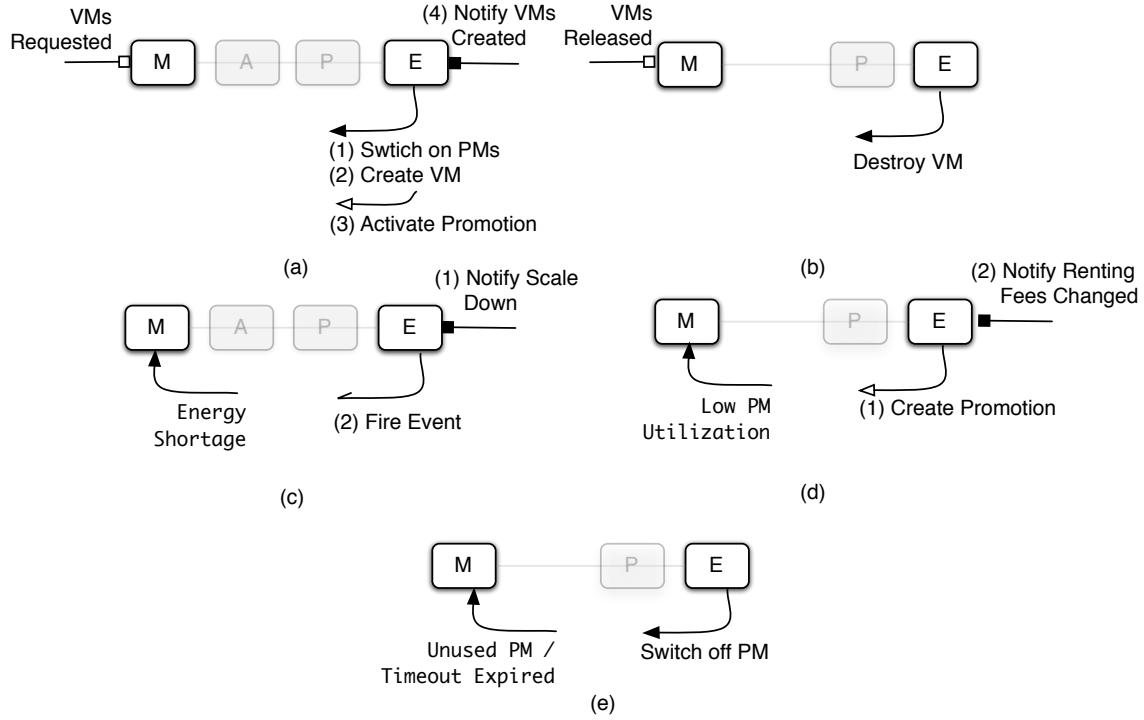


Figure 5.6: Infrastructure Manager Events and Actions.

Energy Shortage. It is an endogenous event that comes from the infrastructure context data (e.g. power meters or an announcement of energy unavailability). It comprises the power consumption availability W which the IM must cope with or the number of PMs to shutdown $P^\#$. Following this event, the IM triggers the analysis task to pack the running VMs in as less PMs as possible as well as to determine which PMs should be switched off. The planning task takes the result of the analysis task and translates it into a set of *Notify Scale Down* interloop actions to notify the concerned AMs about the constraints on the infrastructure (Figure 5.6 (c)). Finally, the planning task creates a *Fire Event* action, whose objective is to trigger the endogenous event *Timeout Expired* in a timely manner, that is, after a given timeout. This event contains the PMs to shutdown ($P^{off} \subseteq P$). Once this event is detected, i.e. after the timeout has expired, the planning task takes the list of PMs to shutdown and translate it into a set of *Switch Off PM* actions on the infrastructure (Figure 5.6 (e)).

Low PM Utilization. It is also an infrastructure endogenous event which is detected anytime a PM pm_i has been under-utilized (when its utilization rate is lower than U_{min}) during a certain period of time t_l , as it is formalized in Equation 5.6. As it can be seen in Figure 5.6(d), the detection of this event triggers directly the planning task that creates, for each concerned PM, a set of actions *Create Promotion* (Change Public Knowledge), followed by a *Notify Renting Fees Changed* interloop action, which notifies the subscribed AMs that the public knowledge has been changed.

$$\min\left(\frac{\sum_{j=1}^v H_{ij} * vm_j^{ram}}{pm_i^{ram}}, \frac{\sum_{j=1}^v H_{ij} * vm_j^{cpu}}{pm_i^{cpu}}\right) < U_{min} \text{ holds during } t_l \quad (5.6)$$

Likewise, *Unused PM* is an infrastructure endogenous event which is detected anytime a PM has not hosted any VM during a certain period of time t_u , as expressed by Equation 5.7. Upon the detection of this event, the involved PMs ($P^{off} \subseteq P$) are directly switched off. That way, there is no need for an analysis task. The IM directly triggers the planning task, which creates a set of *Switch Off PM* actions on the infrastructure (Figure 5.6 (e)).

$$\min\left(\frac{\sum_{j=1}^v H_{ij} * vm_j^{ram}}{pm_i^{ram}}, \frac{\sum_{j=1}^v H_{ij} * vm_j^{cpu}}{pm_i^{cpu}}\right) = 0 \text{ holds during } t_u \quad (5.7)$$

Analysis

Upon the detection of an event in the monitoring task, the IM may trigger the analysis task or directly the planning or execution task, if no analysis is required. The analysis task aims at solving a specific problem regarding the current state of the infrastructure by proposing a new infrastructure configuration. In the IM, the analysis task consists of two different analyzers, which aim at solving three different problems separately, according to the event detected in the monitoring task. All the analyzers (*VM Placement Analyzers* and *Energy Shortage Analyzers*) model their problems as a constraint satisfaction and optimization problems (CSOP) and relies on constraint programming (CP) [RVBW06, RVBW07] to solve them.

VM Placement Analyzer. This analyzer is triggered upon the detection of a *VMs Requested* event and aims at solving a VM placement problem, that is, its objective is to place the requested VMs in the PMs so as to minimize the number of PMs needed. It is modeled in a similar way as in [HLM⁺09] and in [VTM10], except for the fact that in those approaches, the existing VMs can be migrated from one PM to another so as to optimize the infrastructure utilization, whereas in this work we avoid migration operations.

One can make an analogy between this analyzer and a restaurant host service, whose role is to place arriving clients in their respective tables while minimizing the number of regions within the restaurant so as to reduce the number of waiters needed. The restaurant corresponds to the infrastructure, the regions to PMs and tables to VMs. It may be considered annoying for clients to be re-allocated another table during the meal in order to allow the restaurant to minimize the number of regions. Likewise, it might be considered annoying for applications providers to have their VMs migrated to another host PM if it impacts the QoS of applications. This analyzer is modeled as follows. Let $V' = \{vm_i \mid i \in [0, v'], v' = \sum_{j=1}^q rvm_j\}$ be the set of new requested VMs (R^{vms}), collapsed in a set of instantiated VMs. That is, R^{vms} corresponds to a set containing the number of requested VMs of each class, whereas V' corresponds to the VMs themselves instantiated.

First, we define the decision variable h_{ij} , where $D(h_{ij}) \in \{0, 1\} \cdot \forall i \in [1, p]$ and $\forall j \in [1, v']$ which indicates whether PM pm_i hosts VM vm_j .

Secondly, we define a set of constraints over variable h_{ij} . The constraints 5.8 and 5.9 state that the CPU and RAM demands of VMs hosted in one PM should not exceed its available CPU and RAM capacities. It is noteworthy that this analyzer aims at placing the new requested VMs in the minimum number of PMs by meeting the resource constraints. However, it is not intended to pack VMs on PMs by rearranging running VMs in other PMs.

$$pm_i^{cpu} - \sum_{j=1}^v H_{ij} * vm_j^{cpu} \geq \sum_{j=1}^{v'} h_{ij} * vm_j^{cpu} : \forall i \in [1, p] \quad (5.8)$$

$$pm_i^{ram} - \sum_{j=1}^v H_{ij} * vm_j^{ram} \geq \sum_{j=1}^{v'} h_{ij} * vm_j^{ram} : \forall i \in [1, p] \quad (5.9)$$

$$(5.10)$$

Finally, the objective is to minimize the number of PMs necessary to host the VMs instances (V'), as it can be seen in Equation 5.11.

$$\text{minimize}\left(\sum_{i=1}^p (u_i)\right), \text{ where } u_i = \begin{cases} 1, \exists j \in [1, v'] \mid h_{ij} = 1 \\ 0, \text{ otherwise} \end{cases} \quad (5.11)$$

Energy Shortage Analyzer. This analyzer is triggered upon the detection of an *Energy Shortage* event and aims at finding a set of PMs to be shutdown so as to cope with an energy shortage situation. The solution should meet the constraints in terms of resource reduction rate stated by each application and minimum number of PMs to shutdown, while minimizing the impact of resource restriction imposed to applications.

First, we define the decision variable z_i , where $D(z_i) \in \{0, 1\} \cdot \forall i \in [1, p]$ which indicates whether or no a PM $pm_i \in P$ should be shutdown.

The second part consists of a set of constraints over the decision variables z_i . Equations 5.12 and 5.13 are auxiliary expressions used to represent the current amount of resource allocated to a given application a_k and the amount of resources to be released after shutting off the PMs specified in variables z . Equation 5.14 states that for each application $a_k \in A$ the amount of resources to release (RR_k) must not exceed the current amount of resources allocated to it (CR_k) reduced by the resource restriction rate (a_k^{rr}). Lastly, Equation 5.15 states that at least $P^\#$ PMs must be shutdown in order to cope with the energy shortage.

$$CR_k = \sum_{j=1}^v u_j \text{ where } u_j = \begin{cases} vm_j^{cpu}, & \iff vm_j^{app} = a_k \\ 0, & \text{otherwise} \end{cases} \quad (5.12)$$

$$RR_k = \sum_{j=1}^v u_j \text{ where } u_j = \begin{cases} vm_j^{cpu}, & \exists i \in [1, p] (H_{ij} = 1 \wedge z_i = 1) \wedge \\ & vm_j^{app} = a_k \\ 0, & \text{otherwise} \end{cases} \quad (5.13)$$

$$CR_k * (1 - a_k^{rr}) \geq RR_k \cdot \forall i \in [1, \psi] \quad (5.14)$$

$$\sum_{i=1}^p z_i \geq P^\# \quad (5.15)$$

The objective is to minimize the resource restriction imposed to applications, as formalized in Equation 5.16. In other words, the analyzer tries to choose the PMs to be shutdown by minimizing the amount of resources to be released by applications (VMs).

$$\text{minimize} \left(\sum_{i=1}^{\psi} RR_i \right) \quad (5.16)$$

Notice that in this model, we use the number of PMs to be shutdown ($P^\#$) as the input parameter. In order to be able to deal with other parameters (e.g. the power consumption reduction), it is necessary to re-design the Equation 5.15 so as to accommodate the correct unit (e.g. power consumption of each PM) instead of the number of PMs.

Planning

The planning task can be triggered after the execution of the analysis task or directly after the detection of an event by the monitoring task. This task takes into consideration the infrastructure's current state and the desired state resulting from the monitoring or analysis task to create the IM autonomic actions (see Section 5.2.2) in a meaningful fashion.

According to Figure 5.6, there are five different planners, each one to deal with a different event. All of them take as input a set of variables and returns a plan (object of type *Plan*). The object plan aggregates a set of activities, each one containing one or several other activities or autonomic actions. These activities can be executed in sequence (object of type *Sequence*) or in parallel (an object of type *Fork*).

Algorithm 1 presents the *VM Placement Planner*, that is, the planning triggered after *VM Placement Analyzer*, which is used to deal with *VMs Requested* events. The algorithm takes as input the requested promotions (R^{pr}), the current runtime context (C), new mapping VMs/PMs (h), and the set of new VMs (V'). The algorithm can be split into two parts: the creation of

isolated VMs (lines 5-11) and the activation of promotions (lines 12-18), that is, the creation of VMs within promotions.

Firstly, the algorithm creates the plan (line 1) and the activity objects for VMs and promotions creation (lines 2 and 3). The first part of the algorithm iterates the vector of PMs P (lines 5-11) and the vector of VMs V' (lines 6-10) in order to create *CreateVM* actions and add them to a sequence of actions (*seqPlan*). In addition, it adds all the created VMs to a set of VMs (line 10) that is going to be sent to the requesting AM as a notification. It then adds this sequence to a fork activity (line 11) to be executed in a parallel way along with the other sequences.

The second part of the algorithm iterates over the set of requested promotions R^{pr} (lines 12-18) and over the vector of VMs pr^{vms} that each promotion aggregates (lines 14-16). Similarly, it creates a sequence of actions *CreateVM* (line 15) and adds each created VM to the list of created VMs vms (line 16). An *ActivatePromotion* action is created and added to the sequence (line 17), which in turn, is added to a fork activity (line 18) that is in charge of executing all the promotion activations in parallel. Finally, the algorithm adds both fork activities to the plan (line 19 and 20) and creates a *NotifyVMsCreated* action, which is also added to the plan (line 21).

Algorithm 1: Virtual Machine Placement Planner.

Input: R^{pr} : requested promotion.
Input: C : the current runtime context.
Input: h : the new mapping VMs/PMs.
Input: V' : the new set of VMs.
Result: $plan$: the resulting plan

```

1  $plan \leftarrow$  new instance of Plan
2  $vmsCreationFork$  new instance of Fork
3  $promoCreationFork$  new instance of Fork
4  $vms \leftarrow \emptyset$  // iterates over the set of PMs
5 for  $i \leftarrow 1$  to  $p$  do
6    $seqPlan \leftarrow$  new instance of Sequence
   // iterates over the set of incoming VMs
7   for  $j \leftarrow 1$  to  $t$  do
8     // if  $vm_j$  will be hosted by  $pm_i$ 
9     if  $h_{ij} = 1$  then
10      // creates the VM in the physical infrastructure
11       $add(CreateVM(vm_j), seqPlan)$ 
12       $vms \leftarrow vms \cup vm_j$ 
13    $add(seqPlan, vmsCreationFork)$ 
14 // iterates over the set of requested promotions
15 foreach  $pr \in R^{pr}$  do
16    $seqPlan \leftarrow$  new instance of Sequence
17   foreach  $vm \in pr^{vms}$  do
18     // creates each VM in the promotion
19      $add(CreateVM(vm), seqPlan)$ 
20      $vms \leftarrow vms \cup vm$ 
21    $add(ActivatePromotion(pr), seqPlan)$ 
22    $add(seqPlan, promoCreationFork)$ 
23  $add(vmsCreationFork, plan)$ 
24  $add(promoCreationFork, plan)$ 
25  $add(NotifyVMsCreated(vms), plan)$ 

```

Algorithm 2 describes the *VM Release Planner*, which is triggered directly upon the detection of a *VMs Released* event. It takes as input the set of VMs that have to be released R^{vms} and the

current runtime context \mathcal{C} . It starts by creating the plan and a fork activity objects (lines 1 and 2). Then it iterates over R^{vms} while creating a set of *DestroyVMs* actions that should be executed in a parallel way (line 4). Lastly, the algorithm adds to the plan the fork activity (line 5).

Algorithm 2: Virtual Machine Release Planner.

Input: R^{vms} : the VMs to be released
Input: \mathcal{C} : the current runtime context.
Result: $plan$: the resulting plan

- 1 $plan \leftarrow$ new instance of *Plan*
- 2 $releaseFork \leftarrow$ new instance of *Fork*
- 3 **foreach** $vm \in R^{vms}$ **do**
- 4 \lfloor $add(DestroyVM(vm), releaseFork)$
- 5 $add(releaseFork, plan)$

Algorithm 3 describes the *Promotion Planner*, which is triggered directly upon the detection of *Low PM Utilization* events. The algorithm takes as input the set of PMs P^{pr} in which the promotions should be created, the runtime context \mathcal{C} , the VM renting fees M and the promotion reduction rate δ . The algorithm starts by creating a plan and an activity objects (lines 1 and 2). It then finds the smallest VM class among the renting fees (line 4). We presume that several small VMs costs more than a single big one with the same amount of resources and therefore can be more rental to the infrastructure provider. The algorithm iterates over the set of PMs (lines 5-22) and over the existing VMs (lines 10-13) in order to calculate the remaining amount of resource in each PM. It then calculates the number of VMs to be created by dividing the available amount of resources by the resource capacity of the smallest VM class (line 14). It instantiates the VMs while adding them to a set of VMs (lines 15-16). It fills the promotion pack of VMs, which is zero for all VM classes except for m_{small} (line 17). Then, it calculates the discount price, based on the regular price of this pack of VMs and the discount rate of δ (line 19). The new promotion is then created (lines 21), and the *CreatePromotion* actions are added to the fork activity to be executed in parallel (line 22). Finally, the fork activity is added to the plan (line 23), along with a *NotifyRentingFeesChanged* action (line 24), which publishes the promotions so the subscribed AMs can be notified.

Algorithm 4 describes the *PM Shutdown Planner*, which is triggered directly upon the detection of an *Unused PM* or an *Timeout Expired* event. It takes as input the vector of PMs to be switched off P^{off} and the current runtime context (\mathcal{C}). It starts by creating a plan and fork objects (lines 1 and 2). It then iterates over the set of PMs while creating a set of *SwitchOffPM* actions that are added to the fork activity to be executed in parallel (lines 3-4). Lastly, the fork activity is added to the plan (line 5).

Algorithm 5 describes the *Energy Shortage Planner*, which is triggered after the *Energy Shortage Analyzer*, which in turn, is triggered upon the detection of an *Energy Shortage* event. It takes as input the set of PMs (P^{off}) to shutdown, the timeout after which they should be shutdown (*timeout*), and the runtime context \mathcal{C} . It starts by creating a plan and a fork objects (lines 1 and 2). It then iterates over the set of applications A while gathering the set of affected VMs belonging to the applications in question (line 4). This set is used by the *NotifyScaleDown* action (line 5) to inform applications which VMs should be released. Then it adds this set of notification actions to the plan (line 6), creates an action that will fire the *Timeout Expired* containing the PMs to be shutdown after the timeout (line 7) and add it to the plan. It is important to recall that the detection of the *Timeout Expired* will trigger a *Physical Machine Shutdown Planner*, which is described in Algorithm 4.

Algorithm 3: Promotion Planner.

Input: P^{pr} : the PMs in which promotions should be created.
Input: \mathcal{C} : the current runtime context.
Input: M : infrastructure rental fees.
Input: δ : promotion reduction rate.
Result: $plan$: the resulting plan

```

1  $plan \leftarrow$  new instance of  $Plan$ 
2  $creationFork \leftarrow$  new instance of  $Fork$ 
3  $newPromos \leftarrow \emptyset$ 
4  $small \leftarrow j \mid j \in [1, q] \wedge \forall m_i \in M (\max(m_i^{cpu}, m_i^{ram}) \geq \max(m_j^{cpu}, m_j^{ram}))$ 
   // iterates over the sub set of PM in which promotions should
   // be created
5 for  $i \leftarrow 1$  to  $\xi$  do
6   if  $pm_i \in P^{pr}$  then
7      $vms \leftarrow \emptyset$ 
8      $availableCPU \leftarrow pm_i^{cpu}$ 
9      $availableRAM \leftarrow pm_i^{ram}$ 
   // iterates over the set of VMs to calculate the
   // remaining capacity of  $pm_i$ 
10    for  $j \leftarrow 1$  to  $v$  do
11      if  $H_{ij} > 0$  then
12         $availableCPU \leftarrow availableCPU - vm_j^{cpu}$ 
13         $availableRAM \leftarrow availableRAM - vm_j^{ram}$ 
   // calculates how many VMs of type  $small$  fits in  $pm_i$ 
14     $nbVMs \leftarrow \min(\frac{availableCPU}{m_{small}^{cpu}}, \frac{availableRAM}{m_{small}^{ram}})$ 
   // for each new VM instantiate an object VM
15    for  $k \leftarrow 1$  to  $nbVMs$  do
16       $vms \leftarrow vms \cup$  new instance of VM of class  $m_{small}$ 
   // zero for all classes except for  $m_{small}$ 
17     $pr^{pack} \leftarrow \{pack_j \mid \forall j \in [1, q]\}$  where
      $pack_j = nbVMs$  if  $small = j$ ,  $pack_j = 0$  otherwise
18     $pr^{vms} \leftarrow vms$ 
19     $pr^{price} \leftarrow nbVMs * m_{small}^{cost} * (1 - \delta)$ 
20     $pr^{token} \leftarrow \emptyset$ 
21     $newPromos \leftarrow newPromos \cup \{(pr^{pack}, pr^{vms}, pr^{price}, pr^{token})\}$ 
22     $add(CreatePromotion((pr^{pack}, pr^{vms}, pr^{price}, pr^{token}), pm_i), creationFork)$ 
23  $add(creationFork, plan)$ 
24  $add(NotifyRentingFeesChanged(newPromos), plan)$ 

```

5.2.3 Application Manager

This section describes the AM by detailing each one of its autonomic tasks. First, we present the knowledge model. Then we present the set of events and actions used by the AM. We describe the analysis task by detailing the model of each analyzer used by the AM. Finally, we detail how reconfigurations take place by describing the planning task and the algorithms that implement it.

Knowledge Model

Unlike the IM, the AM has only a private knowledge, which is composed of the application model and the runtime context.

Algorithm 4: Physical Machine Shutdown Planner.

Input: P^{off} : PMs to be switched off.
Input: \mathcal{C} : the current runtime context.
Result: $plan$: the resulting plan.

- 1 $plan \leftarrow$ new instance of $Plan$
- 2 $switchOffFork \leftarrow$ new instance of $Fork$
- 3 **foreach** $pm \in P^{off}$ **do**
- 4 $\lfloor add(SwitchOffPM(pm), switchOffFork)$
- 5 $add(switchOffFork, plan)$

Algorithm 5: Energy Shortage Planner.

Input: P^{off} : PMs to be switched off.
Input: $timeout$: timeout to shutdown PMs.
Input: \mathcal{C} : the current runtime context.
Result: $plan$: the resulting plan.

- 1 $plan \leftarrow$ new instance of $Plan$
- 2 $notificationFork \leftarrow$ new instance of $Fork$
- 3 **foreach** $app \in A$ **do**
- 4 // VMs affected by the PMs to be shutdown
 $vms \leftarrow \{vm_j \mid vm_j^{app} = app \wedge \forall i \in [1, p](\forall j \in [1, v](H_{ij} = 1 \wedge pm_i \in P^{off}))\}$
- 5 $\lfloor add(NotifyScaleDown(app, vms), notificationFork)$
- 6 $add(notificationFork, plan)$
- 7 $add(FireEvent(TimeoutExpired(P^{off}), timeout), plan)$

Application Model. The application is composed of a set of components $C = \{c_i \mid i \in [1, n]\}$. Each component c_i corresponds to a tuple (c_i^r, c_i^s) . c_i^r corresponds to set of references, i.e. functional dependencies on other components services. c_i^s is the set of services offered by the component.

The application might have one or several architectural configurations $K = \{k_i \mid i \in [1, m]\}$. Each configuration $k_i \in K$ is composed of a tuple $(k_i^c, k_i^{deg}, k_i^{bind})$, where $k_i^c \subseteq C$ corresponds to the components involved in the configuration, $k_i^{deg} \in [0, 1]$ corresponds to the QoS degradation due to architectural configuration k_i . k_i^{bind} is a set of pairs $\{(r, s) \mid r \in \bigcup_{i \in [1, n]} c_i^r, s \in \bigcup_{i \in [1, n]} c_i^s\}$ that represents the set of binding between references and services.

For each component $c_i \in C$, there is a function $perf_i : \Lambda \times \mathbb{N}^* \times \mathbb{N}^* \mapsto \mathbb{N}^* : \forall i \in [1, n]$ that maps the workload, amount of CPU and RAM to the service response time, where $\Lambda = \{\lambda_1, \dots, \lambda_u\}$ is a set of workload thresholds.

The application provider establishes a Service Level Agreement (SLA) with the application consumer. It states that there are two levels of response time: (i) an ideal one (rt^{ideal}), under which the service is not considered degraded; and (ii) a acceptable one rt^{acc} under which the service is considered degraded at the rate $rt^{deg} \in [0, 1]$ and above which the service is considered not available. The performance degradation is illustrated in Figure 5.7.

The SLA also states that for each configuration $k_i \in K$ there is also a degradation associated $k_i^{deg} \in [0, 1]$.

It is important to remember that the SLA management concerning the penalties, that is, how those degradations are linked to penalties, is out of the scope of this thesis work. Instead, it is used as indicators for goals and objectives in the analysis task (cf. Section 5.2.3).

Finally, variable $budget$ represents the application expenditure rate, which corresponds to the maximum amount of money the application is intended to spend in renting fees at each analysis execution.

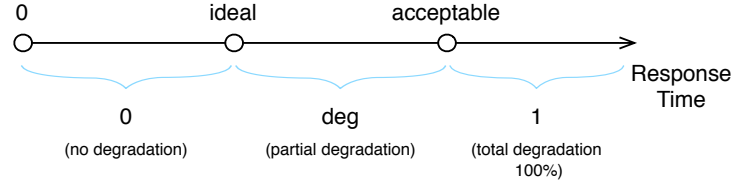


Figure 5.7: Performance Degradation Levels.

Runtime Context. The application runtime context is defined as a tuple $\mathcal{C} = (k, \lambda_L, \lambda_U, Y, V, CPR)$, where:

- $k \in K$ is the current architectural configuration.
- $\lambda_L, \lambda_U \in \Lambda$ is the current lower and upper workload thresholds, respectively.
- $V = \{vm_i \mid i \in [1, v]\}$ is the set of VMs allocated to the application.
- $Y_{n \times v}$ is an incidence matrix where

$$Y_{ij} = \begin{cases} 1 & \text{if } c_i \text{ is deployed on } vm_j \\ 0 & \text{otherwise} \end{cases} \quad (5.17)$$

- $CPR = \{cpr_i \mid i \in [1, t]\}$ is the set of promotions currently used by the application, where each element cpr_i is a tuple $(cpr_i^{price}, cpr_i^{pack})$ composed of a discount price $cpr_i^{price} \in \mathbb{R}^+$ and a set of VMs $cpr_i^{pack} \in V$.

Events and Actions

This section describes the AM events and actions definition based on the hierarchy presented in Figure 5.2. In addition, we present also which actions can be performed followed by the detection of each event.

Workload Increased/Decreased. These are endogenous events which happen whenever the current average workload λ_{avg} (the average workload within the last t_w time units) reaches either the current workload lower threshold λ_L (*Workload Decreased*) or workload upper threshold λ_U (*Workload Increased*), as it is formalized in Equation 5.18. The event contains the new workload upper threshold (Equation 5.19) and the new workload lower threshold (Equation 5.20).

$$\lambda_{avg} < \lambda_L \vee \lambda_{avg} \geq \lambda_U \text{ holds during } t_w \text{ seconds} \quad (5.18)$$

$$\lambda_U^{next} = \begin{cases} \{\min \lambda \in \Lambda \mid \lambda > \lambda_U\} & \text{if } \lambda_{avg} > \lambda_U \\ \lambda_L & \text{if } \lambda_{avg} < \lambda_L \end{cases} \quad (5.19)$$

$$\lambda_L^{next} = \begin{cases} \lambda_U & \text{if } \lambda_{avg} > \lambda_U \\ \{\max \lambda \in \Lambda \mid \lambda < \lambda_U\} & \text{if } \lambda_{avg} < \lambda_L \end{cases} \quad (5.20)$$

As shown in Figure 5.8 (a), before triggering the analysis task, the AM requests the tokens of the available promotions. The analysis task determines which is the best architectural configuration and the minimal amount of resources (VMs) for the new workload thresholds and releases the tokens of promotions that are not useful for the application in question. It means that if there are more promotions available than what the application actually needs, the AM releases the tokens of the useless ones, if there are any. The planning task generates a different plan according to the result produced by the analysis, i.e. whether there is a need for more VMs or not. At the end, planning task produces a *Request VMs interloop action* along with a TOKEN TRANSFER

(if that is the case), followed by a *Schedule Handler* action. The handler, in this case, contains all the actions on the managed system that will be executed when the requested VMs are available (Figure 5.8 (d)). Alternatively, the planning task generates a set of *Stop/Start Component*, *Bind Component*, *Deploy/Remove Component* actions on the application (managed system), followed or not by a *Release VMs interloop action*.

Renting Fees Changed. It is a *Public Knowledge Changed* event that happens whenever the IM creates a new promotion. This kind of event triggers the analysis task that behaves similarly to when a *Workload Increased/Decreased* event is detected. The only difference is that it only requests VMs and never releases, as it can be seen in Figure 5.8 (b).

Scale Down. It is an *interloop event* (exogenous) whose objective is to notify the AM (from the IM) that it should meet some resource constraints. Concretely, the event contains the list of VMs \check{V} that should be immediately released by the application. This event triggers an analysis task to reallocate the components on a smaller number of VMs. To this end, it might be necessary to change the application architectural configuration (e.g. to replace components that are more resource consuming). As a result, a set of *Stop Component*, *Remove Component*, *Deploy Component*, *Bind Component* and *Start Component* actions on the application, followed by a *Release VMs interloop action* are executed (Figure 5.8 (c)).

Promotion Expired. It is an endogenous event that happens whenever an existing promotion becomes expired. This kind of event triggers the analysis task that behaves similarly to when a *Workload Increased/Decreased* event is detected, except for the fact that it may only release VMs (for those that are no longer in promotion) and never request, as it is shown in Figure 5.8 (d).

VMs Created. It is an *interloop event* whose objective is to notify the AM (from the IM) that the Requested VMs are ready for use. It directly triggers the execution task that does nothing but executes the handler by passing as argument the future object (list of requested VMs) needed to execute the handler actions (Figure 5.8 (e)). The executor deploys the components on the created VMs, binds those components to others (existing or new ones) and starts the just-deployed components. In addition, in the case of new promotions, the handler also contains a *Fire Event* action so as to trigger a *Promotion Expired* event after the promotion has been expired.

It should be noticed that some issues concerning the synchronization mechanisms may arise between the acquisition and release of the promotion tokens. For example, let us suppose that there are two promotions available and that applications *A* and *B* managed to get only one promotion each. If both applications wait until all the promotions get available the system gets deadlocked. This is an instance of the classical problem of the dining philosophers [Hoa78]. A workaround for this problem would be to consider all available promotions as a big critical section. Hence, in order to have access to promotions (all of them), applications should wait a token acquisition, but this could increase the waiting time for token acquisition (cf. Section 7.2.1). In a different way, in order to avoid waiting a long time to acquire a token, the AM gets only those which are currently available (i.e. not given to other AMs). Moreover, AMs could set a timeout, after which it gives up of trying to get the promotion. All of these solutions are interesting to avoid deadlocks, but they do not guarantee fairness and thus may cause starvation. Another workaround could be the implementation of a controller whose role would be to distribute tokens according to a given scheduling policy. In this thesis work, for simplicity reasons, we take into consideration only the promotions that are currently available. The starvation issue is left as future work (cf. Section 8).

Analysis

Upon the detection of an event at the monitoring task, the AM might trigger the analysis task or directly the execution task, in case of handler execution.

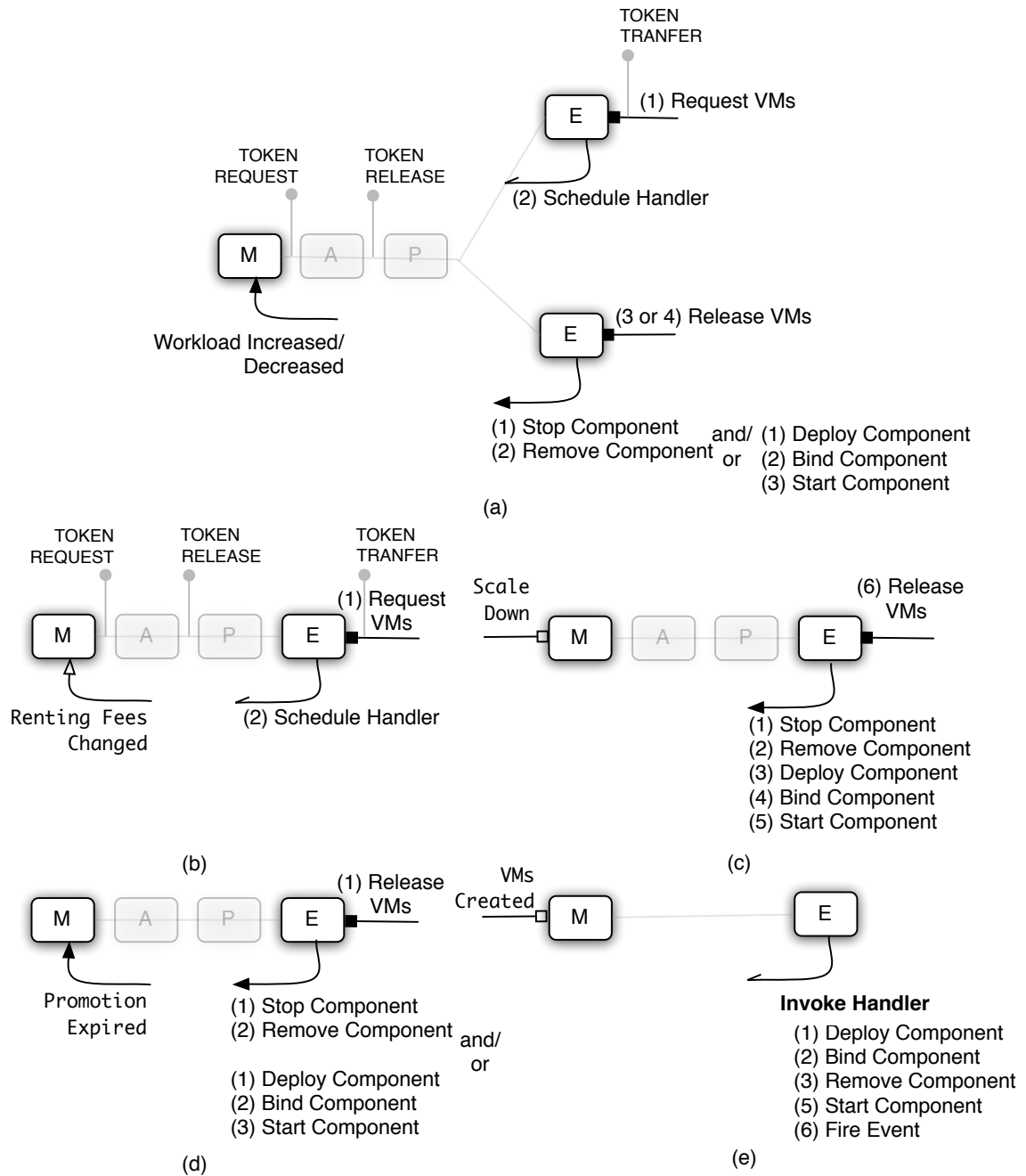


Figure 5.8: Application Manager Events and Actions.

The AM analysis task consists of two different analyzers, which solves separately two problems according to the kind of event detected. Similarly to the IM analyzers, the AM analyzers model these problems as CSOPs and rely on CP to solve them.

Architecture and Capacity Analyzer. This analyzer is triggered upon the detection of a *Workload Increased/Decreased*, *Renting Fees Changed* or *Promotion Expired* event. Its objectives are to determine:

- the best possible architectural configuration (i.e. the components and bindings) so as to minimize the architectural configuration degradation;
- the amount of resources (i.e. number of VMs of each class) necessary so as to minimize the performance degradation and costs due to resource renting fees.

The first part of the CP model for this analyzer is a set of decision variables, which are defined along with the respective domains as follows:

- x , where $D(x) \in [1, m]$: variable indicating the architectural configuration.
- y_{ij} , where $D(y_{ij}) \in [1, \frac{budget}{m_j^{cost}}]$. $\forall i \in [1, n]$ and $\forall j \in [1, q]$: variable indicating the number of VMs of class m_j allocated to component c_i
- u_i , where $D(u_i) \in \{0, 1\}$. $\forall i \in [1, n]$: variable indicating whether the component is used by the architectural configuration x .
- p_i , where $D(p_i) \in \{0, 1\}$. $\forall i \in [1, \xi]$: variable indicating whether the promotion pr_i is used.

The second part of this CP model consists of a set of constraints. Equation 5.22 states that the application total cost due to resource rental cost cannot exceed its *budget*. Equation 5.21 defines an expression variable that stores the discount obtained from promotions. Similarly, Equation 5.23 states a constraint that defines a cost expression variable by taking into consideration the discounts (Equation 5.21) and the decision variable y_{ij} . The cost is normalized ($cost \in [0, 1]$) in function of the application budget.

$$disc = \sum_{i=1}^{\xi} \left(\sum_{j=1}^q (m_j^{cost} * pack_{ij}) - pr_i^{price} \right) * p_i + \sum_{i=1}^t \left(\sum_{\forall vm \in cpr_i^{pack}} vm^{cost} \right) - cpr_i^{price} \quad (5.21)$$

$$budget \geq \sum_{i=1}^n \sum_{j=1}^q y_{ij} * m_j^{cost} - disc \quad (5.22)$$

$$cost = \frac{\sum_{i=1}^n \sum_{j=1}^q y_{ij} * m_j^{cost} - disc}{budget} \quad (5.23)$$

We assume that the AM cannot request and release VMs at the same time. Thus, the amount of VMs for all classes can either increase or decrease, but never increase for one class and decrease for another. In this regard, Equation 5.24 states that if there is at least one VM class, in which the number of allocated VMs increases, it is implied that the number of allocated VMs stays the same or increases, for all the VM classes. Conversely, Equation 5.25 states that if there is at least one VM class, in which the number of allocated VMs decreases, it is implied that the number of allocated VMs stays the same or decreases, for all the VM classes. It also implies that the number of allocated VMs should be at least the number of VMs currently in promotion, for all the VM classes. Finally, it implies also that no new promotions should be requested.

$$\exists j \in [1, q]. \sum_{i=1}^n y_{ij} > \sum_{i=1}^v e_i \implies \forall j \in [1, q]. \sum_{i=1}^n y_{ij} \geq \sum_{i=1}^v e_i \quad (5.24)$$

$$\begin{aligned} \exists j \in [1, q]. \sum_{i=1}^n y_{ij} < \sum_{i=1}^v e_i &\implies \\ \forall j \in [1, q]. \sum_{i=1}^n y_{ij} &\leq \sum_{i=1}^v e_i \wedge \\ \forall j \in [1, q]. \sum_{i=1}^n y_{ij} &\geq \sum_{i=1}^v ep_i \wedge \sum_{i=1}^{\xi} p_i = 0 \end{aligned} \quad (5.25)$$

$$\text{where } e_i = \begin{cases} 1 & \text{if } vm_i^{cpu} = m_j^{cpu} \wedge vm_i^{ram} = m_j^{ram} \\ 0 & \text{otherwise} \end{cases}$$

$$ep_i = \begin{cases} 1 & \text{if } vm_i^{cpu} = m_j^{cpu} \wedge vm_i^{ram} = m_j^{ram} \wedge vm_i \in \bigcup_{l=1}^t cpr_l^{pack} \\ 0 & \text{otherwise} \end{cases}$$

Equation 5.26 defines a constraint indicating that if a configuration is chosen, the corresponding components should be “checked” as used (variable u_i), and the expression variable deg^{arch}

must be equal to the corresponding architectural degradation. With respect to the constraint expressed by Equation 5.27, it indicates that if a component is not used by the configuration, there is no VMs allocated to it.

$$x = l \implies (u_i = 1 \iff c_i \in k_l^c) \wedge deg^{arch} = k_l^{deg} \text{ where } l \in [1, m] \quad (5.26)$$

$$u_i = 0 \implies \sum_{j=1}^q y_{ij} = 0 : \forall i \in [1, n] \quad (5.27)$$

Equation 5.28 expresses a constraint stating the impact of chosen promotions on the allocated resources (y_{ij}). For each new promotion p_l , if it is taken by the application, the number of VMs of each class m_j should be greater or equal to the current number of VMs of that class allocated to the application (e_j) plus the number of VMs ($pack_{lj}$) of the same class offered by the promotion p_l .

$$\forall j \in [1, q] (\sum_{i=1}^n y_{ij} \geq e_j + \sum_{l=1}^{\xi} p_l * pack_{lj}) \text{ where } pack_{lj} \in pr_l^{pack} : \forall l \in [1, \xi] \quad (5.28)$$

The constraints expressed by Equations 5.29 and 5.30 defines respectively the expressions for CPU and RAM requirements for component $c_i, \forall i \in [1, n]$.

$$cpu_i = \sum_{j=1}^q y_{ij} * m_j^{cpu} : \forall i \in [1, n] \quad (5.29)$$

$$ram_i = \sum_{j=1}^q y_{ij} * m_j^{ram} : \forall i \in [1, n] \quad (5.30)$$

The constraint expressed by Equation 5.31 defines the performance degradation expression in function of previously defined SLA conditions. If the response time is above the accepted threshold, the degradation is equal to 1 (i.e. 100% of degradation). If it is between the acceptable and the ideal threshold, it is equal to rt^{deg} . Otherwise, it is equal to zero, i.e. there is no degradation if the response time is inferior or equal to the ideal threshold.

$$deg^{perf} = \begin{cases} 1 & \text{if } \sum_{i=0}^n u_i * perf_i(\lambda_U^{next}, cpu_i, ram_i) > rt^{acc} \\ rt^{deg} & \text{if } rt^{ideal} < \sum_{i=0}^n u_i * perf_i(\lambda_U, cpu_i, ram_i) \leq rt^{acc} \\ 0 & \text{otherwise} \end{cases} \quad (5.31)$$

The last part of the CP model is the objective function. It is important to recall that the objective of this analyzer and consequently the CP model is to find the best architectural configuration while minimizing costs due to resource renting fees. Hence, Equation 5.32 defines an objective function that minimizes the performance and architectural degradations as well as the cost due to resource renting fees.

$$minimize(\alpha_{perf} * deg^{perf} + \alpha_{arch} * deg^{arch} + \alpha_{cost} * cost) \quad (5.32)$$

Where $\alpha_{arch} + \alpha_{perf} + \alpha_{cost} = 1$ and $\alpha_{arch}, \alpha_{perf}, \alpha_{cost} \in [0, 1]$. $\alpha_{arch}, \alpha_{perf}$ and α_{cost} correspond to weights for the architectural degradation, performance degradation and cost, respectively.

Scale Down Analyzer. This analyzer is very similar to the *Architecture and Capacity Analyzer*, since the objectives are the same. The only difference is that the available resources are more constrained in the sense that some VMs may be suppressed. Let $\bar{V} = V - \check{V}$, that is, the difference between the vector of running VMs and the vectors of VMs to be released. Therefore, the constraint expressed in Equation 5.33 must hold.

$$\forall j \in [1, q]. \sum_{i=1}^n y_{ij} \leq \sum_{i=1}^{|\check{V}|} e_i, \text{ where } e_i = \begin{cases} 1 & \text{if } vm_i^{cpu} = m_j^{cpu} \wedge vm_i^{ram} = m_j^{ram} \\ 0 & \text{otherwise} \end{cases} \quad (5.33)$$

Planning

The planning task is triggered after the execution of the analysis task. It takes into consideration the managed application's current state (e.g. architectural configuration and mapping components/VMs) and the desired state (e.g. a new architectural configuration and/or a new mapping components/VMs) resulting from the analysis task, to create the AM autonomic actions (cf. Section 5.2.3) in a meaningful way.

According to Figure 5.8, there are four possible planners, each one is supposed to deal with a different event. However, as they have the same objectives and the same inputs and outputs, they can be implemented as a single planner (hereby called *Architecture and Capacity Planner*). In order to ease the understanding, we divide the description into two parts: (i) one describing the plan when there is a need for new VMs; and (ii) when there is no need for new VMs. The difference between the two cases is that in the first one it is necessary to encapsulate a set of actions in a handler that is executed upon the arrival of a future object, whereas in the second case the entire plan is executed without waiting for anything.

Algorithm 6 describes the planning steps when it is necessary to request VMs. The algorithm starts by creating the plan object (line 1). Then it initializes the variables that are used to store the VMs and promotions to be requested (lines 2 and 3). Then, it adds to the set of VMs to be requested the number of required VMs for each VM class m_j (lines 4-5). Similarly, it adds the desired promotions to the set of promotions to be requested (6-8). Finally, it adds the *RequestVMs* action (line 9) and a *ScheduleHandler* action to the plan (line 10). The action *ScheduleHandler* takes as argument the handler to be registered and executed in the future. The procedure *Handle* defines the handler that deals with the arrival of the requested VMs. It starts by defining the set of new VMs V' and the new mapping components/VMs (lines 2-4). Then, it creates the objects for the plan and the fork activities that are responsible for the execution in parallel of *StopComponent*, *StartComponent*, *RemoveComponent*, *DeployComponent*, and *BindComponents* actions (lines 5-10). These objects are then added to the plan (lines 12-16). A *Fire Event* is created to trigger a *Promotion Expired* after the promotion duration *promoDuration* have reached its term. Then it is added to the plan (line 17). Finally the plan is then executed (line 18).

It should be noticed that all these fork activities are executed in sequence, that is, all the actions added to *stopFork* executes in parallel, but the actions added to *removeFork* starts the execution only when all the actions of *stopFork* have terminated.

The code omitted in line 11 of the procedure *Handle* is split into six parts:

1. **Stopping Components:** the components that depend on the components that will be removed are stopped;
2. **Removing Components:** the component instances that are no longer useful are removed from their respective VMs;
3. **Deploying Components:** deployment of new component instances;
4. **Binding Components:** existing components' references are bound to new components' services, and new components' references are bound to existing components;

Algorithm 6: Architecture and Capacity Planner (when it is necessary to request new VMs).

Input: C : the current runtime context.
Input: k_x : the new architectural configuration.
Input: λ_U^{next} : the new upper workload threshold.
Input: λ_L^{next} : the new lower workload threshold.
Input: y_{ij} : the mapping of components to number of VMs for each class.
Input: p_i : the promotions to be requested.
Input: M : the VM rental fees.
Input: $promoDuration$: duration of promotions.
Result: $Plan$: the resulting plan

- 1 $plan \leftarrow$ new instance of Plan
- 2 $R^{vms} \leftarrow \emptyset$
- 3 $R^{pr} \leftarrow \emptyset$
- 4 **for** $j \leftarrow 1$ **to** q **do**
- 5 $R^{vms} \leftarrow R^{vms} \cup \{(\sum_{i=1}^n y_{ij}) - \{vm_i \mid i \in [1, v], vm_i^{cpu} = m_j^{cpu} \wedge vm_i^{ram} = m_j^{ram}\}\}$
- 6 **for** $i \leftarrow 1$ **to** a **do**
- 7 **if** $p_i = 1$ **then**
- 8 $R^{pr} \leftarrow R^{pr} \cup pr_i$
- 9 $add(RequestVMs(R^{vms}, R^{pr}), plan)$
- 10 $add(ScheduleHandler(handle), plan)$
 - 1: **procedure** HANDLE(vms)
 - 2: $v' \leftarrow v + |vms|$
 - 3: $V' \leftarrow V \cup vms$
 - 4: $Y'_{ij} \leftarrow Y_{ij} : \forall i \in [1, p], \forall j \in [1, v']$
 - 5: $plan \leftarrow$ new instance of Plan
 - 6: $stopFork \leftarrow$ new instance of Fork
 - 7: $removeFork \leftarrow$ new instance of Fork
 - 8: $deployFork \leftarrow$ new instance of Fork
 - 9: $bindFork \leftarrow$ new instance of Fork
 - 10: $startFork \leftarrow$ new instance of Fork
 - 11: ...
 - 12: $add(stopFork, plan)$
 - 13: $add(removeFork, plan)$
 - 14: $add(deployFork, plan)$
 - 15: $add(bindFork, plan)$
 - 16: $add(startFork, plan)$
 - 17: $add(FireEvent(PromotionExpired(R^{pr}), promoDuration)$
 - 18: $execute(plan)$
 - 19: **end procedure**

5. **Starting Components:** the components stopped in item 1 are re-started.

The first part of the handler execution is pseudo-coded in Algorithm 7. The objective is to create autonomic actions to stop all component instances that depend on the components to be removed. So, the algorithm iterates over the set of components by filtering the components that is in the current architectural configuration k but not in the new one k_x (lines 1-10), that is, components that should be removed. For each component c_i , the algorithm iterates over the set of offered services c_i^s (lines 3-10) by searching the references which each service s is bound to (lines 4-10). Then, the algorithm searches the components (lines 5-10) containing each reference r (lines 6-10). Then, it iterates over the set of VMs (lines 7-10) by picking up those hosting instances of component c_l (8-10), that is, the components that depend on the components to be removed. Finally, it

creates a *StopComponent* action which is added to the fork activity *stopFork*.

Algorithm 7: Stopping Components in the Handler Execution.

```

// iterates over the set of components
1 for  $i \leftarrow 1$  to  $n$  do
    // checks if component  $c_i$  belongs to the current
    // configuration  $k$  but not to the new one  $k_x$ 
2 if  $c_i \notin k_x^c \wedge c_i \in k^c$  then
    // iterates over the set of services of component
    // offered by component  $c_i$ 
3 foreach  $s \in c_i^s$  do
    // iterates over the set of references which the
    // service  $s$  is bound to
4 foreach  $r \in \{r \mid (r, s) \in k^{bind}\}$  do
    // iterates again over the set of components
5 for  $l \leftarrow 1$  to  $n$  do
    // filters the components having the reference  $r$ 
    // and belonging to the new architectural
    // configuration  $k_x$ 
6 if  $r \in c_l^r \wedge c_l \in k_x^c$  then
    // iterates over the set of VMs
7 for  $j \leftarrow 1$  to  $v$  do
    // Picks up the VMs hosting an instance of
    // component  $c_l$ 
8 if  $Y'_{lj} = 1$  then
9      $add(StopComponent(c_l, vm_j),$ 
10     $stopFork)$ 

```

The second part of the handler execution aims to create autonomic actions to remove the components that are no longer useful from their respective VMs. As it is detailed in Algorithm 8, it iterates over the set of components and VM classes (lines 1-9). It calculates the current number of VMs of class m_j allocated to the concerned component c_i (*nbVMs*) (line 3). Based on the difference between current and future number of VMs of class m_j allocated to the concerned component c_i ($nbVMs - y_{ij}$), the algorithm iterates over the set of VMs (V , lines 5-8) by searching VMs of a given class m_j that host the concerned component c_i (line 6). Once a VM is found, the algorithm creates a *RemoveComponent* action (line 8) to remove the instance of component c_i from the VM.

The third part of the handler execution is detailed in Algorithm 9. It aims at creating autonomic actions to deploy components in VMs, according to the analysis task result. It also iterates over the sets of components and VM classes, while searching for components whose number of VMs of a given class m_j allocated to it should be increased (lines 2-9). Hence, for each new VM to be assigned to a component c_i in question ($y_{ij} - nbVMs$, lines 5-9), the algorithm iterates over the new set of VMs (V' , lines 5-8) by searching VMs of a given class m_j which are not assigned to any component (line 7). Once a VM is found, the algorithm creates the *DeployComponent* action and add it to the fork activity *deployFork* (line 9).

The fourth part of the handler execution (Algorithm 10) aims at creating the autonomic actions for binding the previously deployed components. From lines 1 to 12 the algorithm creates *BindComponents* actions from existing components to the new ones. It iterates over the set of services c_i^s of each component c_i that belongs to the new configuration k_x but not to the current one k (lines 1-10), then over the set of references which the service s is bound to (4-10). It iden-

Algorithm 8: Removing Components in the Handler Execution.

```

// iterates over the set of components
1 for  $i \leftarrow 1$  to  $n$  do
    // iterates over the set of VM classes
2 for  $j \leftarrow 1$  to  $q$  do
    // calculates the number of VMs of class  $m_j$  currently
    // allocated to component  $c_i$ 
3  $nbVMs \leftarrow \sum_{k=1}^v e_k$ , where
    
$$e_k = \begin{cases} 1 & \text{if } Y_{ik} = 1 \wedge vm_k^{cpu} = m_j^{cpu} \wedge vm_k^{ram} = m_j^{ram} \\ 0 & \text{otherwise} \end{cases}$$

    // iterates as many times as the difference between
    //  $nbVMs$  and future number of VMs of class  $m_j$  ( $y_{ij}$ )
    // allocated to  $c_i$ 
4 for  $k \leftarrow 1$  to  $(nbVMs - y_{ij})$  do
    // iterates over the set of VMs
5 for  $l \leftarrow 1$  to  $v$  do
    // Picks up the first VM of class  $m_j$  allocated to
    //  $c_i$ 
6 if  $Y_{il} = 1 \wedge vm_l^{cpu} = m_j^{cpu} \wedge vm_l^{ram} = m_j^{ram}$  then
7      $Y'_{il} \leftarrow 0$ 
8      $add(RemoveComponent(c_i, vm_l), removeFork)$ 

```

Algorithm 9: Deploying Components in the Handler Execution.

```

1  $newInstances \leftarrow \emptyset$ 
// iterates over the set of components
2 for  $i \leftarrow 1$  to  $n$  do
    // iterates over the set of VM classes
3 for  $j \leftarrow 1$  to  $q$  do
    // calculates the number of VMs of class  $m_j$  currently
    // allocated to component  $c_i$ 
4  $nbVMs \leftarrow \sum_{k=1}^v e_k$ , where
    
$$e_k = \begin{cases} 1 & \text{if } Y_{ik} = 1 \wedge vm_k^{cpu} = m_j^{cpu} \wedge vm_k^{ram} = m_j^{ram} \\ 0 & \text{otherwise} \end{cases}$$

    // iterates as many times as the difference between
    // future number of VMs of class  $m_j$  ( $y_{ij}$ ) allocated to  $c_i$ 
    // and  $nbVMs$ 
5 for  $k \leftarrow 1$  to  $(y_{ij} - nbVMs)$  do
    // iterates over the set of VMs
6 for  $l \leftarrow 1$  to  $v'$  do
    // Picks up the first VM of class  $m_j$  which is not
    // used
7 if  $Y'_{gl} = 0, \forall g \in [1, n] \wedge vm_l^{cpu} = m_j^{cpu} \wedge vm_l^{ram} = m_j^{ram}$  then
8      $Y'_{il} \leftarrow 1$ 
9      $add(DeployComponent(c_i, vm_l), deployFork)$ 
10     $newInstances \leftarrow newInstances \cup \{vm_l\}$ 

```

tifies which components the reference r belongs to (lines 5-10) as well the VMs in which they are deployed (line 7-10). Finally, it creates the *BindComponents* actions and adds it to the fork activity *bindFork* (line 9).

From lines 11-18, the algorithm creates *BindComponents* actions from the new component instances to other components on which they depend. It iterates over the set of components (lines 11-18) belonging to the new architectural configuration k_x , then over the set of new VMs (lines 13-18). The algorithm iterates over the set of VMs by filtering the VMs which are also presented in set *newInstances* (lines 14-18) (created in Algorithm 9). Then, for each component c_i , it iterates over the set of references (lines 15-18) while searching which services are bound to each references r (lines 16-18). Finally, for each reference, the algorithm creates *BindComponents* actions and adds them to the fork activity *bindFork*.

Algorithm 11 aims to create autonomic actions to start the components that were stopped in the beginning of the handler execution (Algorithm 7). That way, the algorithm proceeds identically to Algorithm 7, except line 9, in which the *StopComponent* action creation is replaced by a *StartComponent* action.

Algorithm 12 describes the planning steps when it is not needful to request VMs. The content of line 9 is identical to the content of the handler execution. The algorithm starts by instantiating the objects that are responsible for executing actions in parallel (Fork) or in sequence (Sequence) (lines 1-6). It then starts a variable which stores the VMs to be released (line 7) and defines a variable which corresponds to the temporary mapping components/VMs (line 8). This variable is used similarly to Algorithm 6, i.e. to reestablish a new mapping components/VMs. The difference in this case is that there might be VMs in V that are no longer useful. So, the algorithm iterates over the VMs in order to identify the VMs to be released (lines 10-12). Finally, the algorithm arranges the set of actions in a sequence way, that is, first a set of *StopComponent* actions are executed in parallel (line 13), then also in parallel a set of *RemoveComponent* actions are executed (line 14), followed by the parallel execution of a set of *DeployComponent* actions (line 15). This followed by a set of *BindComponents* actions, which are executed in parallel (line 16). *StartComponent* actions are then executed in parallel (line 17), followed by a *ReleaseVMs* action (line 18).

5.3 Summary

This chapter presented our approach for coordination and synchronization of multiple autonomic managers with the purpose to improve the Quality of Service (QoS) at the application level and the energy efficiency at the infrastructure level.

Firstly, we presented an autonomic model for multiple autonomic managers, which basically consists of a coordination and a synchronization protocols. The coordination protocol is based on a message oriented middleware and promotes asynchronous one-to-one or one-to-many *interloop* communication. The interaction between two autonomic managers is performed by means of actions and events. To this end, we define a general hierarchy of autonomic events and actions that was then extended with cloud-specific actions and events. The synchronization protocol is based on a token protocol and it is used to guarantee exclusive access to critical sections in the public knowledge.

Secondly, we applied the autonomic model to the context of cloud computing. More precisely we defined two kinds of autonomic managers: the infrastructure and the application manager. The infrastructure manager aims at responding the application manager requests by placing the new incoming virtual machines so as to optimize the infrastructure utilization rate, and consequently the energy efficiency. The application manager aims to improve the managed application QoS while reducing costs due to resource renting fees. For each autonomic manager, we provide details on their knowledge model, events and actions. The analysis tasks of both managers are identified as Constraint Satisfaction and Optimization Problems and Constraint Programming [RVBW06, RvBW07] is used to model and solve those problems.

Algorithm 10: Binding Components in the Handler Execution.

```

// iterates over the set of components
1 for  $i \leftarrow 1$  to  $n$  do
    // check if component  $c_i$  belongs to the new configuration
    // but not the current one
2    if  $c_i \in k_x^c \wedge c_i \notin k^c$  then
        // iterates of the set of services offered by  $c_i$ 
3        foreach  $s \in c_i^s$  do
            // iterates over the set references to whom service  $s$ 
            // is bound
4            foreach  $r \in \{r \mid (r, s) \in k_x^{bind}\}$  do
                // iterates again over the set of components
5                for  $l \leftarrow 1$  to  $n$  do
                    // checks if the reference  $r$  belongs to
                    // component  $c_l$ 
6                    if  $r \in c_l^r$  then
                        // iterates over the set of VMs
7                        for  $j \leftarrow 1$  to  $v'$  do
                            // checks if  $vm_j$  hosts an instance of
                            // component  $c_l$ 
8                            if  $Y'_{lj} = 1$  then
9                                 $add(BindComponents(c_l, vm_j, r1, s1),$ 
10                                $bindFork)$ 
// iterates over the set of components
11 for  $i \leftarrow 1$  to  $n$  do
    // checks if component  $c_i$  belongs to the new configuration
12    if  $c_i \in k_x^c$  then
        // iterates over the set of VMs
13        for  $j \leftarrow 1$  to  $v'$  do
            // checks if  $vm_j$  hosts an instance of component  $c_i$  and
            // if it is a new instance
14            if  $Y'_{ij} = 1 \wedge vm_j \in newInstances$  then
                // iterates over the set of references of
                // component  $c_i$ 
15                foreach  $r \in c_i^r$  do
                    // iterates over the set of services which
                    // reference  $r$  is bound to
16                foreach  $s \in \{s \mid (r, s) \in k_x^{bind}\}$  do
17                     $add(BindComponents(c_i, vm_j, r, s),$ 
18                     $bindFork)$ 

```

Finally, in order to promote a synergy between applications and infrastructure managers, we define a public knowledge at the infrastructure level. It consists of a variable renting fees, in which the objective is to improve the infrastructure utilization rate. In other words, the infrastructure manager can decrease the renting fees of certain virtual machines by creating promotions. That way, application managers may take these promotions into account to perform their analysis and benefit of them. As a result, this may impact positively on the infrastructure, since the promotions

Algorithm 11: Starting Components in the Handler Execution.

```

1 for  $i \leftarrow 1$  to  $n$  do
2   if  $c_i \notin k_x^c \wedge c_i \in k^c$  then
3     foreach  $s \in c_i^s$  do
4       foreach  $r \in \{r \mid (r, s) \in k^{bind}\}$  do
5         for  $l \leftarrow 1$  to  $n$  do
6           if  $r \in c_l^r \wedge c_l \in k_x^c$  then
7             for  $j \leftarrow 1$  to  $v$  do
8               if  $Y'_{ij} = 1$  then
9                 add( $StartComponent(c_l, vm_j)$ ),
10                startFork)

```

Algorithm 12: Architecture and Capacity Planner (when it is not necessary to request VMs).

Input: C : the current runtime context
Input: k_x : the new architectural configuration.
Input: y_{ij} : the mapping of components to number of VMs for each class.
Input: M : the VM rental fees.
Output: $plan$: the resulting plan.

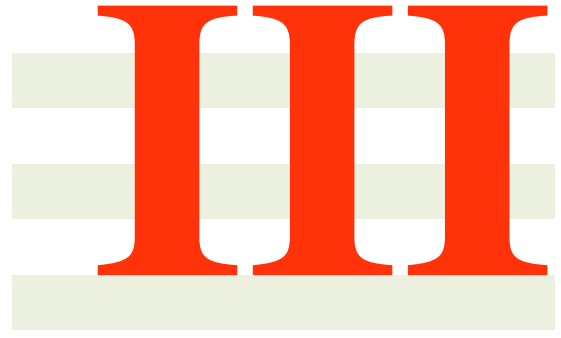
```

1  $plan \leftarrow$  new instance of Plan
2  $stopFork \leftarrow$  new instance of Fork
3  $removeFork \leftarrow$  new instance of Fork
4  $deployFork \leftarrow$  new instance of Fork
5  $bindFork \leftarrow$  new instance of Fork
6  $startFork \leftarrow$  new instance of Fork
7  $R^{vms} \leftarrow \emptyset$ 
8  $Y'_{ij} \leftarrow Y_{ij} : \forall i \in [1, p], \forall j \in [1, v]$ 
9 ...
10 for  $j \leftarrow 1$  to  $v$  do
11   if  $\sum_{i=1}^n Y'_{ij} = 0$  then
12      $R^{vms} \leftarrow R^{vms} \cup \{vm_j\}$ 
13 add( $stopFork, plan$ )
14 add( $removeFork, plan$ )
15 add( $deployFork, plan$ )
16 add( $bindFork, plan$ )
17 add( $startFork, plan$ )
18 add( $ReleaseVMs(R^{vms}), plan$ )

```

are created intentionally to stimulate applications to occupy a given free slot of resource and therefore improve the infrastructure utilization rate.

In addition, the coordination protocol also provides event/action based interfaces (namely *interloop* actions and events) that can be used by autonomic managers to explicitly communicate. They can be used to inform other managers about a situation that may impact them. For example, every time the infrastructure faces an energy shortage situation, its IM uses the coordination protocol to warn the affected AMs that part of the infrastructure will be shutdown within a given time. Hence, AMs have the chance to adapt the concerned applications (e.g. through architectural elasticity) before the shortage takes place. Therefore, this kind of synergy contributes to mitigate the impacts of an energy shortage (at the application level) on the affected applications.



Validation

Research Prototype

This chapter aims at describing the prototype resulting from the concepts and models proposed in Chapter 5. The prototype is composed of two distributed Java sub-systems, namely Application Manager (AM) and Infrastructure Manager (IM). Firstly, we present the common autonomic framework, upon which both sub-systems rely. Secondly, for each sub-system, we provide details on: (i) the managed application's meta-model, (ii) the technical aspects of the managed application; and (iii) the autonomic tasks (monitoring, analysis, planning, execution and knowledge).

Finally, we depict the autonomic managers lifetime by showing how objects within the same and from different autonomic managers interact to each other over the time.

Contents

6.1	Common Autonomic Framework	101
6.2	Application Manager	104
6.2.1	Application Meta-model	105
6.2.2	Managed System Technical Details	105
6.2.3	Autonomic Tasks	106
6.3	Infrastructure Manager	113
6.3.1	Infrastructure Meta-model	113
6.3.2	Managed System Technical Details	114
6.3.3	Autonomic Tasks	116
6.4	Autonomic Managers Lifetime	122
6.5	Summary	127

6.1 Common Autonomic Framework

Based on the autonomic model presented in Chapter 5, we implemented a common autonomic framework which is reused for both types of autonomic managers: AM and IM.

Most of the meta-models presented in this Chapter follows the Unified Modeling Language (UML)¹ standards, which is maintained by the Object Management Group (OMG)². Before starting the description, it is important to precise the semantics of the arrows used in the meta-model diagrams presented in this Chapter, as it is depicted in Figure 6.1.

¹<http://www.uml.org/>

²<http://www.omg.org>

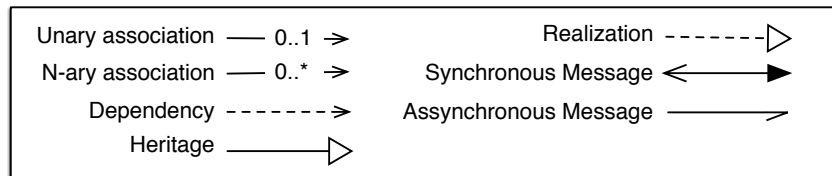


Figure 6.1: Meta-model Arrows Semantics.

As it can be seen in Figure 6.2, an autonomic manager basically consists in the execution of a sequence of autonomic tasks (generalized by the abstract class *AutonomicTask*) [KC03], which are triggered upon the detection of a given event and taking into consideration the current values of the knowledge. The knowledge can be either private (abstract class *PrivateKnowledge*) or public (abstract class *PublicKnowledge*). A knowledge is private if the autonomic manager that maintains the knowledge is the only one that has the rights to access the values stored in it. The public knowledge, on the contrary, can also be accessed from other managers. Besides, the public knowledge also contains a list of critical objects (abstract class *CriticalSection*), which, due to concurrency problems, should be accessed/modified upon the acquisition of a token. Therefore, critical objects aggregates a semaphore to control the distribution of tokens.

Interlocutors (class *Interlocutor*) are other autonomic managers to which the autonomic manager communicates and are maintained by the private knowledge. The public knowledge also maintain the list of interlocutors that are subscribed to it.

Regarding the events, they are implemented by the abstract class *AutonomicEvent* and follow the hierarchy proposed on the model (cf. Section 5.1.1). That is, the class *AutonomicEvent* represents the most generic event and it is specialized by classes *EndogenousEvent* and *ExogenousEvent*, which, in turn, is specialized by classes *InterloopEvent* and *PublicKnowledgeChanged*.

The class *AutonomicManager* is a realization of the interface *AutonomicEventListener* so it may subscribe to listen for the events detected by the monitor (abstract class *Monitor*). The monitor is composed of two objects: one for managing the endogenous events, i.e. the events created based on the monitoring data on the managed system (abstract class *ManagedSystemSensorManager*), and another for managing the exogenous events (abstract class *InterloopSensorManager*), i.e. the events resulting from interloop actions. Basically, classes *InterloopSensorManager* and *ManagedSystemSensorManager* are in charge of monitoring data and managing more fine-grained events from different sources (managed system and other autonomic managers) and generates events that are used by *Monitor* to notify the listeners subscribed to it, including the autonomic manager.

The analysis task is performed by an analyzer (abstract class *Analyzer*). It may take into account the knowledge and some other information contained in the emitted event and produces a result that is stored in an object of class *AnalysisResult*. The planning task is performed by a planner (abstract class *Planner*). It may take also into account some information presented in the knowledge as well as the result from the analysis task, and produces a plan (abstract class *Plan*) as result.

At last, the execution task is performed by an executor (class *Executor*) that takes the plan generated by the planner and executes it. To this end, the executor relies on the Java Executor Framework³, since it provides an efficient way (based on a pool of threads) to address performance issues due to the use of big number of threads in Java applications.

Figure 6.3 shows the implementation of a plan. It follows the composite design pattern [GHJV95], which means that a plan may be composed of a single (*ElementaryTask*) or a composite (*SequenceTask* and *ForkTask*) tasks. A composite task may contain one or several plans, each one implemented as a single or a composite task, and so forth. A composed task can be

³<http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Executor.html>

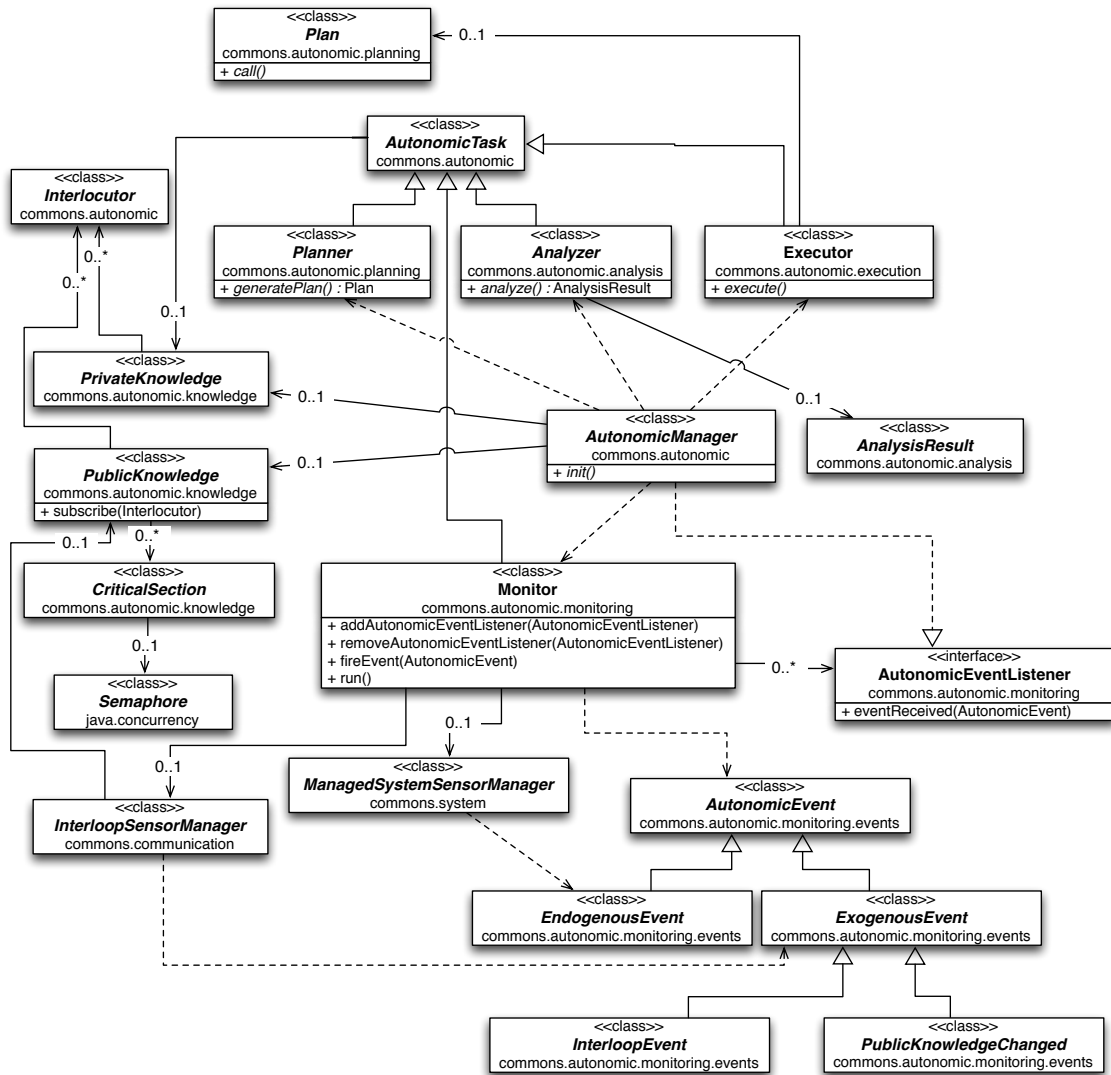


Figure 6.2: Common Autonomic Framework Meta-model.

executed in sequence (*SequenceTask*) or in parallel (*ForkTask*), which waits the termination of all the tasks inside it to terminate its execution.

The resulting plan is a set of autonomic actions (abstract class *AutonomicAction* extended from abstract class *ElementaryTask*), organized in the form of dependence graph, in which a task should not start its execution while all the tasks upon which it depends have not terminated.

Similar to the events hierarchy, the actions hierarchy follows the autonomic model proposed in Section 5.1.1. The class *AutonomicAction* is extended by classes *InterloopAction*, *ActionOnManagedSystem* and *IntraloopAction*. The class *InterloopAction* performs actions on other autonomic managers by delegating this to the abstract class *InterloopActuatorManager*. The class *AutonomicManager* also relies on *InterloopActuatorManager* to access others autonomic managers' public knowledge. The class *ActionOnManagedSystem* performs actions on the managed system and delegates them to the abstract class *ManagedSystemActuatorManager*. Finally, actions performed inside the autonomic manager (intraloop actions) are performed by classes *ChangePublicKnowledge*, *FireEvent* and *ScheduleHandler*, which are both specialization of class *IntraloopAction*. In class *FireEvent*, the attribute *timeout* indicates a time interval after which a given event should be triggered.

It is important to recall that the proposed approach is conceptually designed on a message-oriented fashion. That is, autonomic managers communicate to each other via asynchronous mes-

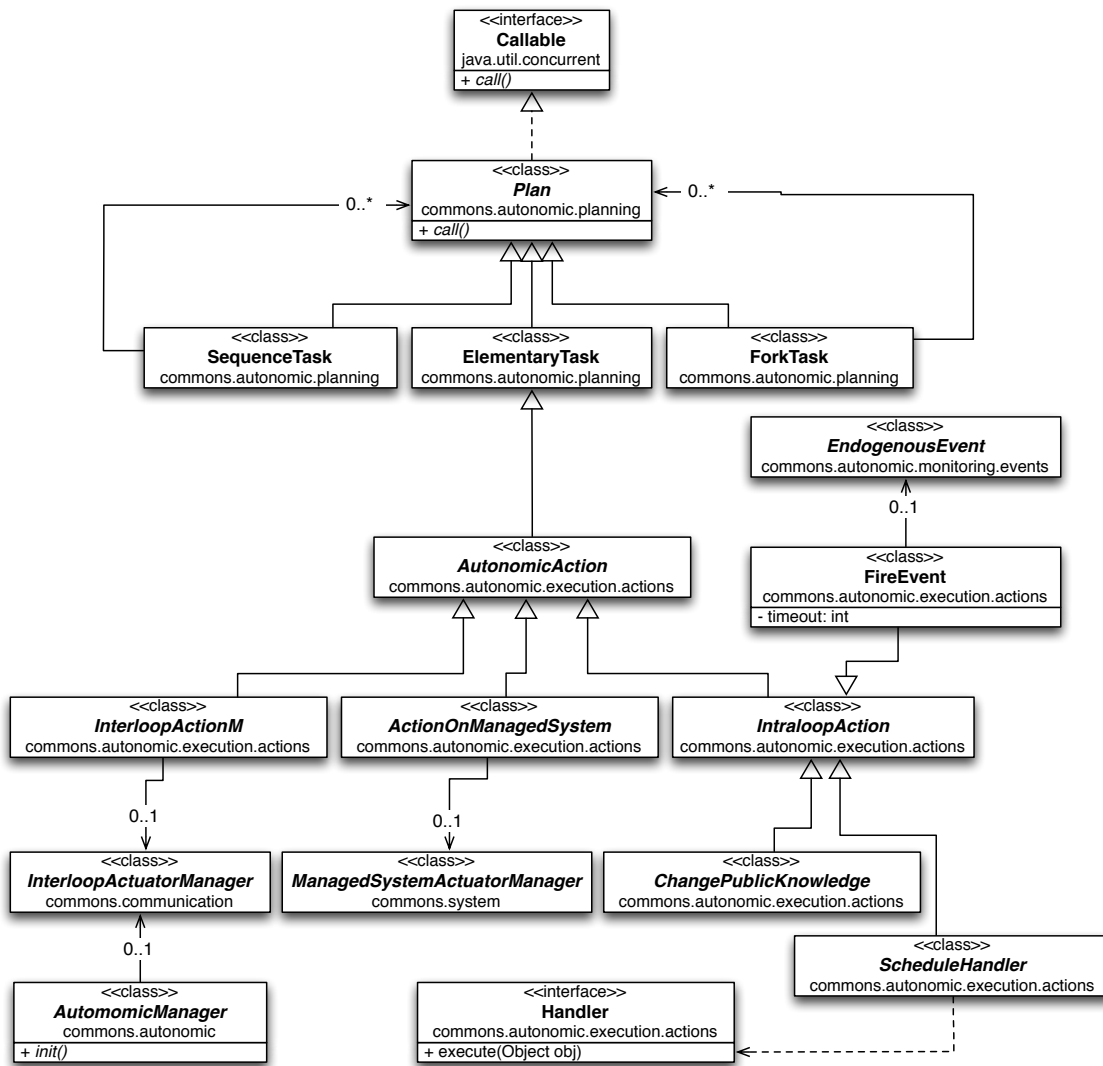


Figure 6.3: The Plan Meta-model of the Common Autonomic Framework Meta-model.

sages via specific queues or broadcasting topics. Generally, the message management task is done by a message broker (e.g. ActiveMQ⁴), in which queues and topics can be created, and which every message should pass through. For sake of simplicity, in this prototype implementation, autonomic managers communicate directly to their corresponding peers via HTTP asynchronous messages without passing through an intermediate broker. Regarding the public knowledge, the access is also performed directly via HTTP synchronous messages. A broker-oriented implementation based on Java Message Service (JMS)⁵ is an ongoing work (cf. Chapter 8).

6.2 Application Manager

This section details the implementation of the AM MAPE-K control loop. First, the application meta-model is presented in order to give a concrete idea about how the managed application is modeled within the AM. Then, we discuss the technical details about the managed system (the application). Finally, the implementation of each autonomic task (monitor, analysis, planning, execution and knowledge) is detailed by means of class diagrams.

⁴<http://activemq.apache.org>

⁵<http://www.oracle.com/technetwork/java/docs-136352.html>

6.2.1 Application Meta-model

Figure 6.4 presents the application meta-model of the prototype implementation. An application (class *Application*) is composed of a set of components (class *Component*). Each component consists of a set of services (class *Service*) and references (class *Reference*). A service is a set of functionalities provided by the component, whereas a reference corresponds to a functional dependency on a service provided by another component. Services and References are connected by bindings (class *Binding*). An application may have several configurations (class *Configuration*), which are defined by a sub-set of components and the way their services and references are bound. Configurations also maintain a list of promoted services, that is, the services that are externally accessed from the outside. According to the architectural configuration, the application may be degraded (attribute *degradation*) in a functional or nonfunctional way. The degradation degree is determined at design time based on the impacts of such an architectural change.

For scalability purposes, a component can be replicated into several component instances (class *ComponentInstance*). That way, requests are dispatched to the different instances through a component frontend (class *ComponentFrontend*), which is a specialized component instance. Each component instance is hosted by one virtual machine (class *VirtualMachine*), which has its computing capacity and price attributes (cpu, ram and price) defined by a virtual machine class (class *VirtualMachineClass*). The renting fees (class *RentingFees*) is a piece of information maintained by the IM's public knowledge. It consists of the set of virtual machine classes offered by the IM. The promotions are implemented by class *Promotion* and represent a group of VMs that are offered together for a lower price. It is also a piece of information kept by the IM public knowledge. However, as they are part of a critical section, it is required to acquire a token every time the AM needs to request a promotion. A promotion is defined by a price (attribute *price*) and an offer, which corresponds to a map of virtual machine classes, and the respective number of virtual machines offered. Finally, each component may have several performance requirements (class *PerformanceRequirement*), defined in terms of cores of CPU and MB of RAM (attributes *cpu* and *ram*), necessary for dealing with a given number of simultaneous requests within a given period of time (attributes *workload* and *responseTime*).

6.2.2 Managed System Technical Details

As previously mentioned, we assume that managed applications are developed in a component-based manner so that it is possible to perform internal reconfiguration by seamlessly replacing, adding or removing components. To this end, we rely on Service Component Architecture (SCA) as reference model to conceive applications. The prototype utilizes FraSCAti [SMF⁺09], an SCA runtime implementation based on the Fractal component model [BCL⁺04].

So, every VM provided by the IM is created containing a FraSCAti runtime started with a remote API, as shown in Figure 6.5. This API allows the remote deployment, introspection, and reconfiguration of SCA components via a RESTful interface. The deployment consists in uploading a contribution archive that contains the SCA component description file with the respective implementation (e.g. a jar file). The introspection and reconfiguration APIs relies on FraSCAti Script, a Domain Specific Language (DSL) for introspection and reconfiguration of SCA components. This DSL was built on top of FScript [DLLC08], a reconfiguration DSL for Fractal components.

Apart from a FraSCAti runtime, the VMs are delivered with pre-installed load balancer called Nginx⁶. This tool was chosen for many reasons such as the possibility to dynamically reconfigure it (e.g. changing its configurations without having to stop it) and low CPU and memory overheads. Nginx provides a web page where the AM can retrieve basic statistics on the load balancing (e.g. number of current accesses, amount of data processed, etc).

Each component instance is deployed in a different VM, and one VM is arbitrarily chosen to play also the role of the component frontend, i.e. the load balancer. The component frontend is

⁶<http://nginx.org/en/>

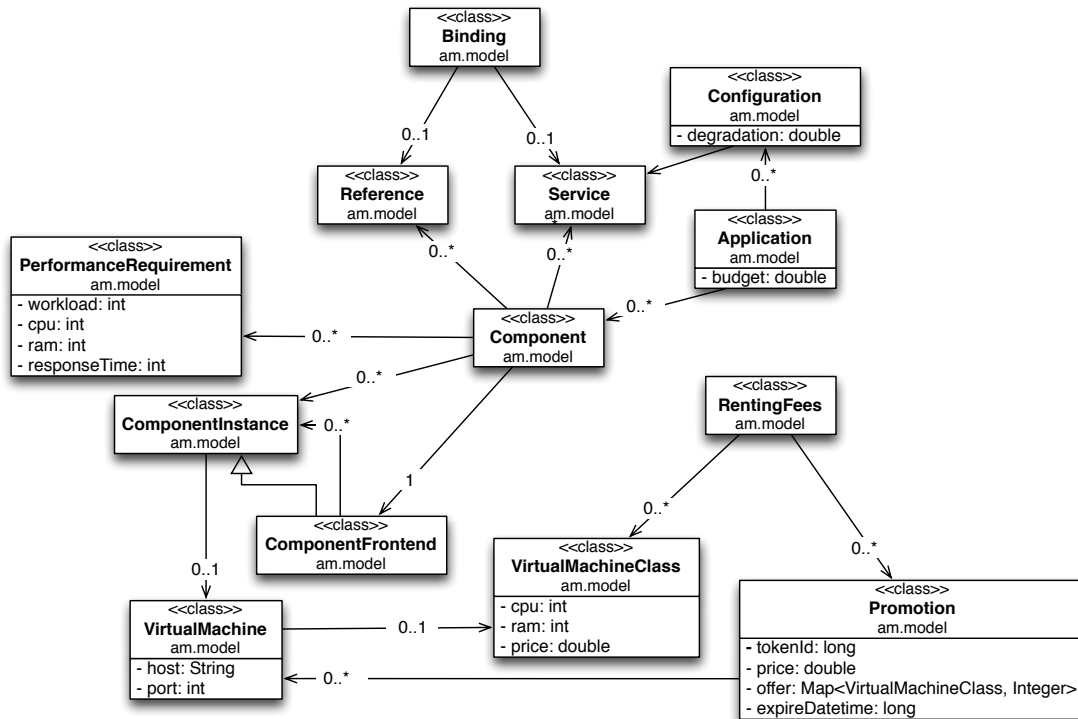


Figure 6.4: Application Meta-model.

wrapped by a FraSCAti component which is used to control the frontend (add/remove instances) through the FraSCAti Remote API. Hence, the AM is able to dynamically and remotely add or remove instances while performing scaling up/down of the application without needing to stop the load balancer.

6.2.3 Autonomic Tasks

As it can be seen in Figure 6.6, the AM (class *ApplicationManager*) is implemented as an extension of the class *AutonomicManager* and implements the method *init()* to retrieve the renting fees and available promotions from the IM's public knowledge, and to perform the managed application initial deployment. The access to the public knowledge is performed by class *HttpClientComponent*, which relies on Restlet⁷, a RESTful Java library that includes an HTTP Server. As HTTP and RESTful are respectively standard protocol and architectural style, the technical details of this part of the prototype implementation is not described in this document.

The initial deployment is a classical scaling up case from zero amount of resources to the minimum amount of resources needed to run the application (i.e. the first workload upper threshold, cf. Section 5.2.3). In order to know which is the minimum amount of resources necessary to run the application, the AM observes the resource requirements (class *PerformanceRequirement* in Figure 6.4) for lowest workload and perform a complete MAPE-K loop based on that.

The knowledge is implemented by the class *ApplicationManagerKnowledge* and aggregates the managed application, the current configuration, the available promotions and the ones that are currently being used, the renting fees, and a queue of scheduled handlers that are ready to be executed. In addition, it maintains the current workload threshold (attribute *currentWorkloadThreshold*), which is used to know in which range of workload is the managed application.

⁷<http://www.restlet.org>

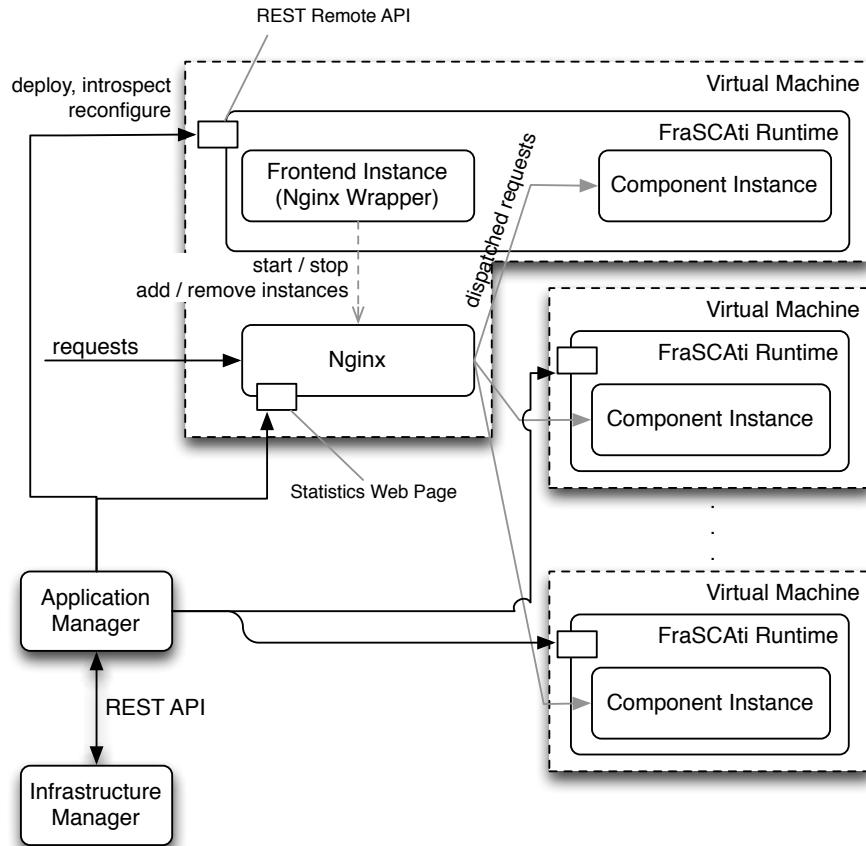


Figure 6.5: Managed Application Technical Details.

Monitoring

The AM monitoring task is performed by a monitor, which is implemented by the class *Monitor* (Figure 6.7) that is provided by the common autonomic framework. It aggregates two objects: one of class *WildCATManager* and another of class *HttpServerComponent*. The former is an extension of *ManagedSystemSensorManager* which is equipped with a complex event processing tool to be able to monitor data from the managed system and to generate endogenous events. The latter extends *InterloopSensorManager* by creating an HTTP server to listen for messages sent by the IM and convert it into interloop or public knowledge changed events.

The AM's autonomic events are implemented as Java classes (*WorkloadIncreased*, *WorkloadDecreased*, *PromotionExpired*, *RentingFeesChanged*, *VirtualMachinesCreated* and *ScaleDown*). Classes *WorkloadIncreased* and *WorkloadDecreased* contains the current managed application's workload (attribute *workload*). The class *PromotionExpired* contains a list of expired promotions. The class *RentingFeesChanged* aggregates the renting fees and promotions, whereas the class *VirtualMachinesCreated* aggregates the list of VMs and promotions created. Finally, the class *ScaleDown* aggregates a list of VMs that should be released and the timeout given by the IM to the reconfiguration take effect.

As stated before, an application has one or several promoted services that are offered by some components. That way, there is one frontend sensor (class *FrontendSensor*) per component that offers a promoted services. More precisely, every component frontend (class *ComponentFrontend*) has a sensor, and those whose components offer promoted services are added to the *WildCATManager*.

For complex event processing, we rely on WildCAT [DL05], a Java framework for context-

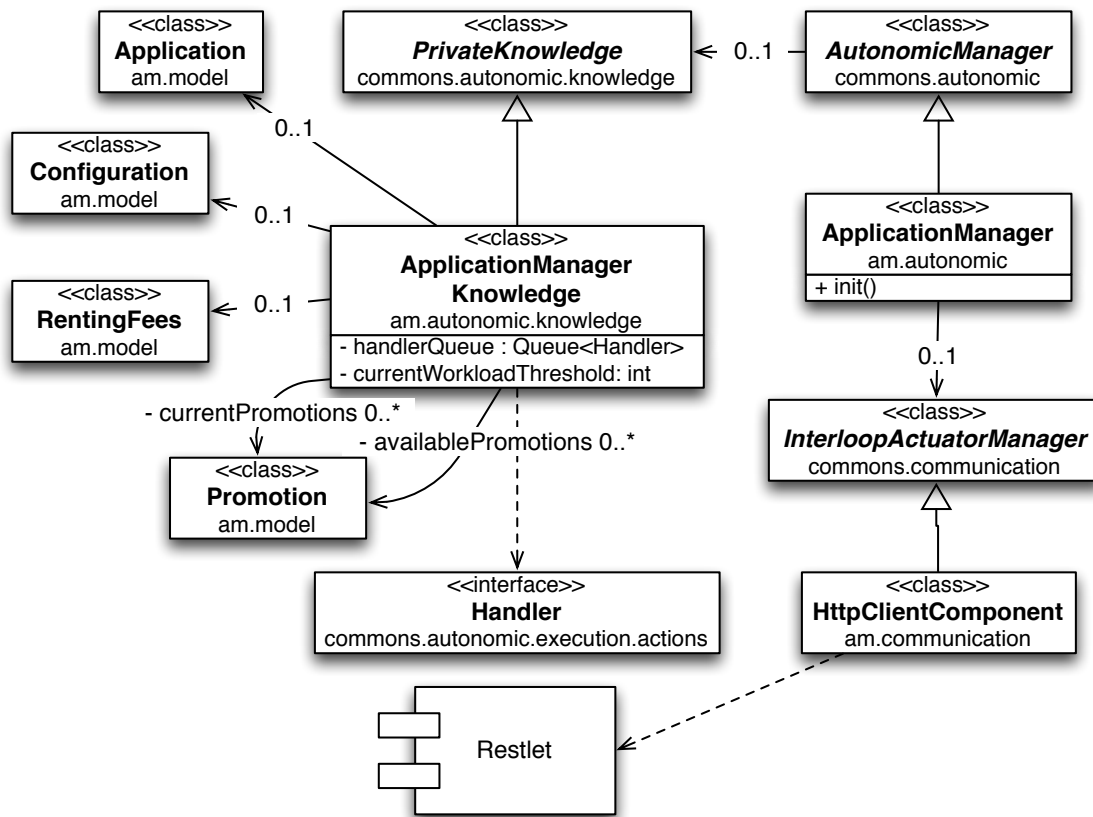


Figure 6.6: Application Manager Autonomic Meta-model.

aware applications that allows the deployment of sensors for monitoring large applications. The application is defined as a hierarchy of resources (cf. Figure 6.8), where circles represent resources and squares represent resources' attributes. The hierarchy root (*self://*) contains the resource *self://application* (representing the application itself), which contains the resource *self://application/frontends*. The latter may contain several attributes *self://application/frontends#frontend_**, each one representing the component frontend subscribed to the monitor. These attributes are used to store the monitoring data from the respective component frontends.

Attributes can be of three types: basic attributes, which hold static values; active attributes, which are fed by sensors deployed on the target application; and synthetic attributes, resulting from expressions on other attributes. That way, each attribute *self://application/frontends#frontend_** (Figure 6.8) is implemented as a sensor (represented by class *FrontendSensor*), which is in charge of retrieving the statistical values from the running component frontend and updating the resource hierarchy with the just-retrieved values. Any Plain-Old-Java-Object (POJO) is allowed to be stored in the hierarchy. At last, a poller is created and attached to the sensor in order to retrieve the data in a periodic manner.

The WildCAT event model consists of two types of events: *WHierarchyEvents*, which report events related to the hierarchy (e.g. creation of an attribute or a resource in the hierarchy); and *WAttributeEvents*, which are emitted every time an attribute value is changed.

In order to process events, WildCAT relies on Esper⁸, an engine for complex event processing. In Esper, one can define queries on events by using Event Query Language (EQL), a language whose syntax is similar to Structured Query Language (SQL), which is used for querying database tables. Listing 6.1 defines a query in EQL to detect the events *WorkloadIncreased* and *WorkloadDecreased* on the managed system (application).

⁸<http://esper.codehaus.org/>

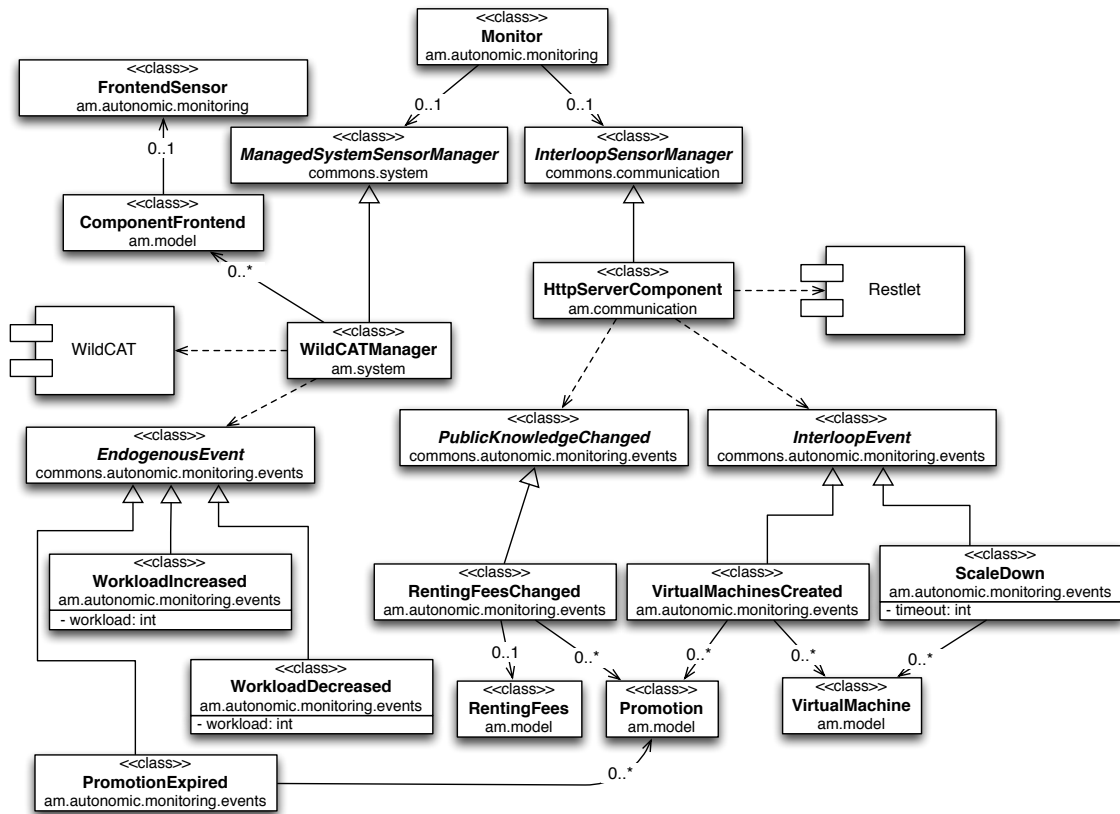


Figure 6.7: Application Manager Monitoring Task Meta-model.

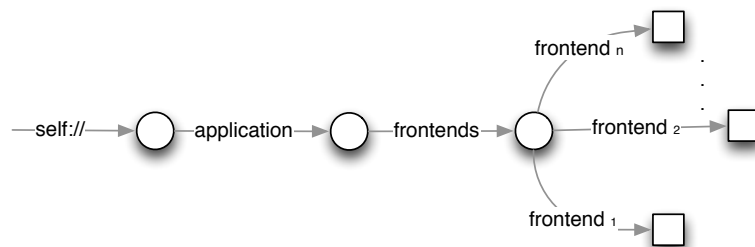


Figure 6.8: Application Manager Wildcat Resource Hierarchy.

```

SELECT avg(current.value('workload')) as avgWorkload
  FROM WAttributeEvent(source = '@PATH_TO_ATTRIBUTE@'),
       win:time_batch("@TIME_BATCH@_min") AS current
HAVING avg(current.value('workload'))
        <= @LOWER_WORKLOAD_THRESHOLD@
       OR avg(current.value('workload'))
        > @UPPER_WORKLOAD_THRESHOLD@

```

Listing 6.1: Workload Increased/Decreased EQL Query.

Basically, it selects the average of field *workload* from attribute '@PATH_TO_ATTRIBUTE@' within a time window of '@TIME_BATCH' minutes in which the average is less or equal a given '@LOWER_WORKLOAD_THRESHOLD@' or greater than a given '@UPPER_WORKLOAD'

`_THRESHOLD@'`. In other words, it retrieves the average workload when the current average workload exceeds a lower/upper bound. The query is registered along with an action that is executed every time a record is retrieved. Lastly, the action creates the endogenous (*Workload Increased* or *Workload Decreased*) according to the exceeded boundary.

Analysis

The analysis task is performed by an analyzer, which may behave differently according to the detected event. In the AM, there are two possible scenarios: a workload variation and scale down scenario. The first scenario occurs whenever the endogenous events *Workload Increased/Decreased* or *Promotion Expired* are detected. In this case, the AM launches the analysis implemented by the class *ArchitectureCapacityAnalyzer*, as depicted in Figure 6.9. This analysis is modeled as a constraint satisfaction and optimization problem (CSOP) [RVBW06], as described in Section 5.2.3. To solve this problem, class *ArchitectureCapacityConstraintSolver* takes as input the current workload, the renting fees, the application definition, the current configuration, the current promotions allocated to the application, and returns as result the most suitable architectural configuration and the resource allocation to each component of this configuration. The class *ArchitectureCapacityAnalysisResult* aggregates the information resulting from the solver.

The second scenario happens when the AM detects an event *Scale Down*. It is similar to the first scenario, except for the fact that the IM imposes a constraint on the resources. Hence, the application can only shrink and never stretch. The AM launches the analyzer which is implemented by class *ScaleDownAnalyzer*. It is also modeled as a CSOP problem that is solved by class *ScaleDownSolver*. Similarly, it takes as input the same arguments as in *ArchitectureCapacityConstraintSolver* as well as the VMs that should be released and also returns the most suitable architectural configuration and the resource allocation to each component of this configuration. The class *ScaleDownAnalysisResult* gathers the information resulting from the solver.

It is noteworthy that both solvers (classes *ArchitectureCapacityConstraintSolver* and *ScaleDownConstraintSolver*) rely on Choco [Tea10], a constraint programming library implemented in Java.

Planning

Figure 6.10 presents the AM planning task meta-model. The planning task is performed by single planner implementation (class *ArchitectureCapacityPlanner*) regardless the preceding analyzer. This implementation corresponds to the *Architecture and Capacity Planner*, which is described in Algorithms 6, 7, 8, 9, 10 and 11. The result of both of them is an object of class *Plan* that contains a set of autonomic actions described in Section 5.2.3. The next section details the implementation of such actions.

Execution

The AM execution task is performed by an executor, which is implemented by the class *Executor* (cf. Figure 6.2). It takes as input the plan resulting from the planning task and executes it. This plan (class *Plan* in Figure 6.3) is composed of several tasks, which in turn, are composed of autonomic actions. Figure 6.11 presents the AM interloop actions. The action implemented by the class *RequestVirtualMachines* aggregates a map of objects of the class *VirtualMachineClass* to the respective desired number of virtual machines, and a list of desired promotions (class *Promotion*). The action implemented by the class *ReleaseVirtualMachines* aggregates the list of virtual machines to be released (class *VirtualMachine*). Both classes extend from the abstract class *InterloopAction* which relies on the abstract class *InterloopActuatorManager* to communicate with the IM and to perform the action. So, this class is specialized by class *HttpClientComponent*, which relies on Restlet to act an HTTP client and access web resources offered by the IM in a RESTful manner.

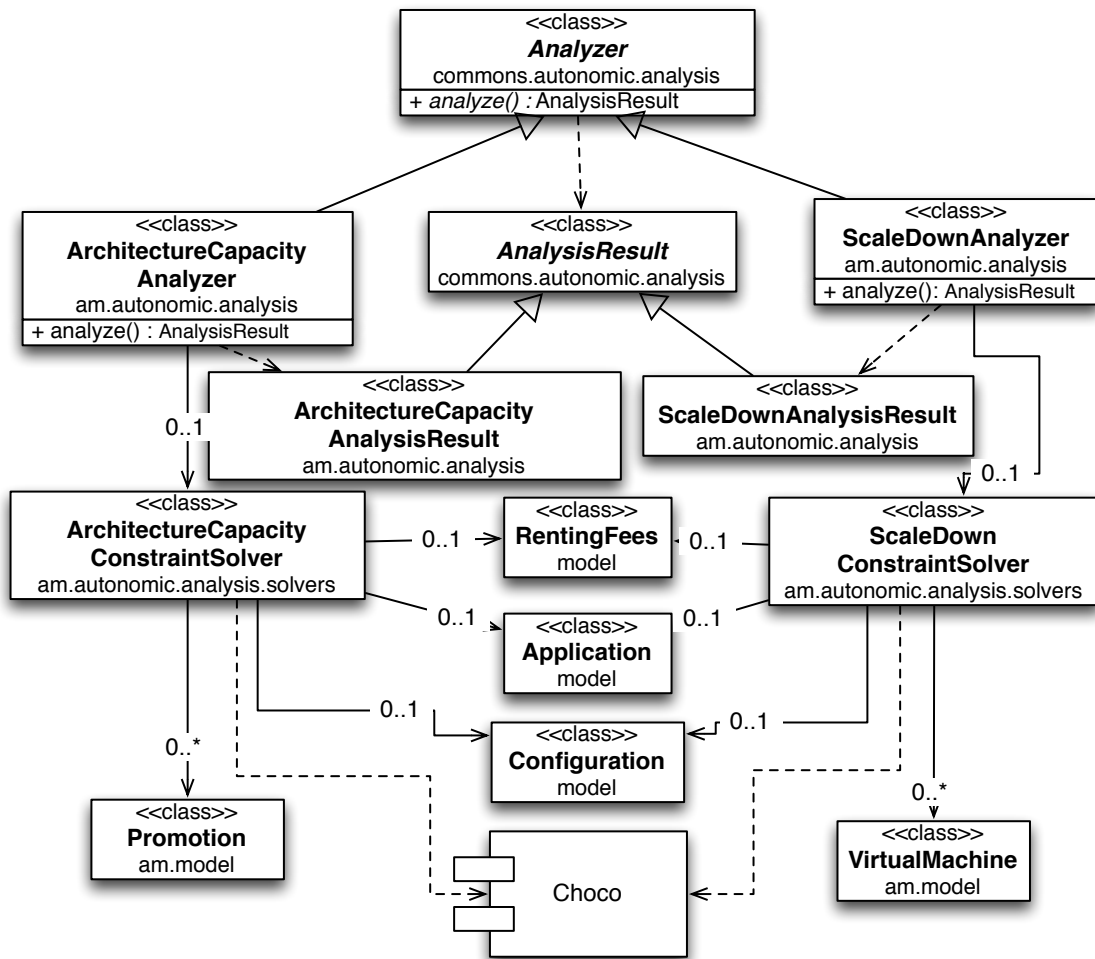


Figure 6.9: Application Manager Analysis Meta-model.

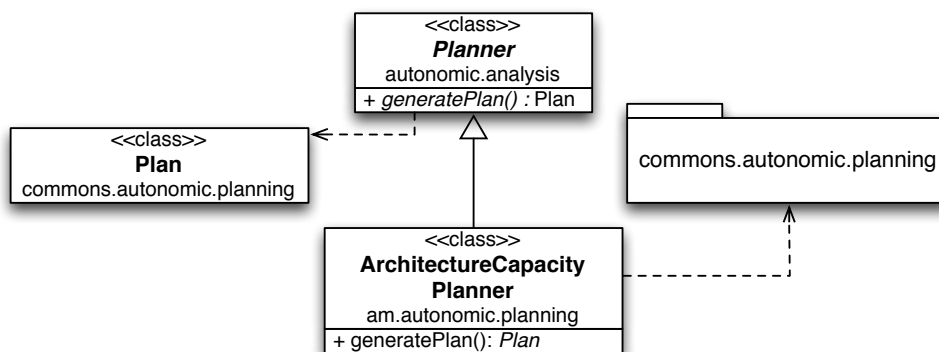


Figure 6.10: Application Manager Planning Meta-model.

Figure 6.12 depicts the actions on the managed system. The classes implementing these actions extend from the class *ActionOnManagedSystem*, which relies on the abstract class *ManagedSystemActuatorManager* to perform the actions on the managed system. In this context, the class *FraSCAtiAPIManager* extends from class *ManagedSystemActuatorManager* by relying on the FraSCAti Remote API to perform the deployment, introspection and reconfiguration of FraSCAti applications via RESTful interfaces.

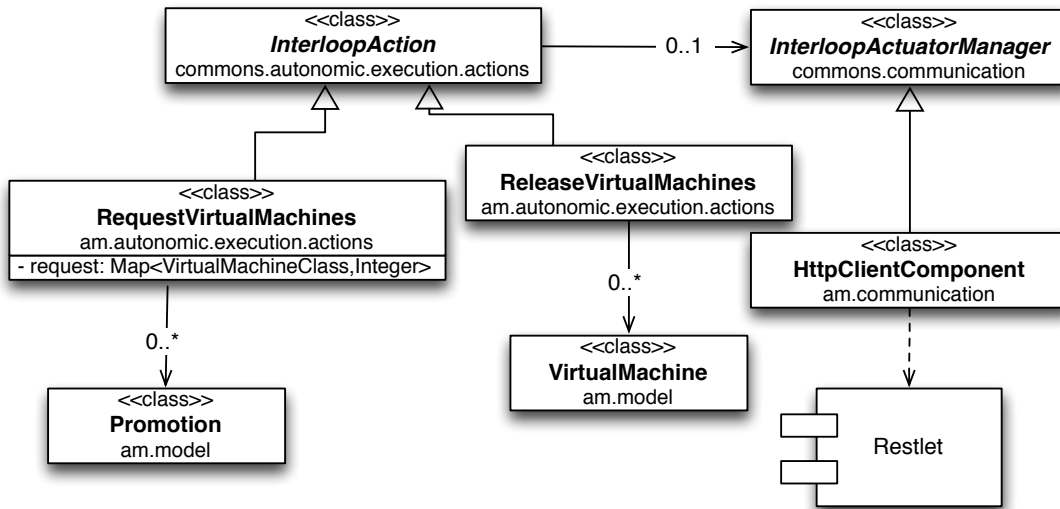


Figure 6.11: Application Manager Execution Meta-model (Interloop Actions).

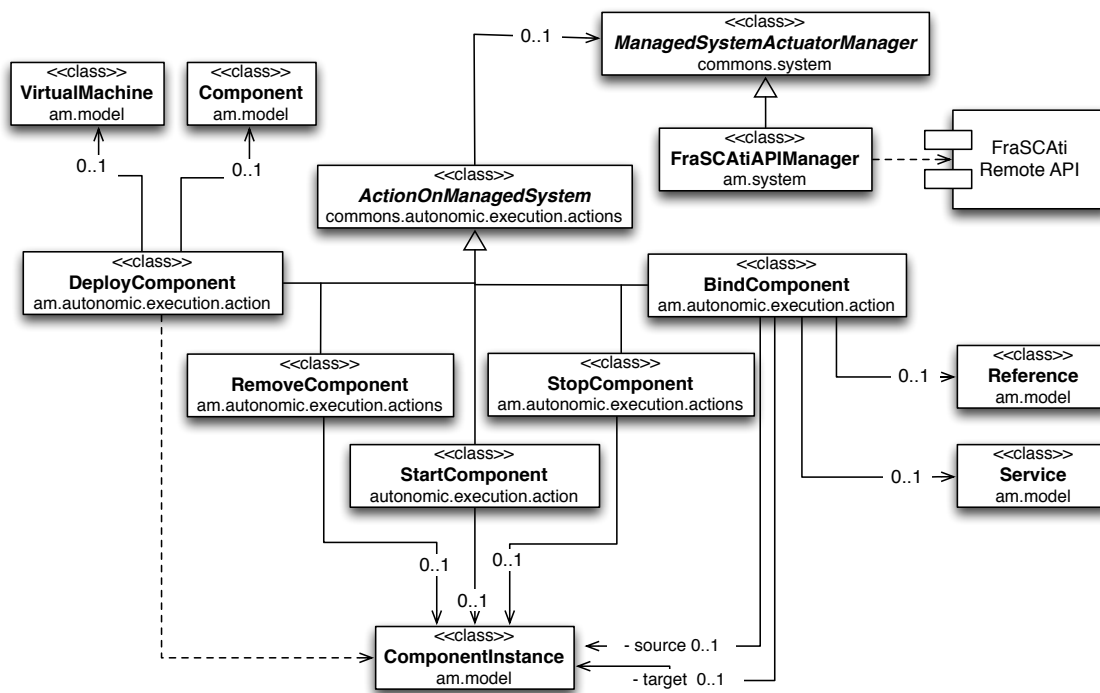


Figure 6.12: Application Manager Execution Meta-model (Actions on the Managed System).

The class *DeployComponent* aggregates classes *VirtualMachine* and *Component* and makes use of the deployment interface to perform the deployment. It consists in uploading a compressed file that contains the component definition and the respective implementation. For the rest of the actions, the AM relies on the reconfiguration interface. Listing 6.2 show how those actions are implemented in FraSCAti Script. Basically, the script updates the value of the wrapper property *instances*, which is a string value that contains the list of component instances that should be added to the component frontend. In order to remove an instance (class *RemoveComponent*), it is necessary to regenerate the list without the undesired instance.

```
#adding to/ removing from frontend
set-value($domain/scachild::wrapper/scaproperty
::instances, '@INSTANCES@');

#removing component
sca-remove(' @COMPONENT_NAME' );
```

Listing 6.2: FraSCAti Script Definition for Adding/Removing Component Instances to/from Component Frontend

The classes *StartComponent* and *StopComponent* aggregates class *ComponentInstance*. They consists in starting/stopping a given component instance. These actions are performed by the FraSCAti Script defined in Listing 6.3. It simply calls the FraSCAti Script action *start/stop* with the targeted component as parameter. This works for both components instances and component frontends. Indeed, once a fronted wrapper is stopped, it stops the associated component frontend (ngnix component).

```
#starting component
start($domain/scachild::@COMPONENT_NAME@);

#stopping component
stop($domain/scachild::@COMPONENT_NAME@);
```

Listing 6.3: FraSCAti Script Definition for Starting/Stopping a Component

The action *BindComponents* links one component's reference to another components' service. The FraSCAti Script implementation for this action is presented in Listing 6.4. First, it retrieves the first and single SCA binding of reference *@REFERENCE_NAME@* of component *@COMPONENT_NAME@*. Then the binding is stopped, its value is updated with a new URI pointing to the service (of the target component frontend) that it should be linked to, and started again.

```
bind = $domain/scachild::@COMPONENT_NAME@/scareference
::@REFERENCE_NAME@/scabinding::*[1];
set-state($bind,"STOPPED");
set-value($bind/attribute::uri, '@SERVICE_URI@');
set-state($bind,"STARTED");
```

Listing 6.4: FraSCAti Script Definition for Binding Component Instances to Other Components' Frontends

It is important to mention that, since the AM does not have any public knowledge, it does not implement any action of class *ChangePublicKnowledge*.

6.3 Infrastructure Manager

This section details the implementation of the IM. We firstly present the infrastructure meta-model and technical details in order to ease the understanding of the managed system. Then, we present a set of class diagrams which depicts the implementation of each autonomic task.

6.3.1 Infrastructure Meta-model

Figure 6.13 presents the infrastructure meta-model. The infrastructure consists of a set of physical machines (PMs) which are implemented by the class *PhysicalMachine*. A PM can host one or several VMs. A VM is represented by the class *VirtualMachine* and is composed of a VM class (implemented by the class *VirtualMachineClass*) and the application to which it is leased. Promotions are implemented by the class *Promotion* and each promotion has an attribute *tokenId*, which

is used for synchronization purposes. In addition, it is associated to an application to which the promotion is sold. Renting fees are implemented by the class *RentingFees* that aggregates a list of VM classes and a list of promotions. *VirtualMachineRequest* is a class that implements the requests sent by the AM on VMs and promotions. Hence, it aggregates a mapping of number of VMs desired of each VM class, and a list of desired promotions.

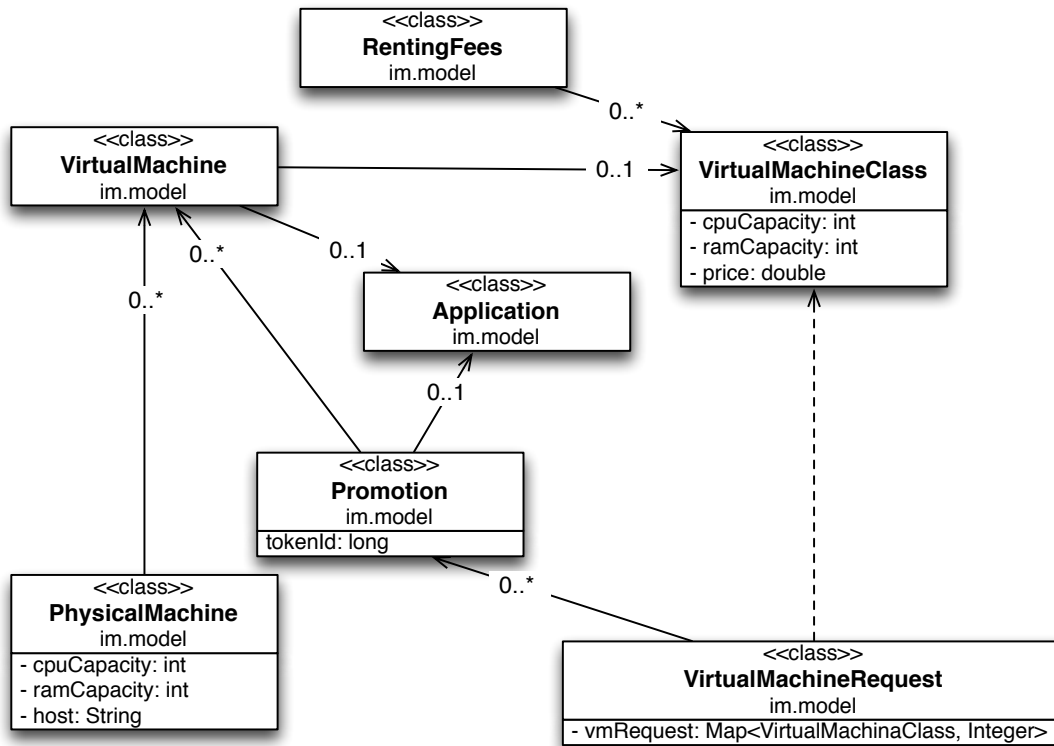


Figure 6.13: Infrastructure Meta-model.

6.3.2 Managed System Technical Details

In order to manage VMs, the IM relies on Xen⁹, a Virtual Machine Monitor (VMM), also known as hypervisor. Basically, it allows the simultaneous execution of several guest VMs on the same PM. It runs directly on top of a PM (*bare metal hypervisor*) instead of relying on an operating system to interface the hardware. Xen guest VMs are known as domains. *dom0* is a special domain which is in charge of controlling the hypervisor and starting/stopping new guest VMs.

Figure 6.14 illustrates the managed infrastructure. We assume that the PMs are equipped with Xen hypervisor and that the *xen-hyper* script is placed in the domain *dom0* of each PM. *xen-hyper* is script written in Shell Script [Mic08] that eases the management of VMs on top of Xen. Apart from that, there must exist one or several reference disk image and one reference Xen-based configuration file. The reference image disk should be suitable to the hypervisor and must contain an operating system installed. So, we have generated a Xen-based reference image disk with Debian Lenny linux distribution and FraSCAti installed. The reference configuration file is used as a template file every time a VM is created.

xen-hyper consists of two main procedures: *create* and *destroy*. The procedure *create* performs the following tasks:

1. Make a copy of the image reference.

⁹<http://xen.org>

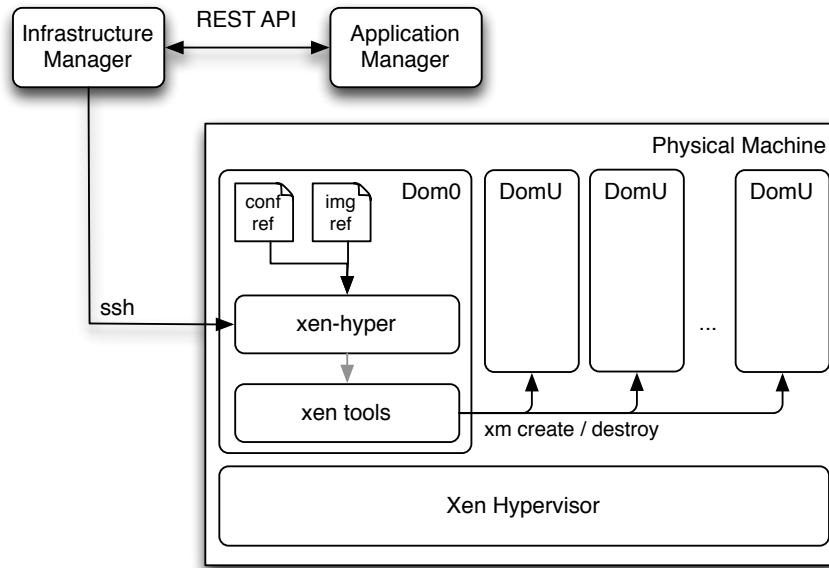


Figure 6.14: Managed Infrastructure Technical Details.

2. Configure the network inside the image
3. Generate a Xen configuration file with all the VM specification: name, CPU and RAM requirements, and network specification (IP and MAC addresses).
4. Execute the Xen Tools command to create the VM based on the configuration file
5. Waits until the VM reboots
6. Launch FraSCAti Runtime in the VM

The procedure *destroy* behaves as follows:

1. Check if the VM really exists
2. Execute the Xen Tools command to destroy the VM.
3. Delete the image file.

Listing 6.5 shows the way these two procedures should be used. @VM_NAME@ corresponds to the name of the VM that is being created, @CPU@ represents the number of CPU cores required for the VM, @RAM@ corresponds to the amount of RAM required for the VM, @IP@ means the VM IP address, @PORT@ represents the port that the FraSCAti runtime should listen on.

```
#vm creation
xen-hyper create -v @VM_NAME@ -c @CPU@ -r @RAM@ \
                -i @IP@ -p @PORT@
```

```
#vm supression
xen-hyper destroy -v @VM_NAME@
```

Listing 6.5: Virtual Machine Creation/Destruction Command.

In order to execute *xen-hyper*, the IM performs a simple Java system call that remotely executes the command via SSH protocol.

It is important to mention that the choice for Xen as hypervisor is due to compatibility reasons, meaning that until the time this work was being validated, Xen was the most suitable hypervisor in the testbed environment used for the validation, namely Grid5000¹⁰. Even though, the support for other hypervisors can be easily implemented. It is necessary however to generate the reference images and configuration file for the target hypervisor. Besides, the script *xen-hyper* should also be adapted to suit the target hypervisor.

6.3.3 Autonomic Tasks

Figure 6.15 depicts the IM autonomic meta-model. Similar to the AM, the IM is implemented as an extension of the class *AutonomicManager*. However, it does not do anything special inside the method *init()*. It aggregates a private knowledge, which is implemented by the class *InfrastructureKnowledge*, which in turn, extends from the class *PrivateKnowledge*; and a public knowledge, which is implemented by the class *InfrastructurePublicKnowledge*, which in turn, extends from the class *PublicKnowledge*. The latter relies on the abstract class *InterloopSensorManager* to listen for requests from other autonomic managers. The class *HttpServerComponent* extends the class *InterloopSensorManager* by creating an HTTP server to listen for requests from other autonomic managers willing to access the public knowledge. The private knowledge is composed of the current promotions, that is, the sold promotions, and the list of PMs (class *PhysicalMachines*). From this list, it is possible to know which VMs are hosted by each PM. The public knowledge is composed of the renting fees (class *RentingFees*) and a list of available promotions (class *Promotion*). As promotions should have concurrency control, each promotion is considered as a critical section and for this reason class *Promotion* extends from class *CriticalSection*.

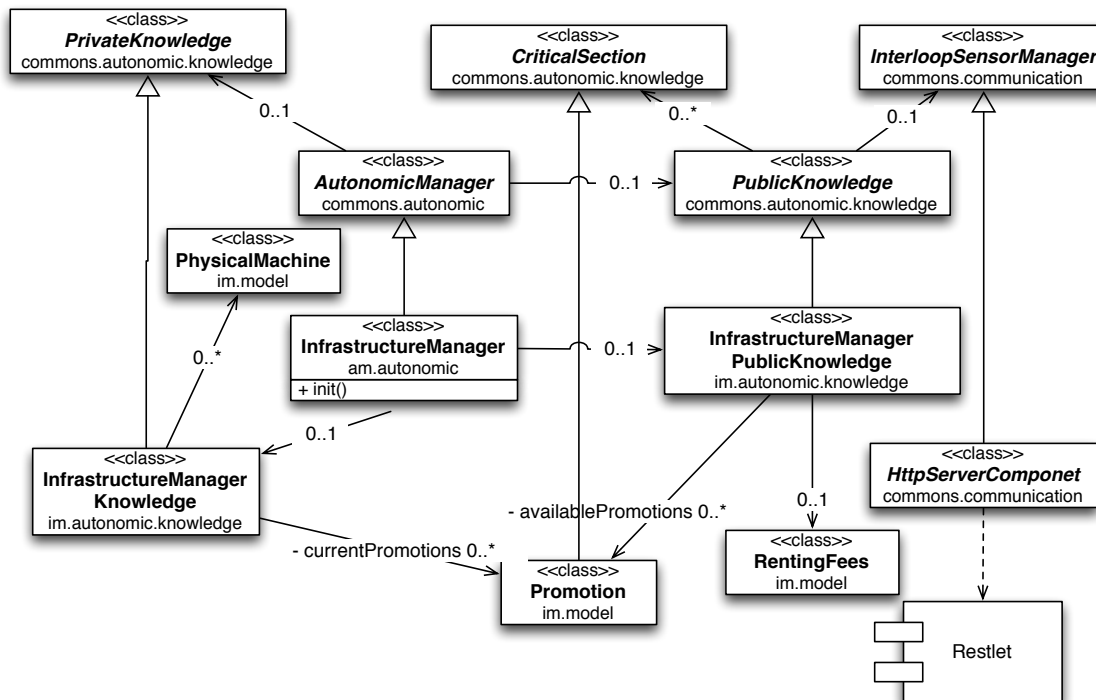


Figure 6.15: Infrastructure Manager Autonomic Meta-model.

¹⁰<https://www.grid5000.fr>

Monitoring

The IM monitoring task is implemented in a very similar way to the AM. It relies on the class *Monitor*, which is provided by the common autonomous framework. As it is shown in Figure 6.16, the monitor also aggregates two objects: one implemented by the class *WildCATManager* and another implemented by the class *HttpServerComponent*, both objects play the same role as in the AM implementation.

The IM autonomous events are also implemented as Java classes (*LowPhysicalMachineUtilization*, *PhysicalMachineUnused*, *EnergyShortage*, *TimeoutExpired*, *VirtualMachinesRequested* and *VirtualMachinesReleased*). The classes *PhysicalMachineUnused*, *TimeoutExpired* and *LowPhysicalMachineUtilization* aggregate a list of PMs (class *PhysicalMachine*) that should be powered off (for the two first events) or receive promotions (for the last one). The class *EnergyShortage* has an attribute to indicate the number of PMs that should be powered off. This is an endogenous event that can be triggered either manually by the administrator or based on data obtained from power measurements. The way this event is triggered is out of the scope of this work. The class *VirtualMachinesRequested* aggregates a mapping that indicates the number of VMs desired of each VM class, and the list of desired promotions. Lastly, the class *VirtualMachinesReleased* is composed of a list of VMs to be released (class *VirtualMachine*).

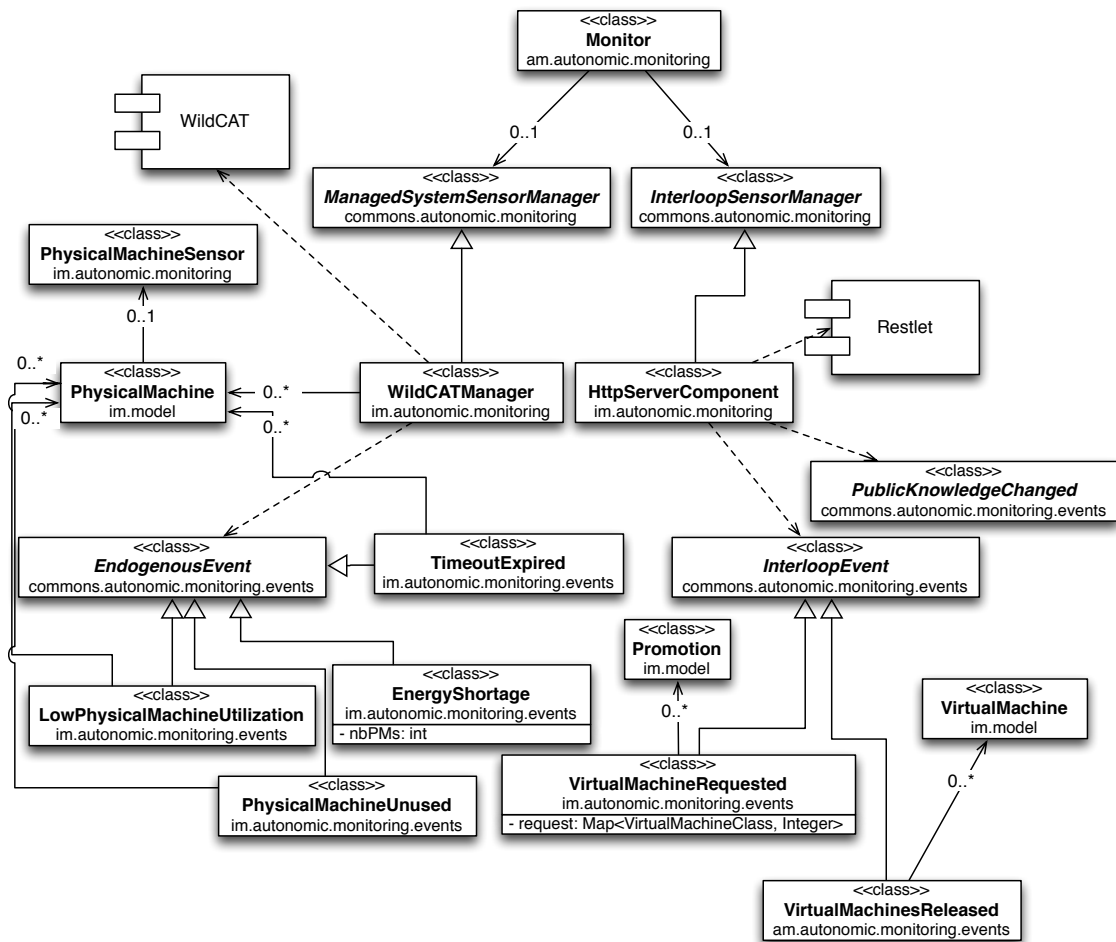


Figure 6.16: Infrastructure Manager Monitoring Meta-model.

Figure 6.17 shows the WildCAT resource hierarchy for the infrastructure. It consists of a resource representing the infrastructure itself (*self://infrastructure*), followed by another resource that represents the set of PMs (*self://infrastructure/pms*). Finally, this resource may have several attributes, each one containing the monitoring data of a specific PM.

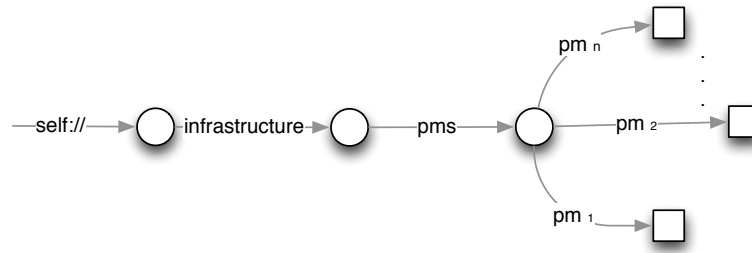


Figure 6.17: Infrastructure WildCAT Resource Hierarchy.

The IM implements two EQL queries: one for creating and firing the event *LowPhysicalMachineUtilization* (described in Listing 6.6); and another that creates and fires the event *Physical Machine Unused*.

The query described in Listing 6.6 retrieves any attribute in *self://infrastructure/pms* (any PM) whose emitted events follow a specific pattern. That is, it takes every PM that is powered on and whose utilization is lower than a certain value (`@UTILIZATION_RATE@`) and remains lower for a certain amount of time (`@TIME@`).

```
select a.source from pattern [
  every a=WAttributeEvent(
    source like 'self://infra/pms#pm%' and
    value('on')=true and
    value('utilization') < @UTILIZATION_RATE@)
-> (timer:interval(@TIME@ min) and not
WAttributeEvent(source = a.source and
value('utilization') > @UTILIZATION_RATE@))]
```

Listing 6.6: Low Physical Machine Utilization EQL Query.

The EQL query described in Listing 6.7 is a particular case of the one described in Listing 6.6. In fact, it retrieves any attribute in *self://infrastructure/pms* (any PM) that is powered on and whose utilization is equal to zero and remains equal to zero for a certain period of time (`@TIME@`).

```
select a.source from pattern [
  every a=WAttributeEvent(
    source like 'self://infra/pms#pm%' and
    value('on')=true and
    value('utilization') = 0)
-> (timer:interval(@TIME@ min) and not
WAttributeEvent(source = a.source and
value('utilization') > 0))]
```

Listing 6.7: Physical Machine Unused EQL Query.

Both queries are registered along with an action that creates and fires the respective autonomic event: *Low Physical Machine Utilization* or *Physical Machine Unused*.

Analysis

Figure 6.18 shows the IM analysis task meta-model. In the IM, the analysis task is launched only when the autonomic events *Virtual Machines Requested* or *Energy Shortage* are detected. For all the other IM autonomic events, the analysis task is dispensable, as already explained in Section 5.2.2. Every time the event *Virtual Machines Requested* is detected, the IM launches the analysis implemented by the class *VirtualMachinePlacementAnalyzer*. The analyzer is instantiated with

information sent along with the event such as the number of VMs of each class, the desired promotions and the requesting application. This information is encapsulated by the request message (class *VirtualMachineRequest*). Like the AM analysis task, this analyzer is also modeled as a CSOP, as detailed in Section 5.2.2. In order to solve this problem, the analyzer relies on the solver implemented by the class *VirtualMachinePlacementConstraintSolver*. It takes as input all the PMs and the just-requested VMs. The solver returns the optimal mapping VMs hosted on PMs, which is then encapsulated in an object of the class *VirtualMachinePlacementAnalysisResult*.

When the IM detects the event *Energy Shortage*, the IM launches the analyzer implemented by the class *EnergyShortageAnalyzer*. The analyzer is created with the number of PMs (attribute *nbPMs*) that should be shutdown as well as the list of existing PMs. Again, this analyzer is also modeled as a CSOP, which is solved by the solver implemented by the class *EnergyShortageConstraintSolver*. It takes as input all the existing PMs and returns the PMs that should be switched off. This result is encapsulated by the class *EnergyShortageAnalysisResult*.

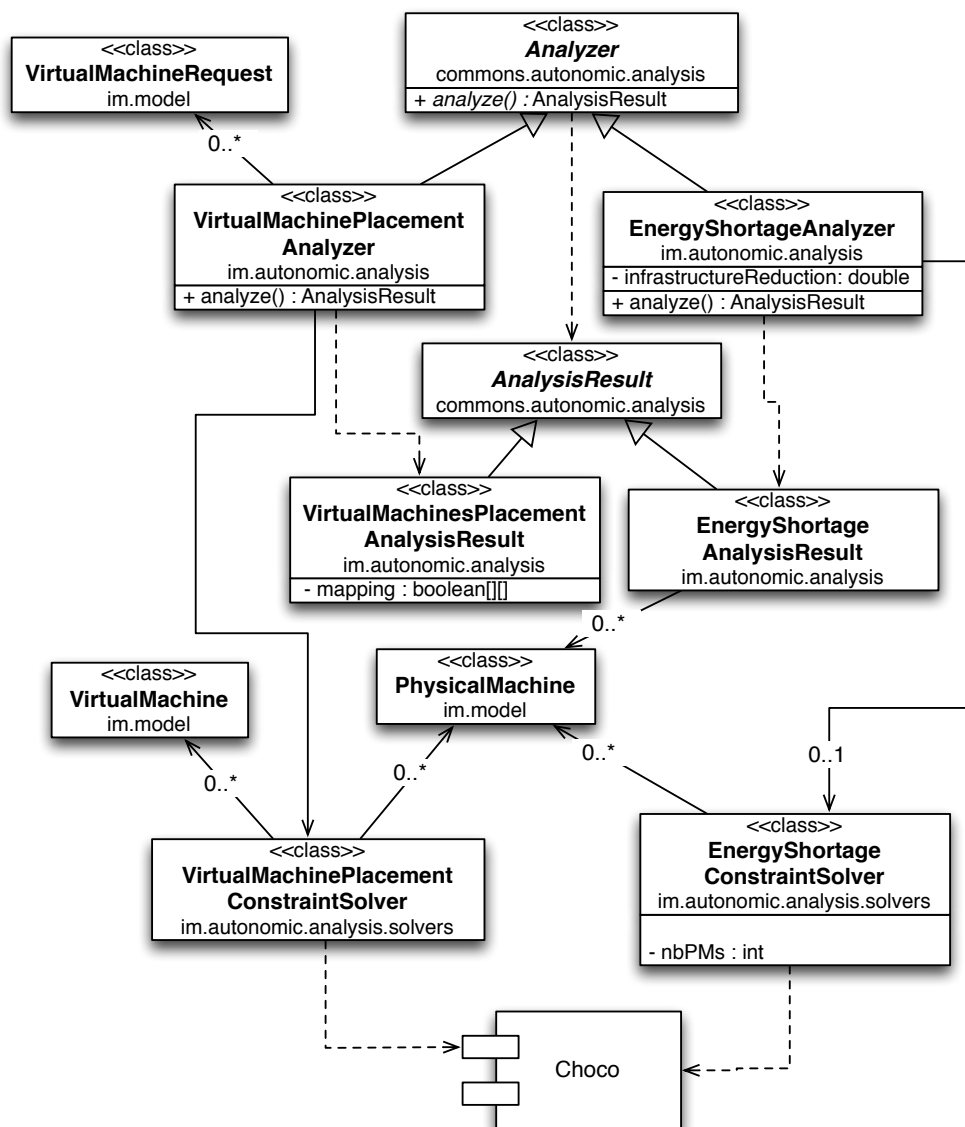


Figure 6.18: Infrastructure Manager Analysis Meta-model.

Like the AM solvers, the implementation of the IM solvers (*VirtualMachinesPlacementConstraintSolver*, and *EnergyShortageConstraintSolver*) rely on Choco.

Planning

The planning task can be performed by five different planners: one for each autonomous event. Figure 6.19 shows the meta-model for the IM analysis task. The class *VirtualMachinePlacementPlanner* implements the homonym planner, which is described by Algorithm 1. It takes as input the knowledge (public and private), and the list of VMs that should be created (class *VirtualMachines*). The class *PromotionPlanner* implements the *Promotion Planner*, which is described in Algorithm 3. It also takes as input the public knowledge and the PMs in which promotions should be created (class *PhysicalMachines*). The class *PhysicalMachineUnusedPlanner* implements the homonym planner, which is described in Algorithm 4. It takes as input the private knowledge and the list of PMs that should be switched off (*PhysicalMachine*). The class *EnergyShortagePlanner* implements the *Energy Shortage Planner*, which is described in Algorithm 5. It also takes as input the public knowledge and the list of PMs that should be switched off so as to cope with the energy shortage condition. Finally, the class *VirtualMachinesReleasePlanner* implements the *Virtual Machine Release Planner*, which is described in Algorithm 2. It takes as input the private knowledge and the list of VMs that should be released (class *VirtualMachine*).

The result of all those planners is an object of the class *Plan*, which is composed of a set of autonomous actions. The next section details the implementation of those action and how they are executed.

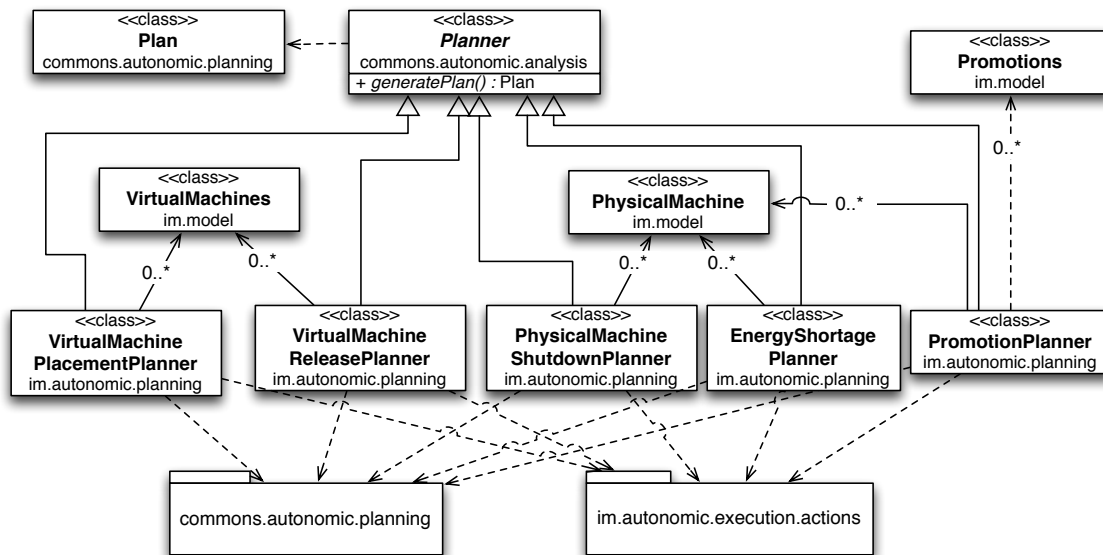


Figure 6.19: Infrastructure Manager Planning Meta-model.

Execution

Similar to the AM implementation, the IM execution task is performed by an executor implemented by the class *Executor* (Figure 6.3). Figure 6.20 shows the IM interloop actions *NotifyVirtualMachinesCreated*, *NotifyScaleDown* and *NotifyRentingFeesChanged*. The first two actions aggregate each one a list of VMs (class *VirtualMachine*) which represents respectively either the just-created or the to-be-released VMs. The third one aggregates a list of promotions (class *Promotion*) which correspond to the just-created promotions. They are specialization of the abstract class *InterloopAction* which relies on the abstract class *InterloopActuatorManager* to communicate with the AMs (represented by the class *Interlocutor*). Like the AM implementation, the class *HttpClientComponent* extends the class *InterloopActuatorManager* by relying on Restlet to act as an HTTP client and access web resources offered by the AM in a RESTful manner.

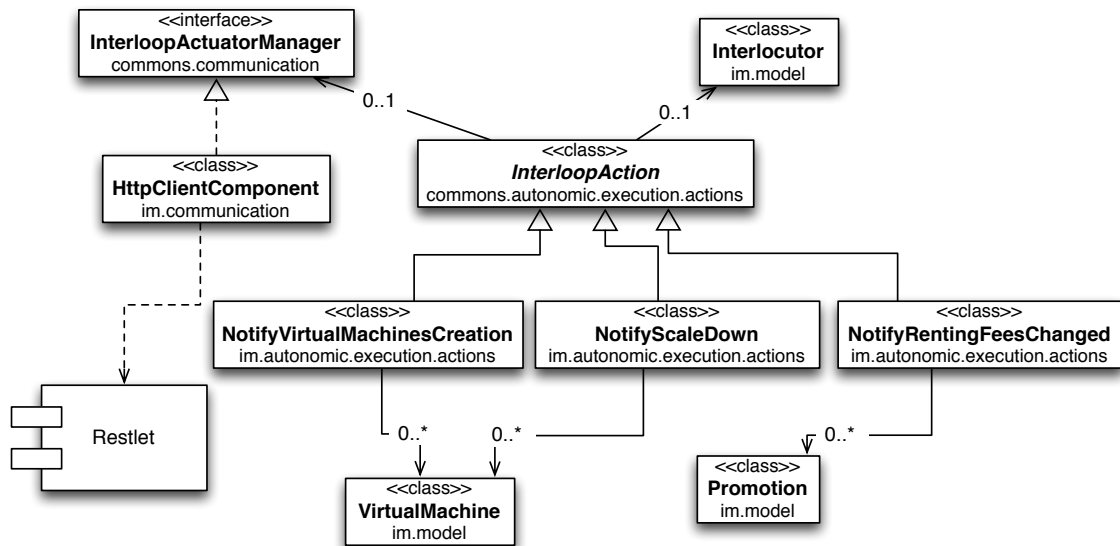


Figure 6.20: Infrastructure Manager Execution Meta-model for Interloop Actions.

Figure 6.21 presents the Change Public Knowledge actions implemented by classes *CreatePromotion* and *ActivatePromotion*. Each class extends from the abstract class *ChangePublicKnowledge* by aggregating the class *Promotion*, which represents the promotion to be created or activated on the public knowledge.

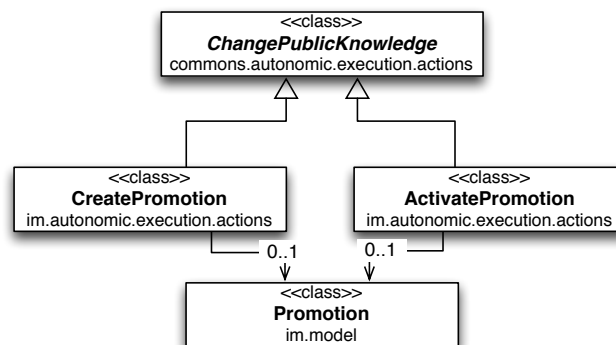


Figure 6.21: Infrastructure Manager Execution Meta-model for Change Public Knowledge Actions.

The IM actions on the managed system are shown in Figure 6.22. The objective is to perform actions on the existing PMs of the infrastructure. To this end, the IM relies on SSH to perform these actions remotely on the target PMs.

The actions implemented by the classes *SwitchPhysicalMachineOn* and *SwitchPhysicalMachineOff* both extend from the class *ActionOnManagedSystem* by aggregating the class *PhysicalMachine*, which corresponds the PM that should be switched on or off. The implementation of the class *SwitchPhysicalMachineOff* is very simple, since only a system call (e.g. 'shutdown' on Linux based operating systems) on the PMs side is needed. However, the implementation of the class *SwitchPhysicalMachineOn* depends whether or not a wake-on-lan mechanism is supported and/or enabled on the target hardware/platform (i.e. the network interface controller and/or operating system). That means, whether a PM is capable of being powered on via its network interface. If that is the case, the implementation of *SwitchPhysicalMachineOn* would consist simply in a system call to wake the PM up via its network interface (e.g. 'wakeonlan network_interface_address')

in Linux based operating systems).

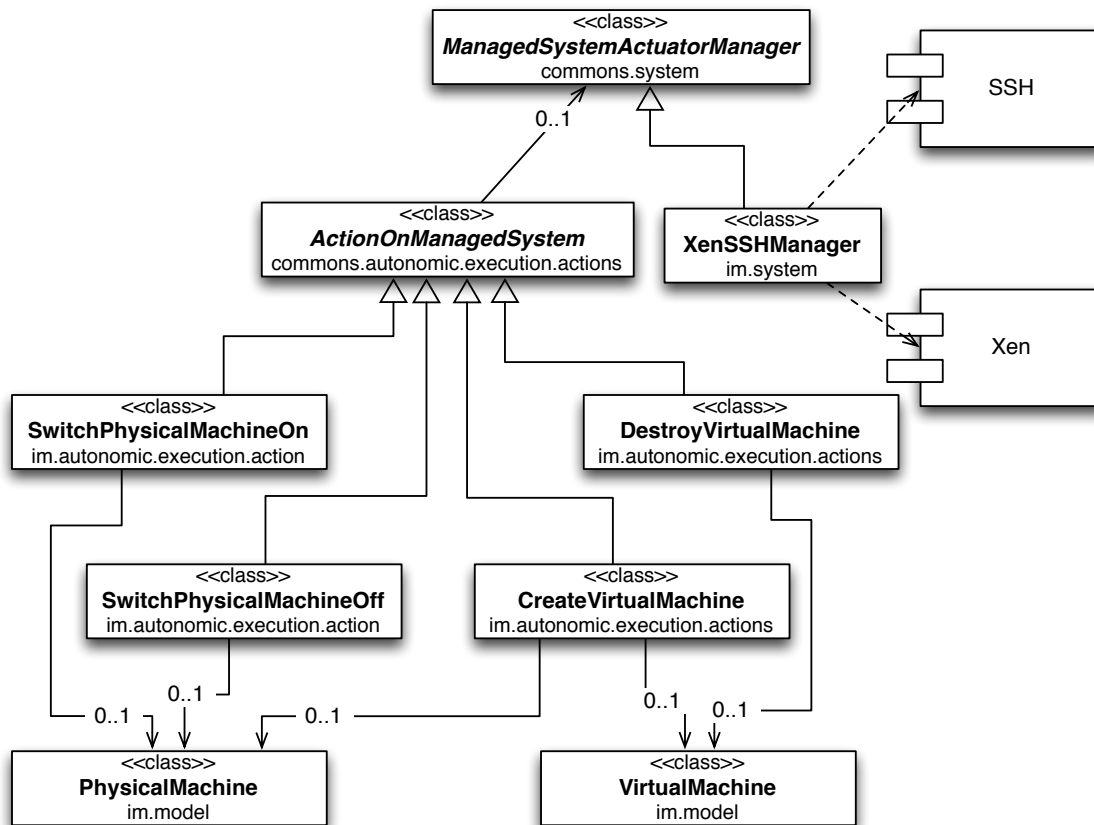


Figure 6.22: Infrastructure Manager Execution Meta-model for Actions on the Managed System.

The action implemented by the class *DestroyVirtualMachine* extends from the class *ActionOnManagedSystem* by aggregating the class *VirtualMachine*, which represents the VM that should be destroyed. Regarding the action implemented by the class *CreateVirtualMachine*, it also extends from the class *ActionOnManagedSystem* by aggregating the class *VirtualMachine*, which corresponds to the VM that should be created. In addition this action also aggregates class *PhysicalMachine*, which corresponds to the PM where the VM should be deployed in.

Finally, the actions implemented by the classes *CreateVirtualMachine/DestroyVirtualMachine* perform a system call to remotely execute (via SSH) the procedures *create* or *destroy* of the *xen-hyper* script (cf. Section 6.3.2).

6.4 Autonomic Managers Lifetime

This section describes how autonomic managers interact with other objects (internal objects and other autonomic managers) over the time. For this purpose, we present a set of sequence diagrams which objective is to ease the understanding of autonomic managers progression over a period of time.

Figure 6.23 describes how events are created and managed over the time. It is important to recall that events can be either endogenous or exogenous. Endogenous events are fired based upon fine-grained events from the managed system. Once those fine-grained events are detected by the managed system sensor (object of the class *WildCATManager*), an endogenous event is created with some monitoring data (e.g. the current application workload), and fired. This makes the manager (object of the class *ApplicationManager*) be notified on the event creation.

With respect to exogenous events, an interloop actuator (object of class *HttpClientComponent*) from the IM side sends an HTTP message to the AM. This message is received by an interloop sensor (object of the class *HttpServerComponent*), which interprets the content of the message by creating some event-specific objects, creates the autonomic event object and fires it. At last, the AM is notified of the event reception.

It is noteworthy that although the diagram in Figure 6.23 shows the autonomic events creation only for the AM, it is also applicable for IM.

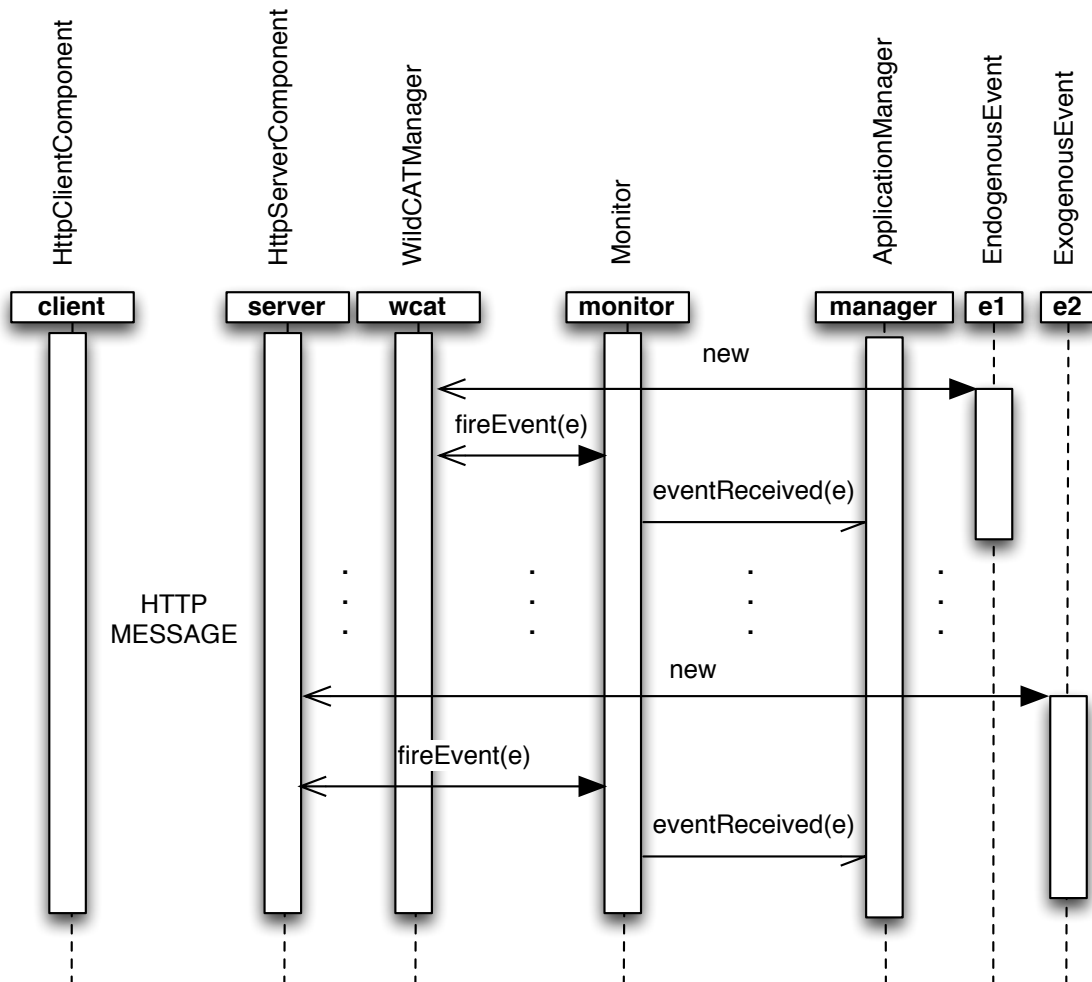


Figure 6.23: Autonomic Events Creation Over the Time.

Figure 6.24 depicts the process of action creation and execution over the time. Autonomic actions in the AM can be performed either on the managed system or on the IM (interloop). Both kinds of actions are executed by an executor (object of the class *Executor*), which is instantiated by the manager (AM). Actions on the managed system relies on an object of the class *FraSCAtiAPIManager* to perform the action, whereas interloop actions relies on an object of class *HttpClientComponent* to send an HTTP message to the IM. This message is received by an object of the class *HttpServerComponent* on the other side (IM).

By replacing the class *FraSCAtiAPIManager* by the class *XenSSHManager*, the same sequence could also be applied to IM objects.

For the token acquisition there might be two different possible implementations: (i) a blocking one, that is, the AMs stays blocked waiting for the token acquisition; and a (ii) non-blocking one, meaning that every time a token is not possible the AM gives up on it so there is no waiting time. The first approach has all the promotions as a big critical section, which means that an AM with a token has access to all the promotions and can freely choose which ones to take. Meanwhile the

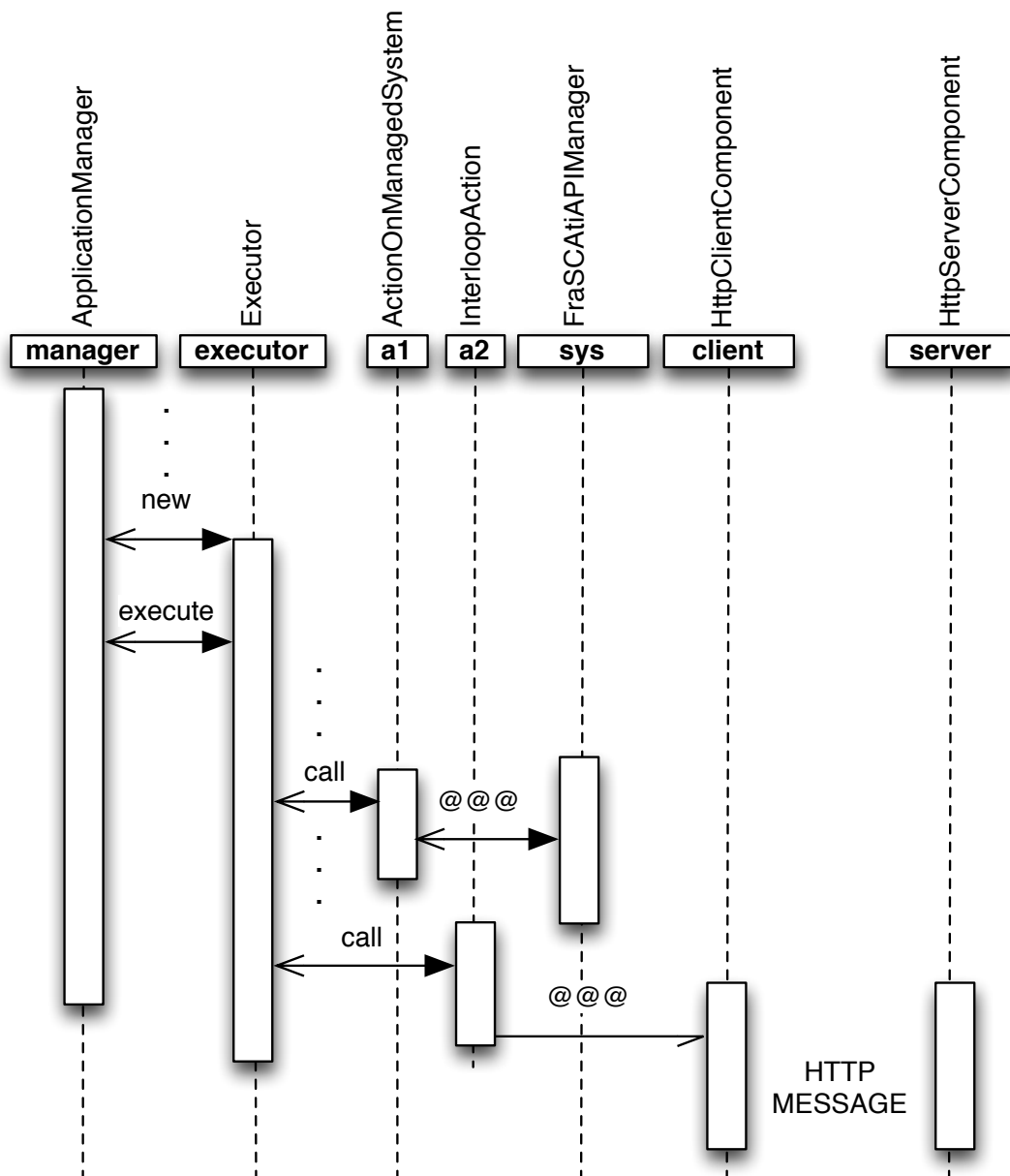


Figure 6.24: Autonomic Actions creation over the time.

other AMs wait for their turn. While it may optimize the use of promotions, it may, on the other hand, impact the performance of the global system as the number of AMs increase (cf. Section 7.2.1). The second approach considers that there is one critical section per available promotion, which means that there is one token per available promotion. In order to avoid deadlock scenarios, AMs should not stay blocked until it succeeds in acquiring a token. Indeed, it may happen that two AMs succeed in acquiring one token each one and stays blocked waiting for each other to release the remaining tokens. In this implementation, we considered the second implementation, although the impacts of the first one is evaluated in Section 7.2.1.

Figure 6.25 shows how the tokens are acquired and released over the time. In the prototype, the public knowledge is implemented as an object of the class *InfrastructureManagerPublicKnowledge*, which is maintained by the IM and can be accessed from outside via an HTTP server (object of class *HttpServerComponent*). The AM willing to acquire a token for a given promotion should rely on an HTTP client (an object of class *HttpClientComponent*) to send an HTTP get message

that is received by the HTTP server on the IM side. The IM finds the desired promotion and tries to acquire the token. If the token is not available, it returns an error message to the AM so it can try again later. Similarly, in order to release a token, the AM relies on the HTTP client to send an HTTP put message which is received by the HTTP server on the IM side. It finds the concerned promotion in the public knowledge and releases the token.

The token transfer operation, as described in Section 5.1, is implemented along with the VM requested event scenario. In fact, promotions are requested along with the interloop actions/events *Request Virtual Machines / Virtual Machine Requested* and are accepted upon the verification (on the IM side) of the tokens transmitted inside the interloop action/event message.

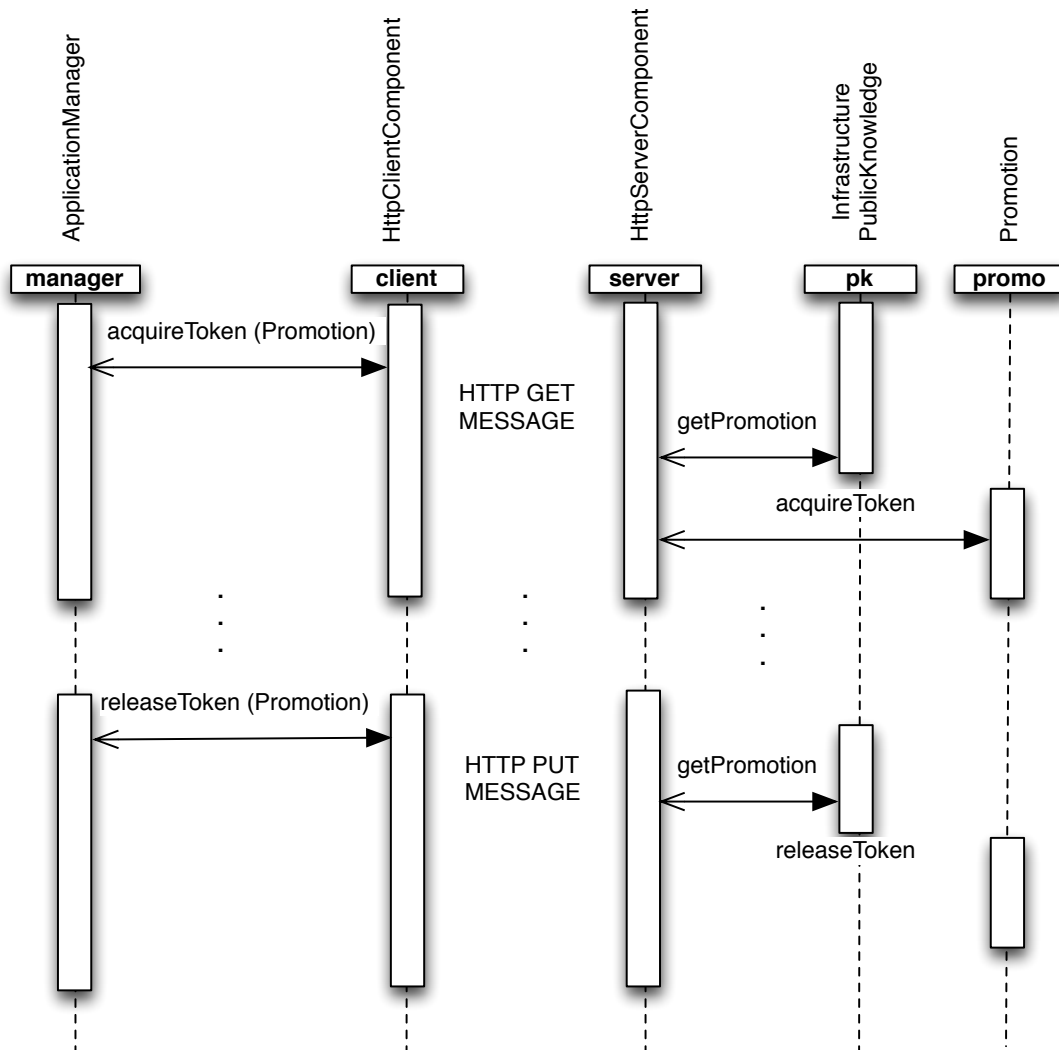


Figure 6.25: Token Acquisition / Release Over the Time.

The autonomic managers extending the class *AutonomicManager* should implement the abstract method `init()` which is the autonomic manager's bootstrap method. The IM does not need any special task to be performed during the initialization. The AM, on the contrary, needs to subscribe to the IM public knowledge so as to be notified when promotions are created. It is necessary also to get the renting fees and the existing promotions from the IM public knowledge. Moreover, it has to deploy the managed application on the infrastructure. Figure 6.26 describes the bootstrap process of the AM. As it can be seen, the execution of the method `init()` triggers the public knowledge subscription and a request for the renting fees, which is performed by an object of the class *HttpClientComponent*. This object sends an HTTP get message that is received

by an object of the class *HttpServerComponent* on the IM side. This object proceeds with the subscription and gathers the renting fees and available promotions, and encapsulates them in the response to the HTTP get message. The AM updates the renting fees and available promotions values in the knowledge (object of the class *ApplicationManagerKnowledge*). Next, it creates an object of the class *WorkloadIncreased* by setting the value of attribute *workload* to the minimum workload possible in the performance requirements (class *PerformanceRequirement*). At last, the AM notifies itself of the creation of this event. As a consequence, the AM starts an analysis to find the optimal configuration and amount of resources suitable for the current workload. This process is detailed in the next paragraph.

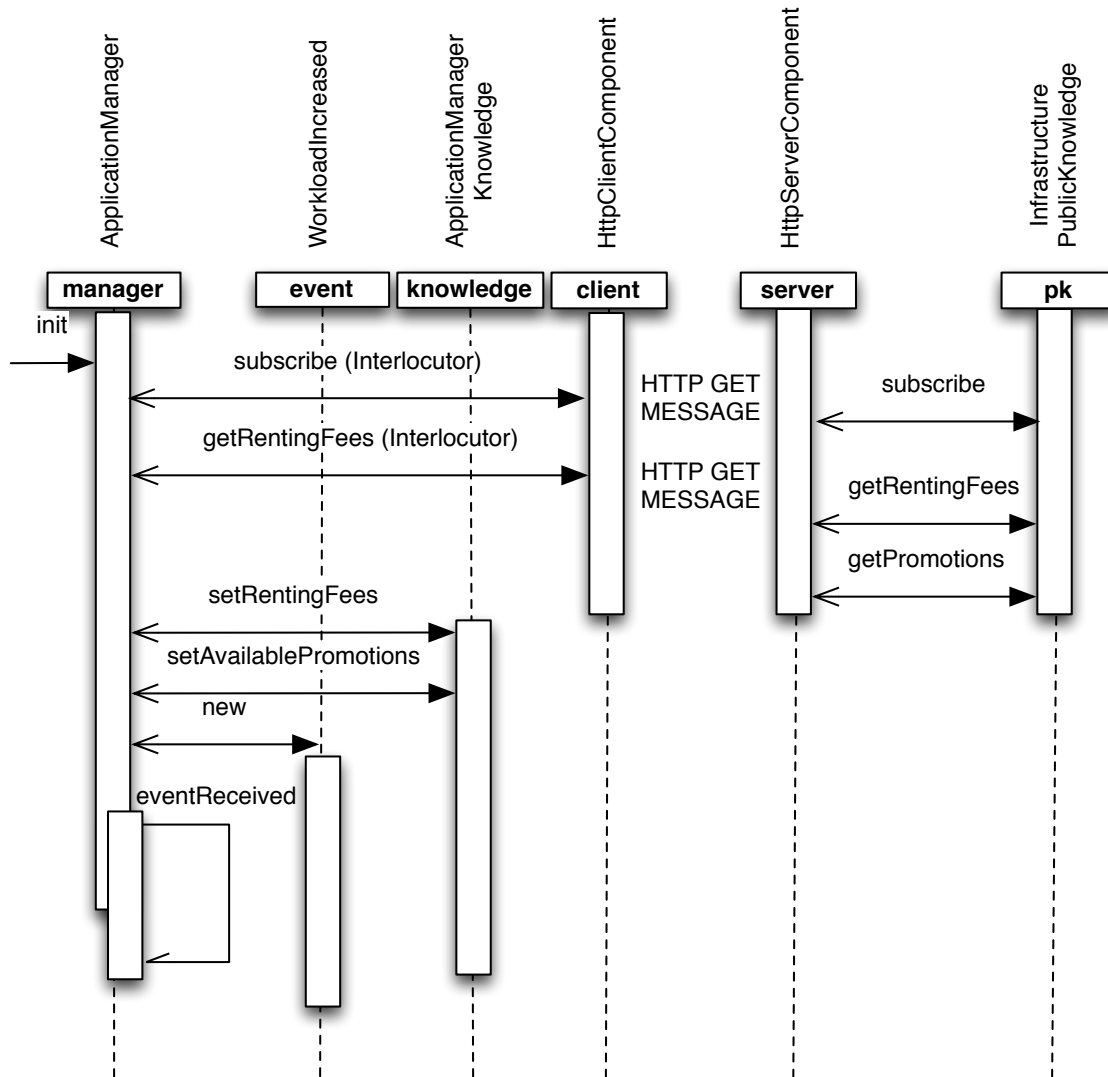


Figure 6.26: Application Manager Initialization Procedure.

Figure 6.27 depicts the behavior of the AM over the time upon the detection of an autonomic event. It should be noticed that the AM acts differently according to the event. The scenario presented in Figure 6.27 corresponds to the detection of an *Workload Increased* event. However, it could be also applicable for the events *Workload Decreased*, *Renting Fees Changed* or *Promotion Expired*.

Once the event is detected, the first thing the AM does is to try to acquire the token of all available promotions, so they can be taken into account in the CSOP. The promotions that the AM failed in acquiring the token are ignored. So, an object of the class *ArchitectureCapacityAnalyzer* is created. Inside the method *analyze()*, the analyzer creates an object of the class *Architecture-*

CapacityConstraintSolver to solve the CSOP. After the analysis task, the IM releases the token of the promotions that are not going to be used. The next step is to create a planner (an object of the class *ArchitectureCapacityPlanner*) and to generate a plan to be executed by an executor (object of class *Executor*). The plan is composed of a set of actions that should be executed in a specific order, as described in Section 5.2.3. So, if there is more VMs to be requested to the IM, the action implemented by the class *RequestVirtualMachines* should be executed first and the rest of actions should be encompassed by a handler (interface *Handler*) which is scheduled on the knowledge (object of the class *ApplicationManagerKnowledge*) to be executed afterwards. Upon the detection of an event of class *VirtualMachinesCreated*, the scheduled handler is retrieved and executed.

The diagram shown in Figure 6.27 assumes that new VMs will be requested to the IM and that is why there is a need for the handler. However, if no new VMs are needed or even if some VMs needs to be released (which is the case also of a scale down scenario), all the actions in the plan could be executed in a roll and there would be no need for a handler.

Due to readability reasons, we have omitted several objects that should be created in a real usage. For instance, objects of classes *Plan* and its child classes, and *ArchitectureCapacityAnalyzerResult*. In fact, the objective of the diagram presented in Figure 6.27 is to give a high level description of how some of the main objects created during the execution of the AM are used. Therefore, the lack of these objects in the diagram should not interfere with the overall understanding.

Figure 6.28 describes how the IM interacts with other objects over the time upon the arrival of a *Virtual Machines Requested* event (object of the class *VirtualMachinesRequested*). Assuming that the AM requested some promotions among other VMs with regular price. So, the first thing the IM does is to get the promotions from the public knowledge (object of the class *InfrastructurePublicKnowledge*) and checks for each promotion requested if the token sent corresponds to the actual promotion token. Then, it creates an analyzer (object of the class *VirtualMachinePlacementAnalyzer*). Inside the method *analyze()*, the analyzer creates a solver to solve the CSOP. After the analysis task, the IM creates a planner (object of the class *VirtualMachinePlacementPlanner*) that generates a plan for the solution found by the analyzer. The IM then creates an executor (object of the class *Executor*) and executes the plan generated by the planner. Among other things, the result is a set of actions on the managed system (more precisely *CreateVirtualMachine*), performed by an object of the class *XenSSHManager* and one interloop action to notify the AM of the VMs creation.

Likewise the diagram of Figure 6.27, we have omitted some objects (e.g. of the classes *Plan* and its child classes, and *VirtualMachinesPlacementAnalysisResult*) for readability reason. As the objective is to explain the interactions between the IM and some of the main objects over the time, the missing objects do not interfere with the global comprehension.

6.5 Summary

This chapter presented the implementation details of concepts and models of the multiple autonomic manager approach proposed in Chapter 5. The research prototype conceived was implemented in Java and consists of two sub-systems: the application manager (AM) and the infrastructure manager (IM).

First, we described the common autonomic framework, which is a set of Java classes that are used by both the AM and IM. Then, we detailed the implementation of both AM and IM by presenting the technical details on the managed systems (application and infrastructure, respectively) and the meta-models for each autonomic tasks (i.e. monitoring, analysis, planning and execution). The meta-models show how the concepts and models presented in the previous chapters are implemented in Java classes.

We assume that managed applications are developed in a component-based architectural manner. Hence, a Service Component Architecture model implementation, namely FraSCAti, was

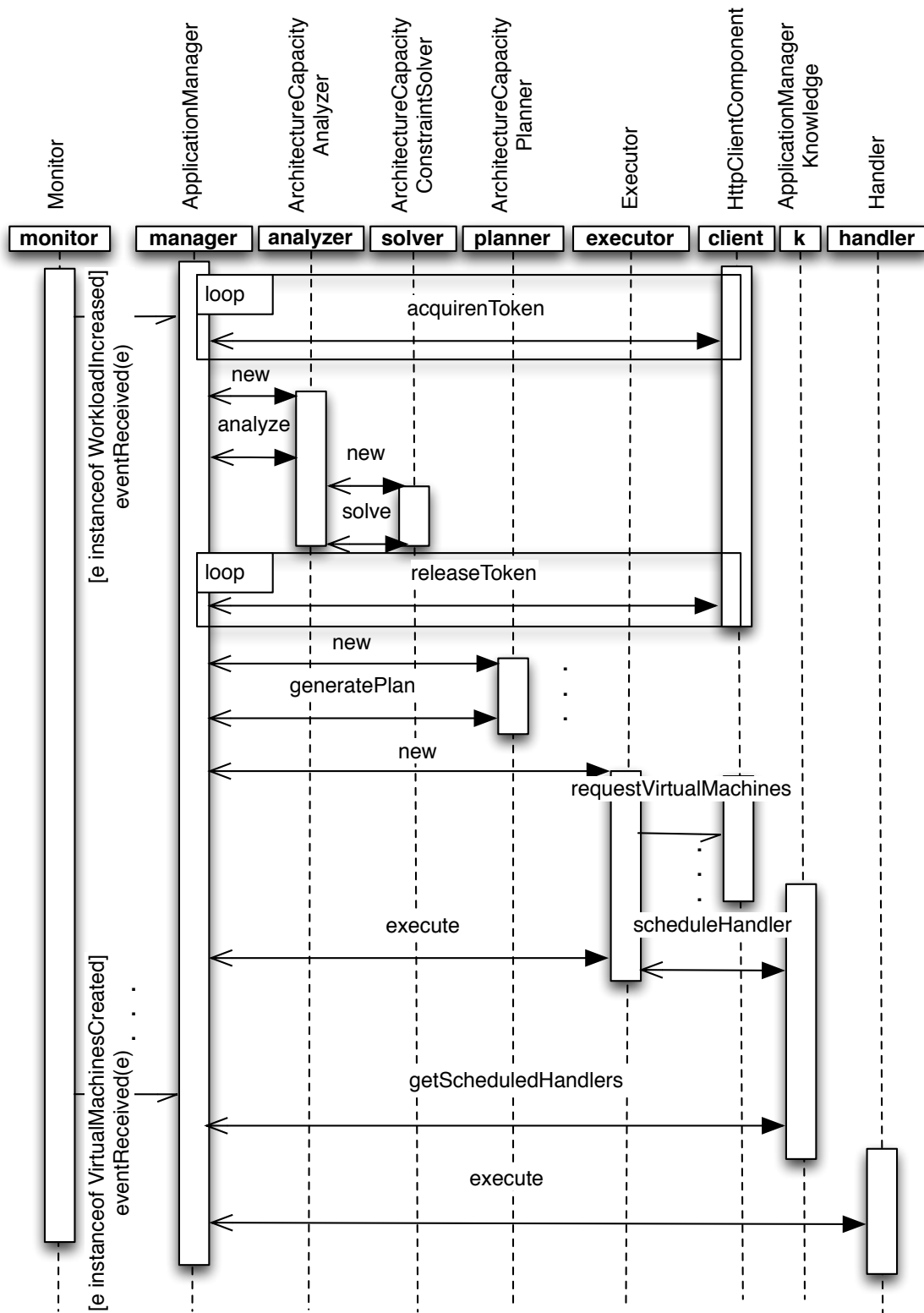


Figure 6.27: Application Manager Interactions Over the Time.

used to ease the deployment and reconfiguration of application components. In order to enable scalability, components can be replicated in several instances which are managed by Nginx, a lightweight load balancing software. Finally, in order to facilitate the monitoring of the managed systems, the AM relies on WildCAT, a context-aware framework. Regarding the infrastructure, we relied on Xen hypervisor to manage the creation/destruction of virtual machines (VM), which can

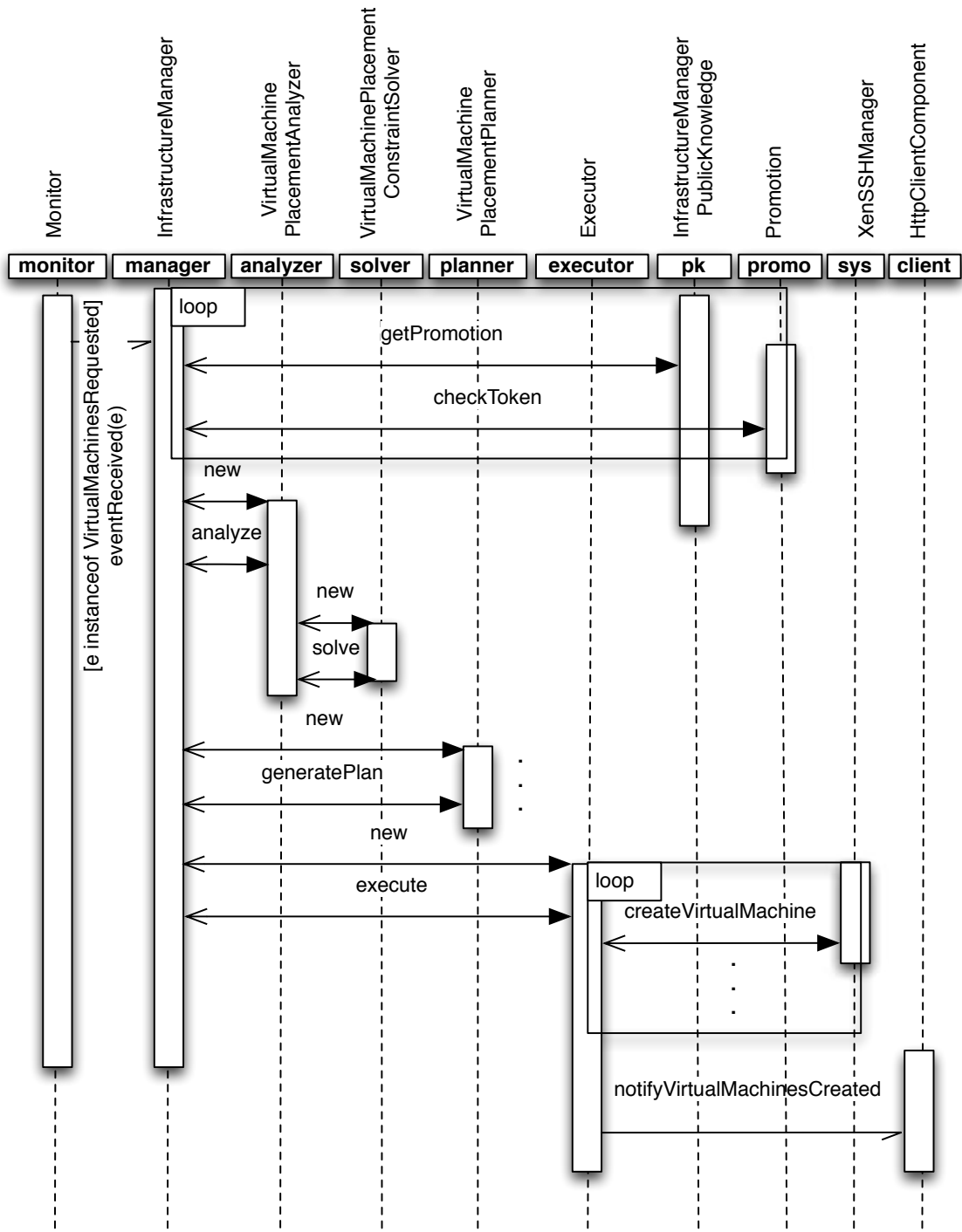


Figure 6.28: Infrastructure Manager Interactions Over the Time.

be done remotely via SSH. The IM also relies on WildCAT to ease the monitoring of the managed infrastructure.

At last, we depict the interactions among autonomic managers' objects over the time. The idea was to show the interactions between objects within the same autonomic manager as well as between objects from different autonomic managers in some of the main scenarios of execution.

Experiments and Evaluation

This chapter presents the a set of experiments performed over the research prototype presented in Chapter 6 with the purpose to validate this thesis work.

First, we present a qualitative evaluation analysis, whose objective is to evaluate the effectiveness of the approach via a set of experimental scenarios. To this end, we relied on a real experimental grid infrastructure to play the role of the infrastructure and a service-based application that was implemented to serve as the application.

Finally, we present a quantitative evaluation analysis, which aims at evaluating the critical parts of the proposed solutions such as algorithms and protocols in an simulation-based way.

Contents

7.1	Qualitative Evaluation	131
7.1.1	Experimental Testbed	132
7.1.2	Budget-driven Architectural Elasticity Scenario	134
7.1.3	Scale Down Scenario	137
7.1.4	Promotion Scenario - With Architectural Elasticity	139
7.1.5	Promotion Scenario - Without Architectural Elasticity	143
7.1.6	Discussion	147
7.2	Quantitative Evaluation	148
7.2.1	Synchronization and Coordination Protocols	149
7.2.2	Results	150
7.2.3	Optimization Problems	153
7.3	Summary	157

7.1 Qualitative Evaluation

This section presents the qualitative experiments carried out to evaluate this thesis work. The objective of this evaluation is to show the effectiveness of the proposed approach in improving the Quality of Service (QoS), at the applications level, and energy consumption, at the infrastructure level. First we describe the experimental testbed, which corresponds to a service/component-based application that was implemented specifically for the evaluation, a real physical infrastructure upon which the experiments took place and a client simulation framework. Then we present the results obtained from the three scenarios executed to evaluate the approach: (i) a budget-driven

architectural elasticity, in which applications benefit of the architectural elasticity to cope with an increasing workload, while meeting the budget constraints; (ii) a scale down scenario, in which applications also benefit of the architectural elasticity to cope with a period of resource restriction caused by an energy shortage at the infrastructure level; and (iii) a promotion scenario in which the infrastructure provider tries to establish a synergy with application providers by stimulating their behaviour in terms of consumption. It results sometimes in an improvement of the resource utilization and consequently the energy efficiency. Finally, we provide some discussion on the obtained results.

7.1.1 Experimental Testbed

The qualitative experiments relies on a testbed environment composed of a synthetic application and a real physical infrastructure in which one or several applications instances are deployed. By synthetic we mean that the application was implemented especially with the purpose to evaluate this thesis work. We first describe this application, by detailing its characteristics in terms of composition and performance requirements. Then we survey the physical infrastructure platform used to run the experiments.

Synthetic Application

We developed a Service Component Architecture (SCA) based application, in which the components were implemented in Java 6. As SCA runtime, we relied on FraSCAti [SMF⁺09], since it allows architectural reflection, which enables dynamic reconfiguration of the applications described in FraSCAti.

Components and Architectural Configurations. As depicted in Figure 7.1, the application is composed of three components $C = \{c_1, c_2, c_3\}$, where $c_1^r = \{r_1\}$, $c_1^s = \{s_1\}$, which means that component c_1 offers a service s_1 , has a functional dependency r_1 . Regarding the others components, $c_2^r = \emptyset$ and $c_2^s = \{s_2\}$. Similarly, $c_3^r = \emptyset$ and $c_3^s = \{s_3\}$. It means that components c_2 and c_3 offer each one a service (s_2 and s_3 , respectively), whereas they do not have any functional dependency. The application has two architectural configurations $K = \{k_1, k_2\}$, where $k_1^c = \{c_1, c_2\}$, $k_1^{deg} = 0$, $k_2^c = \{c_1, c_3\}$, $k_2^{deg} = 0.50$, $k_1^{bind} = \{(r_1, s_2)\}$, $k_2^{bind} = \{(r_1, s_3)\}$ meaning that configuration k_1 has no architectural degradation, components c_1 and c_2 are bound via reference r_1 and service s_2 , whereas in configuration k_2 there is a degradation of 50% and components c_1 and c_3 are bound via reference r_1 and service s_3 . This degradation percentage value was chosen since it had to be significant enough to, when multiplied by its respective weight (α_{arch}), prevent the solver tending to choose always a degraded configuration so as to consume less resources while reducing the costs regarding the renting fees. To make an analogy with the motivation scenario presented in Section 4.3.3, the configuration k_1 would be equivalent to a configuration with a Video Advertisement Server, whereas the configuration k_2 would correspond to a configuration with a Image or Text Advertisement Server component.

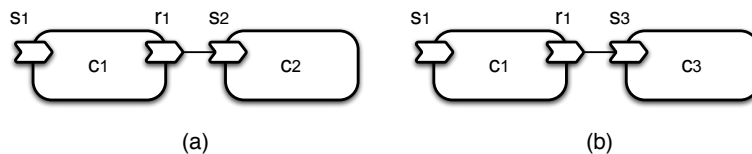


Figure 7.1: Synthetic Application Architectural Configurations: (a) k_1 and (b) k_2 .

Performance Profile. Each component performs some CPU intensive operation, more precisely they execute a function to find the *fibonacci* number for a given input. In order to emulate

Component	Workload	Response Time (ms)	CPU Cores	RAM (GBs)
c_1/c_2	30	200	•	•
		600	•	•
	55	300	••	••
		100	•••	•••
		900	•	•
	80	550	••	••
		300	•••	•••
		100	••••	••••
		1200	•	•
	170	700	••	••
		450	•••	•••
		300	••••	••••
		100	•••••	•••••
		100	•••••	•••••
c_3	30	100	•	•
	55	200	•	•
	80	250	•	•
		50	••	••
	170	350	•	•
		100	••	••

Table 7.1: Synthetic Application Resource Requirements.

two different levels of resource consumption we pseudo-randomly choose a uniformly distributed input value for that function varying from 20 to 40 for components c_1 and c_2 ; and from 10 to 20 for component c_3 . The components were calibrated in order to profile the application’s performance requirements. Table 7.1 shows the requirements of each component in terms of resources in order to ensure a certain level of QoS under a given workload. This calibration was performed in a specific hardware of the testbed physical infrastructure (cf. Section 7.1.1). That way, it is straightforward that components c_1 and c_2 are more resource consuming than component c_3 . As a consequence, although the imposed degradation, the architectural configuration k_2 can be an alternative so as to consume less resources, when necessary.

Based on the performance profile in Table 7.1, we define the set of workload threshold $\Lambda = \{0, 30, 55, 80, 170\}$. The thresholds are used to detect workload variation events, which may lead to reconfigurations at the application level.

Optimization Variables Weighting. We fixed the weights for each optimization variable as follows: $\alpha_{perf} = 0.6$, $\alpha_{arch} = 0.3$ and $\alpha_{cost} = 0.1$. These values were chosen such that $\alpha_{perf} > (\alpha_{arch} + \alpha_{cost})$, that is, the performance criterion is always privileged with respect to the others. Also, the *cost* criterion should never be privileged in comparison to the other criteria. That way, it is certain that the analysis tries first to find a configuration without architectural degradation, in which the response time is inferior to the thresholds (rt^{ideal} and rt^{acc}), and minimizes the cost. If it is not possible, then it seeks for a solution in which the architecture is degraded. Also, it prevents the solver engine to find a solution that does not respect the performance thresholds (rt^{ideal} and rt^{acc}) or degrades the architecture configuration in order to minimize the costs.

Physical Infrastructure

Grid’5000. For the physical infrastructure, we relied on Grid’5000, which is a French grid for experimental testbed. It is composed of ten geographically distributed sites: Bordeaux, Grenoble, Lille, Luxembourg, Lyon, Nancy, Reims, Rennes, Sophia and Toulouse. Each site comprises

one or more clusters, which are characterized by a set of physical machines (PMs) with similar characteristics in terms of resources (e.g. CPU, memory and disk). Some of the Grid's 5000 clusters are equipped with devices for power measurements. Reims, Nancy and Rennes sites are equipped with power distribution unit (PDU) with power monitoring capabilities. This kind of monitoring is not very precise, since the information collected is shared by several PMs, which may interfere in the results. The site of Lyon, on the other hand, is the only one which is equipped with more precise and per-PM wattmeters¹. The site of Lyon is composed of only one cluster, namely *sagittaire*². This cluster is composed of 79 PMs Sun Fire V20z, which have single CPU/dual core (AMD Opteron 250) with 2048MB of DRAM and 65GB of storage space.

Infrastructure Settings. For these experiments, the set of PMs P of our infrastructure consists of a sub-set of the PMs of *sagittaire*. Hence, for each $pm_i \in P$ $pm_i^{cpu} = 2$ and $pm_i^{ram} = 2$. We do not precise the exact number of PMs, since it may vary from one scenario to another. Thus, it is specified in the experimental setup of each scenario. We also defined two classes of virtual machines (VMs) (e.g. small and large) $M = \{m_1, m_2\}$, where $m_1^{cpu} = 1$, $m_1^{ram} = 1$, $m_1^{cost} = 0.18$, $m_2^{cpu} = 2$, $m_2^{ram} = 2$ and $m_2^{cost} = 0.27$. The pricing values correspond to the hourly use cost for small/medium and large VM instance, respectively. Notice that the cost of a large VM instance is not the double of that of the small/medium. Instead, we reduce a little bit its price (per CPU/RAM unit) in order to privilege the leasing of large instances.

Client Simulation and Managers

Clif. In order to simulate clients so as to produce a real incoming charge, we relied on Clif³, a load testing framework. With Clif, it is possible to define experimental scenarios in terms of client behaviour and population over the time. We define the behavior of a client as a request followed by a thinking time such that one client can perform only one request per second. That way, the desired number of requests per second is achieved by increasing the population of simultaneous clients. As a result of a scenario execution, Clif provides log files from which one can extract important information about the performance of the system under test. The system architecture basically consists of one registry server and one or several code server. In our experiments, we deployed a code server for each application to be evaluated. In order to avoid interferences, we deployed each server and the registry in a separate VM, with one CPU core and 1GB of DRAM capacity.

Application and Infrastructure Managers. Similarly, in order to avoid interference, we reserved a VM with one CPU core and 1GB of DRAM capacity for each application manager (AM) or infrastructure manager (IM).

7.1.2 Budget-driven Architectural Elasticity Scenario

This scenario consists in deploying one application and evaluate how it responds to an increasing workload while having its budget constrained. The objective is to show how the AM can benefit of the architectural elasticity under a constrained situation.

Experimental Setup

For this scenario we deployed one application and performed several runs while changing the budget threshold. Each run took about one hour long. In addition, we also enabled / disabled

¹Some nodes of Toulouse are also equipped with wattmeters, but by the time the experiments were performed, these wattmeters were unavailable.

²There is another cluster in the site of Lyon, but by the time these experiments were carried out the cluster was inoperative.

³<http://clif.ow2.org/>

Run	Arch. Configurations	Budget
1	$\{k_1, k_2\}$	0.7
2	$\{k_1\}$	0.7
3	$\{k_1\}$	0.9
4	$\{k_1\}$	1.0

Table 7.2: Application Settings for the Budget-driven Architectural Elasticity Scenario.

the alternative architectural configuration (k_2 in the case of the application described in Section 7.1.1) so as to evaluate the impact of the architectural elasticity. More precisely, we performed four different runs for the same application while changing its settings: (i) with budget fixed at 0.7 and with the two architectural configurations available; (ii) with the budget fixed at 0.7 and with only one architectural configuration available; (iii) with the budget fixed at 0.9 and only one architectural configuration available; and (iv) with the budget fixed at 1 and only one architectural configuration available. These settings are summarized in Table 7.2.

Figure 7.2 presents the application workload defined for this scenario. As it is shown, we defined an increasing workload (from 0 to 95 requests per second) so as to observe how the application copes with the workload variation by performing either resource capacity elasticity or architectural elasticity according to the budget variation.

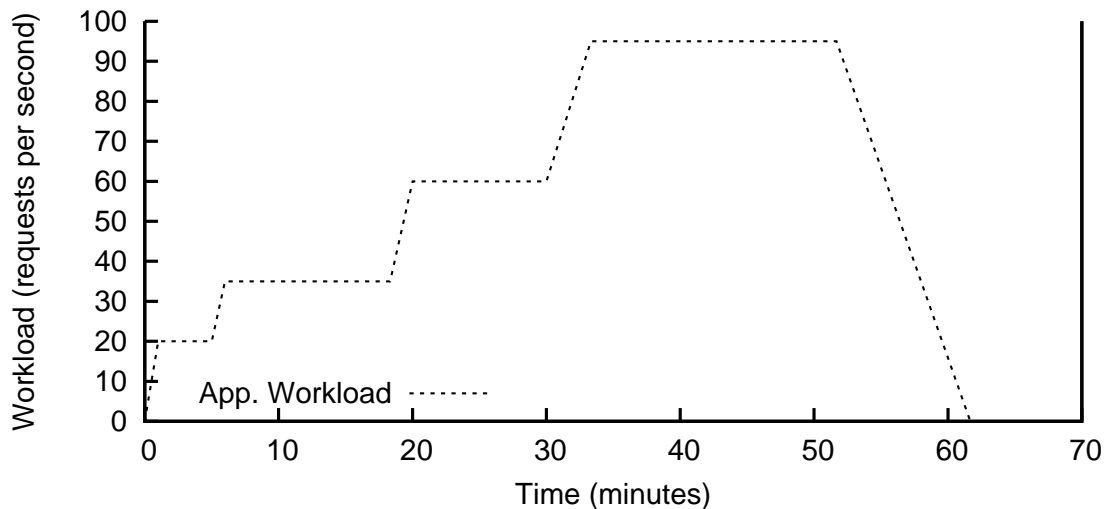


Figure 7.2: Workload for the Budget-driven Architectural Elasticity Scenario.

Lastly, with respect to the infrastructure, we utilized a sub-set of ten PMs from cluster *sagittaire*. Two PMs to host the managers' and Clif servers' VMs and eight that were used to host the VMs containing the application components.

Results

Figure 7.3 shows the average response time (y-axis) of the application described in Section 7.1.1 under a certain workload (dashed line) over the time (x-axis). As it can be seen, as long as the application does not reach the budget threshold, it is capable of coping with an increasing workload by requesting more VMs to the infrastructure provider. That is, the higher budget the lower is the response time (i.e. the QoS). It should be observed, however, that thanks to the architectural elasticity (both configurations k_1 and k_2 enabled, and referred as AE in the chart), the application tuned with the lowest budget (0.7) manages to keep the response time at an acceptable level (approximately the same response time as it had 0.9 of budget). The switching between architec-

tural configurations can be seen in Figure 7.4. By the time of 10 minutes of execution, the AM detects a workload increased event, which is triggered because the current workload exceeds the upper threshold (30). With a budget greater or equal to 0.9, the AM manages to request more VMs to cope with the workload. For a budget constrained at 0.7, the application in the current configuration is not able to maintain the acceptable level of response time (800ms). Alternatively, the application may degrade its architecture so as to consume less resource and thereby meet the budget constraints while coping with the workload. It keeps the ideal response time (500ms) until 30 minutes of execution. By that time, the workload sharply increases, although the response time stays below the acceptable level (800ms).

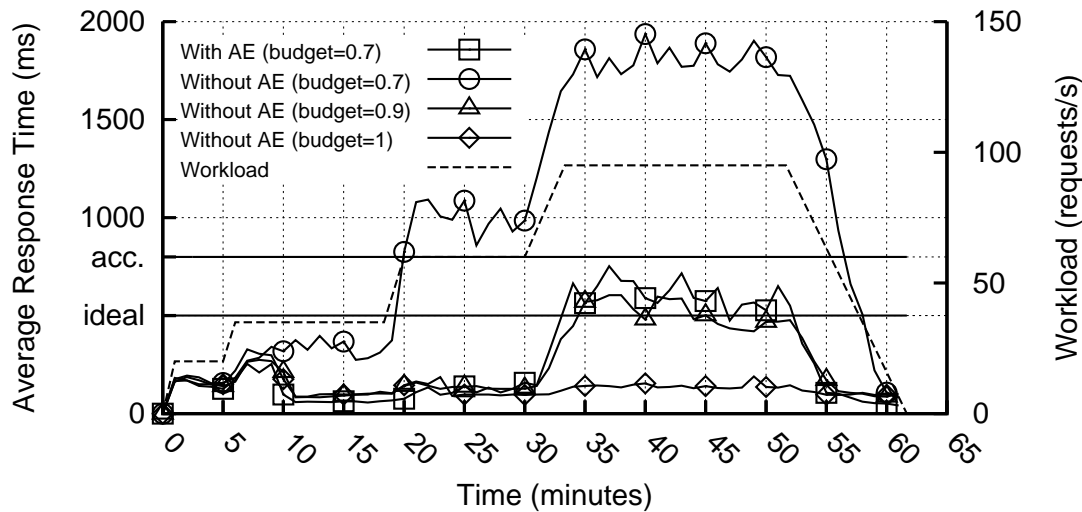


Figure 7.3: Average Response Time for the Budget-driven Architectural Elasticity Scenario.

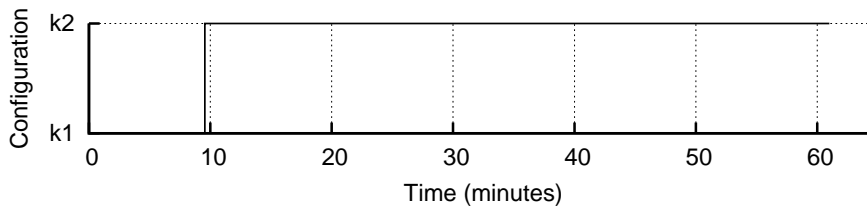


Figure 7.4: Switch Between Architectural Configurations.

Although the architectural degradation allows the application to cope with imposed constraints on budget or resources, this flexibility comes at a price. Indeed, the architectural degradation corresponds to a lower level of a specific QoS or functionality, which may lead to financial penalties according to the Service Level Agreements (SLAs) between application providers and their consumers⁴. It is then necessary to evaluate this degradation along with the actual monitored response time and the cost implied by the VMs leasing.

For this purpose, Figure 7.5 shows the ratio of the normalized values (between 0 and 1) of QoS to the cost due to VMs leased to the application operating with and without architectural elasticity. The QoS is given by the arithmetic mean of the normalized measured response time and the complement of the current architectural degradation ($1 - k_i^{deg}$). It is noteworthy that we considered the measured response time instead of the estimated one that is used in the optimization

⁴As already stated, the management of SLA and more precisely between application providers and end users is out the scope of this work.

problems (cf. Section 5.2.3, Equation 5.31). Hence, the value of the normalized response time is equal to 1 if the measured response time is lower or equal than 500 (rt^{ideal}), 0.5 (rt^{deg}) if it is between 500 and 800 (rt^{acc}), and 0 otherwise. The cost, in turn, is given by the ratio of the current cost due to VMs rental to the application budget (cf. Equation 5.23). Equation 7.1 summarizes the y-axis function for a given measured response time (x), current architectural configuration (y) and the total cost (z). α and β are weights valuing between 0 and 1. In this case, since it is an arithmetic mean the values for these variables are determined as follows $\alpha = \beta = 0.5$.

$$f(x, y, z) = \frac{\alpha * (1 - deg^{perf}(x)) + \beta * (1 - y^{deg})}{\frac{z}{budget}} \quad (7.1)$$

As shown in Figure 7.5, the interval from the beginning of the experiments to 10 minutes of execution the ratio is similar, since there is no difference in terms of architectural configuration between the two runs. By the time of 10 minutes, we can observe a difference between the two curves (with and without architecture elasticity (AE in the chart)), which can be explained by the switching between architectural configuration in the application with architectural elasticity enabled (cf. Figure 7.4). This can be explained by the difference between response time degradations in comparison to the architectural degradations. Moreover, when the application runs with architectural configuration k_2 (degraded mode) it consumes less resources with respect to configuration k_1 , which reduces the cost and thus increases the ratio. Figures 7.6(a) and 7.6(b) show the effects taken by varying the values of α and β (weighted average) for the response time and architectural degradation. As expected, the higher the weight for the architectural QoS (complement of architectural degradation) the lower the ratio, in the case of the run with architectural elasticity in comparison to the run without architectural elasticity.

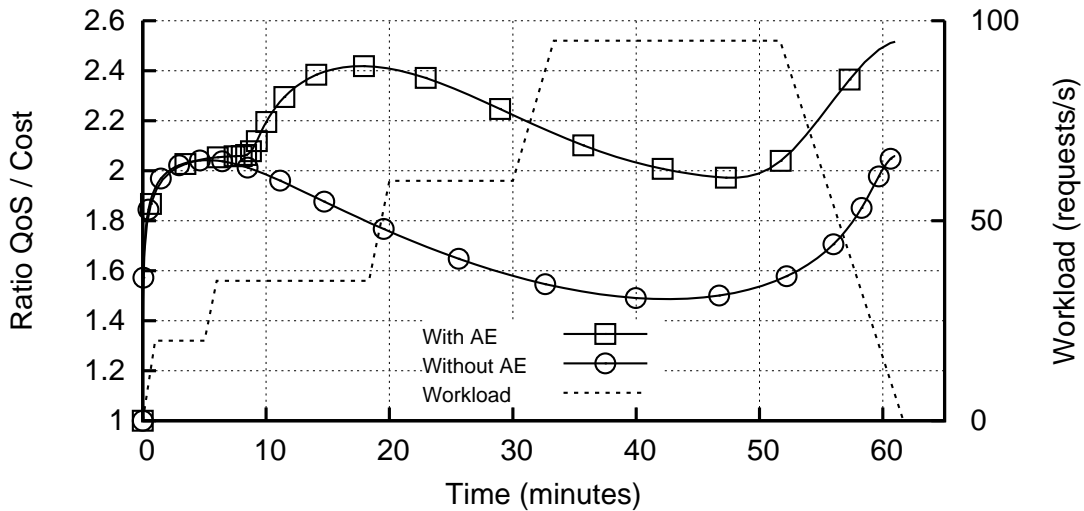


Figure 7.5: Ratio of the Arithmetic Mean of Quality of Service to Cost.

7.1.3 Scale Down Scenario

This scenario consists in triggering an *Energy Shortage* event at the infrastructure level and observe its impacts on the energy consumption as well as on the applications QoS. The objective is to evaluate how applications (thanks to the architectural elasticity) and infrastructure are capable of reacting to energy restriction situations.

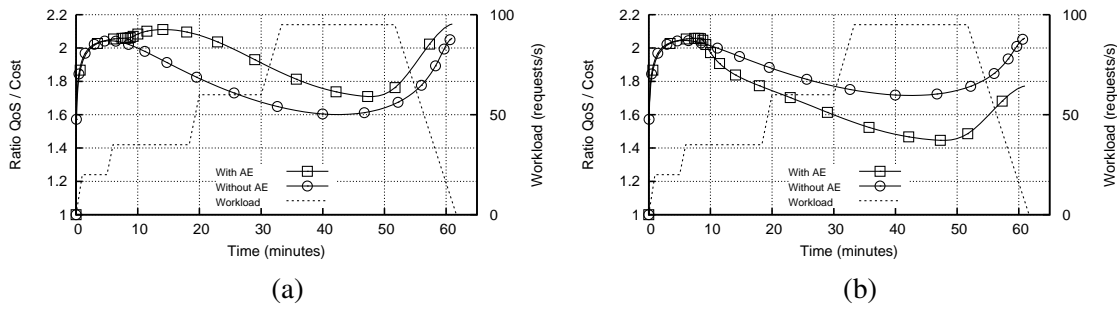


Figure 7.6: Ratio of Application Weighted Average Quality of Service to Cost for (a) $\alpha = 0.4$ and $\beta = 0.6$; and (b) $\alpha = 0.3$ and $\beta = 0.7$.

Instance Name	Arch. Configurations	Budget
<i>App1</i>	$\{k_1, k_2\}$	1.5
<i>App2</i>	$\{k_1\}$	1.5

Table 7.3: Application Settings for the Scale Down Scenario.

Setup

For this scenario we deployed two instances of the application described in Section 7.1.1. For one instance we enabled both architectural configurations (k_1 and k_2) and for the other we enabled only one (k_1). For both instances the budget was fixed at 1.5, as shown in Table 7.3. This value was defined to enable applications to request as much resource as necessary to cope with a workload increase before receiving a scale down event. The duration of the experiment was of one hour, during which the workload increases from 0 to 140 request per second, as depicted in Figure 7.7.

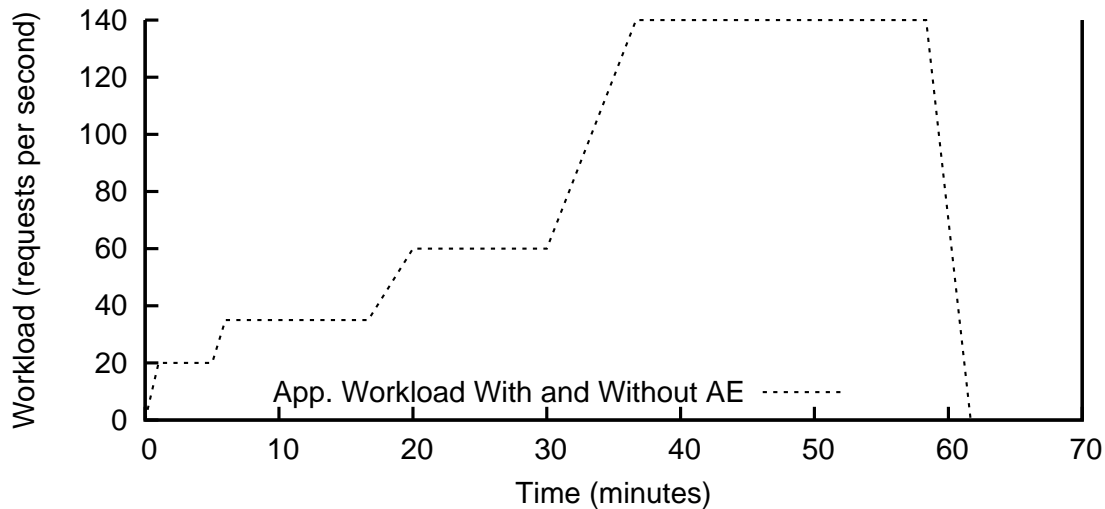


Figure 7.7: Workloads for the Scale Down Scenario.

With respect to the infrastructure, we used a sub-set of 13 PMs of cluster *sagittaire*. Three PMs to host the VMs for the managers' and Clif servers' and ten that were used to host the VMs containing the applications themselves.

We performed several runs by changing the energy shortage degree, that is, the number of PMs to be shutdown: 1, 3 and 5. The *Energy Shortage* event was scheduled to be triggered at 45 minutes of execution for all the three runs. The timeout before the IM shutdown the PMs was fixed to 10 minutes.

The objective is to observe both the reduction in the energy consumption as well as the capability of applications adapting to a scenario of resource shrink. In order to evaluate both applications (with and without architectural elasticity) in equal conditions, we modified the *Energy Shortage Analyzer* (cf. Section 5.2.2) so that both applications had the same amount of resources shrank instead of relying on the resource restriction rate (a_k^{rr}) of each application to determine which PMs should be shutdown and consequently which VMs have to be released. Otherwise, one application could have a bigger reduction rate in comparison to the other and the evaluation would thus be compromised.

Results

Figure 7.8 presents the results of all the three runs. The first run (a), that is, when an *Energy Shortage* event is triggered aiming at shutting down only one PM. More precisely, the chart shows the average response time for the two application instances previously described (with and without architecture elasticity (AE in the chart)). Not surprisingly, it does not affect the applications, because the PM chosen does not host any VM and consequently no application components.

When three PMs have to be shutdown (Figure 7.8(b)), it is already possible to see the impacts on the QoS. In fact, the application with architectural elasticity (AE in the chart) is able to adapt itself in order to support the resource shrink and keep the ideal level of response time. However, for the application without architectural elasticity, it does nothing but releasing the VMs required by the IM to be released. As a result, the response time sharply increases from the moment the *Scale Down* event is detected, even though it stays between the ideal and acceptable levels.

For the shutdown of five PMs (Figure 7.8(c)), on the other hand, the impacts on the applications' QoS are more visible. Upon the detection of a *Scale Down* event, the application without architectural elasticity exceeds by almost 100% the acceptable level of response time, while the application with architectural elasticity remains on the borderline of the acceptable level.

Finally, Figure 7.9 shows the total power consumption of the infrastructure, before and after the *Energy Shortage* event, for all the three runs (1, 3 and 5 PMs shutdown). It should be noticed that the interval between the event detection (vertical line) and the decrease in the power consumption is due to the timeout between the *Scale Down* notification to applications and the actual shutdown of PMs, which corresponds to 10 minutes. With respect to the power consumption, it is straightforward that the more PMs shutdown the lower the power consumption.

To sum up, this scenario is important to show how architectural elasticity can improve the synergy between AMs and IM. As it is shown in Figures 7.9 and 7.8, it allows the IM to seamlessly cope with extreme situation at the infrastructure level (energy constraints) by minimizing the negative side-effects at the application level (QoS).

7.1.4 Promotion Scenario - With Architectural Elasticity

This scenario consists in stimulating the creation of promotions over the infrastructure and observe how they impact the current infrastructure incomes and power consumption. That is to say that the objective is to evaluate how promotions are effective in improving the synergy between application and infrastructure layers.

Setup

For this scenario we deployed four instances of the application described in Section 7.1.1, all of them with both architectural configurations (k_1 and k_2) enabled, as it is shown in Table 7.4. For two of them (*App1* and *App3*) we restrained the budget to 2.0 whereas for *App2* and *App4*, the budget was restrained to 0.8. By doing so, we can force instances *App2* and *App4* to degrade its architecture due to budget restriction (as shown in the budget-driven scenario in Section 7.1.2). This is particularly interesting in this scenario, since those applications, under a certain workload, can only be upgraded (in terms of architecture) if a promotion that fits their budget is made

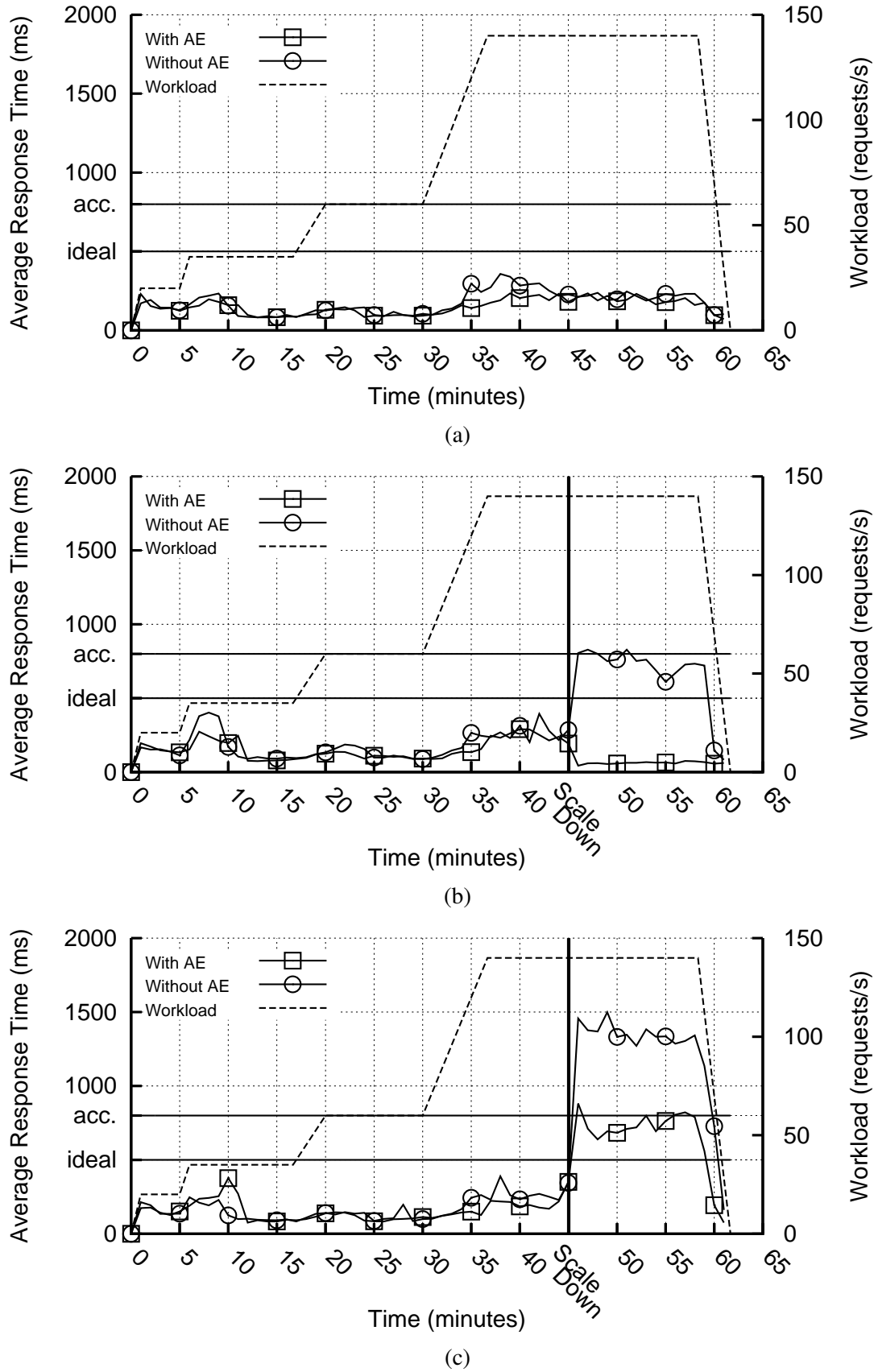
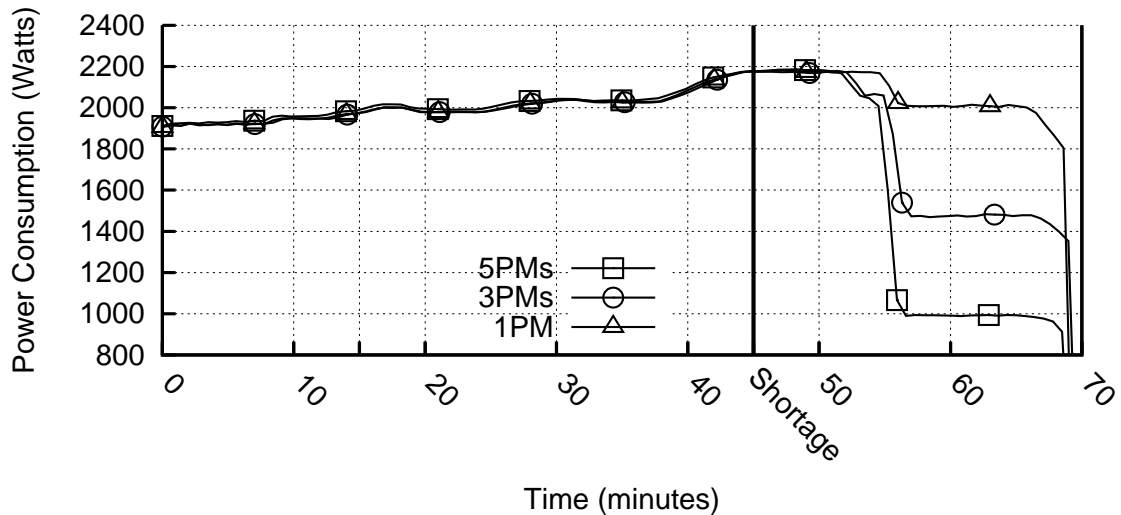


Figure 7.8: The Average Response Time upon a *Scale Down* event for: (a) 1 Physical Machines, (b) 3 Physical Machines and (c) 5 Physical Machines to Shutdown.

Figure 7.9: Power Consumption upon an *Energy Shortage* Event.

available.

Instance Name	Arch. Configurations	Budget
<i>App1</i> and <i>App3</i>	$\{k_1, k_2\}$	2.0
<i>App2</i> and <i>App4</i>	$\{k_1, k_2\}$	0.8

Table 7.4: Application Settings for the Promotion Scenario (with Architectural Elasticity).

As it can be seen in Figure 7.10, we defined a different workload for each application. For *App1* and *App3*, it starts by increasing until it reaches its peak at 100 requests per second. After few minutes, the workload of application *App1* decreases to 65 requests per second, whereas the one of *App3* drops to 40 requests per second. Then, both of them stay like that until the end of the experiment. The workload of applications *App2* and *App4*, in turn, grows up to 35 requests per second and stay constant until the end of the experiment. The intentions behind these variations is to, in the case of applications *App1* and *App3*, initially occupy spots in the infrastructure and then release them in order to allow the creation of promotions. Regarding applications *App2* and *App4* is to make them degrade their architectures forced by the budget constraints. Hence, they can benefit of the promotions to upgrade their architecture as long as the promotions are valid.

With respect to the infrastructure, we used a sub-set of 20 PMs from cluster *sagittaire*. Five PMs to host the VMs for the managers and Clif servers and 15 that were used to host the applications themselves. The condition to activate a promotion is expressed in Equation 5.6, which states that the utilization rate of the given PM should be lower than a threshold U_{min} during t_l units of time. Thus, we fixed U_{min} to 0.6 and t_l to 10 minutes. According to Equation 5.7, a PM is shutdown if the resource utilization rate is equal to zero during at least t_u units of time. We then assigned t_u to 25 minutes. That way, if one PM hosts at least one VM of class vm_1 during at least 15 minutes, a promotion will be created in that PM. If the PM is empty during at least 25 minutes, it will be shutdown. These parameters values allow that both the creation of promotions and shutting down of PMs take place within the experiment time.

The experiment consists in performing several runs by varying some parameters related with the promotion. First, we ran the scenario once with the promotions disabled, then run the scenario several times by varying the discount rate (0.4, 0.5 and 0.6) given by the promotions. These runs are summarized in Table 7.5. For all runs, we ran set up promotions to last 45 minutes. Hence, there is enough time to observe what happens when a promotion is expired.

To sum up, the objective is to evaluate the impacts of the promotions on the infrastructure

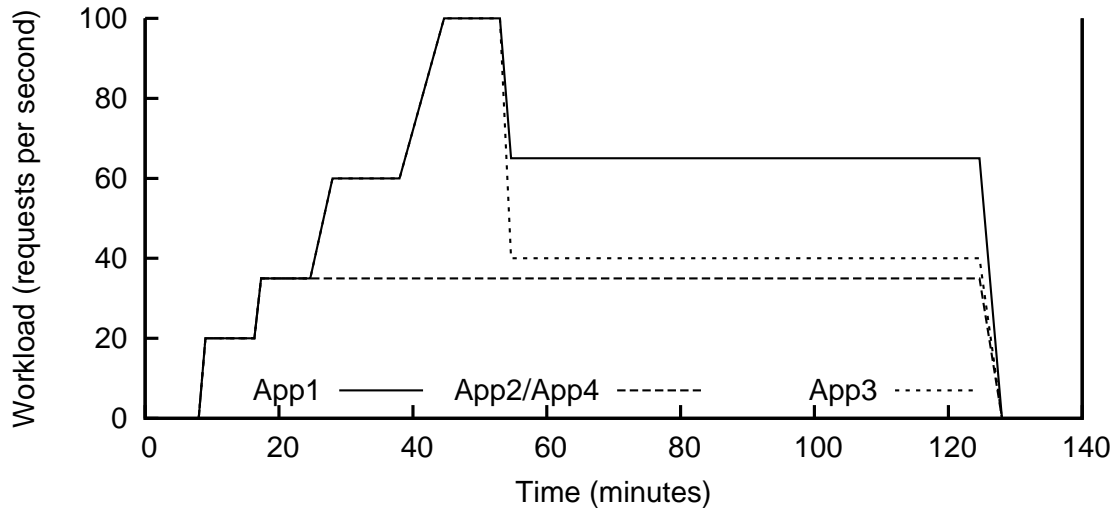


Figure 7.10: Workloads for the Promotion Scenario (with Architectural Elasticity).

Run	Discount (%)	Promotion Duration (minute)
1	w/o	-
2	0.4	45
3	0.5	45
4	0.6	45

Table 7.5: Promotion Settings for the Promotion Scenario (with Architectural Elasticity).

provider incomes and energy consumption, while varying those parameters.

Results

This section describe the results obtained from the execution of the scenario detailed in the previous section.

Figure 7.11 shows the percentage of infrastructure income increase (y-axis) over the time (x-axis). The percentage of income is determined through the ratio of each run with promotion to the income obtained from the run without promotion. For all the runs, until 80 minutes of execution, the incomes do not have any (positive or negative) difference in comparison to the run without promotion. This can be explained by the fact that there is nothing different in the behavior (e.g. promotions created) in all the runs. From 80 minutes on, it can be noticed an increase in the incomes from 4.5% (for the run with discount rate 0.6) to 7% (for the run with discount rate 0.4), which is explained by promotions taken by the applications *App2* and *App4* in order to upgrade their architectural configurations. These promotions are created to fill up the free spots left by applications *App1* and *App3*, that is, after a workload increase (from 0 to 45 minutes of execution) and a workload decrease (from 45 to 55 minutes of execution) and a time of inactivity, promotions are created and thereby taken by applications *App2* and *App4*. It should be also observed the inverse relation between discount rate and income, which is to say that the more discount given the less is the income.

Figure 7.12 shows the total power consumption (y-axis) over the time (x-axis). As all the PMs are previously shutdown, the power consumption is near null at the beginning of the experiment. Similar to the incomes, the power consumption increases (between 0 and 50 minutes) in the same proportion as the applications' workload (Figure 7.10). Also following the workloads, after few minutes, it slightly decreases. Then, for all runs, the power consumption stays constant until 90 minutes of execution and then it decreases again, which can be explained by PMs that

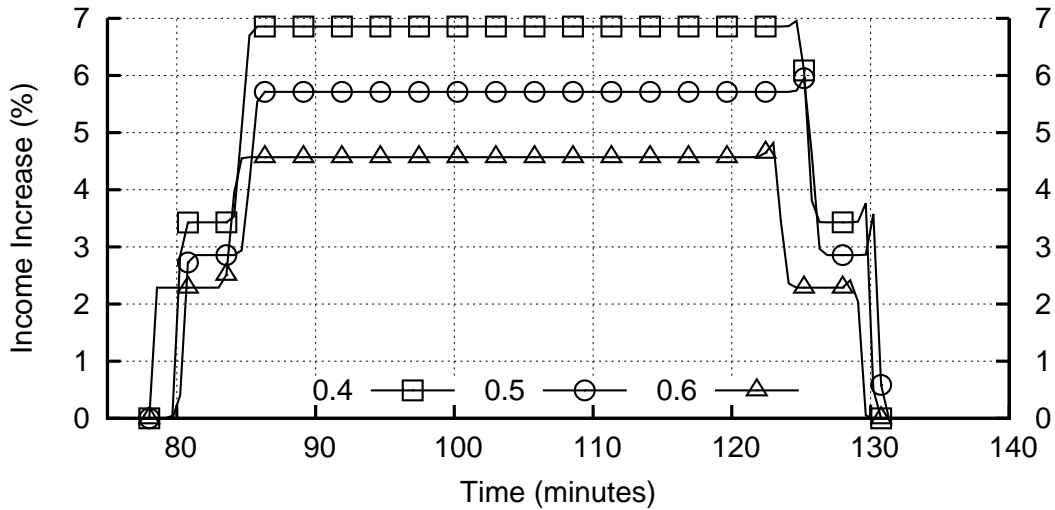


Figure 7.11: Infrastructure Income Increase (%) with Respect to the Promotion-disabled Run (with Architectural Elasticity).

are shutdown due to non-utilization. The power consumption stays constant until the end of the experiment. Therefore, there is no difference of power consumption between runs, since there is no PMs that are shutdown or started in one run which are not in another run. This is particularly interesting because the power consumption stayed almost the same, while the incomes increased. Furthermore, the created promotions allowed applications *app2* and *app4* to upgrade their architectures.

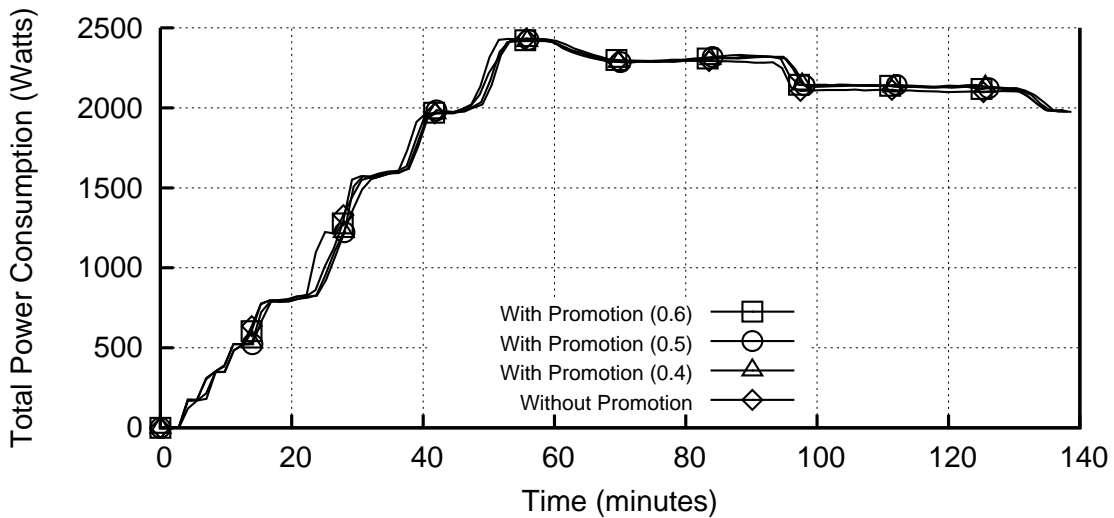


Figure 7.12: Total Infrastructure Power Consumption (with Architectural Elasticity).

7.1.5 Promotion Scenario - Without Architectural Elasticity

The previous scenario is useful to show a situation in which both application and infrastructure providers benefit of the promotion strategies. Indeed, degraded applications have the chance to upgrade their architectures and infrastructure leases more VMs by optimizing the overall infrastructure utilization rate. However, there might be also situation in which no degraded applications are ready to take a promotion. Moreover, there might happen that another application requests

VMs regardless whether the VMs are on promotions or not. As a consequence, the infrastructure provider may waste money by giving promotions. On the other hand, the consumption of a promotion for a non-architecturally-degraded application may also have a positive impact on the infrastructure, since it may guide applications by proposing the kinds of VMs that are more suitable to the current state of the infrastructure. Thus, this scenario aims at evaluating the promotion strategy without applications architecturally degraded and its impact on the VM placement and consequently on the infrastructure power consumption.

Setup

For this scenario we deployed four instances of the application described in Section 7.1.1 (*App1*, *App2*, *App3* and *App4*), all of them with both architectural configurations (k_1 and k_2) enabled and budget of 2.0.

In Figure 7.13, we define the workload for all the four applications. As it can be noticed, applications *App1* and *App3* have the same workload, which starts with a positive ramp up to 100 requests per second. It stays constant for a short time before dropping to 60 requests per second, where it stays until the end of the execution. The workloads of applications *App2* and *App4* start with a positive ramp until 35 requests per second. The workload of *App4* remains constant until the end of the execution, whereas the one of *App2* increases at 65 minutes of execution to 60 requests per second, where it stays until the end of the execution. The reason for this varying workloads is to promote the creation of promotions (triggered by workload decreases) as well as their consumption by other AMs (triggered by workload increases).

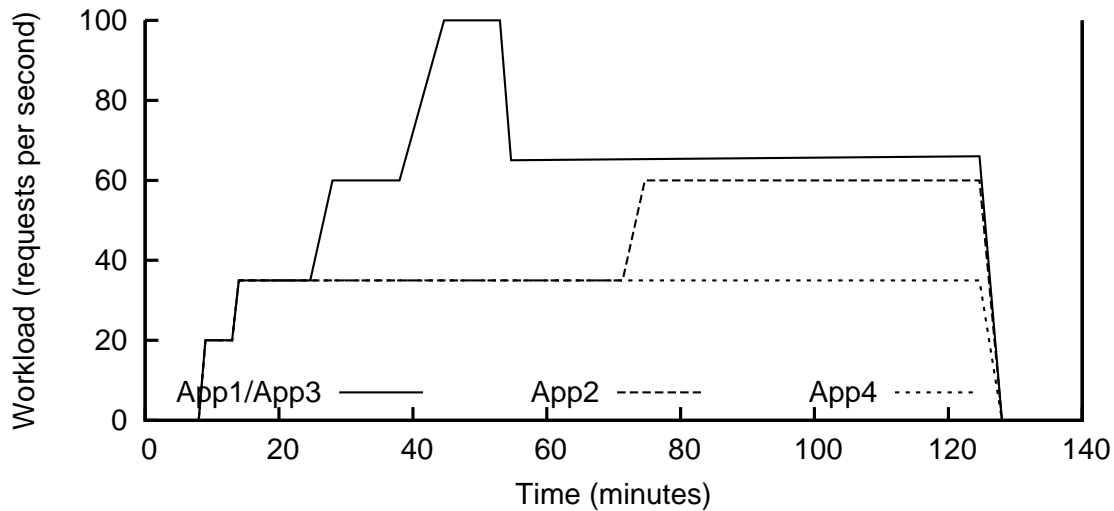


Figure 7.13: Workloads for the Promotion Scenario (without Architectural Elasticity).

With respect to the infrastructure, we used a sub-set of 20 PMs of cluster *sagittaire*. Four PMs to host the VMs for the managers and Clif servers and 15 that were used to host the applications themselves. The time and the condition to activate a promotion or to shutdown a PM are defined identically to the scenario described in the previous section.

Similar to the previous scenario, in this scenario, we also performed several runs by varying the discount rate given by promotions, namely 0.4, 0.5 and 0.6. The promotion duration was fixed to 45 minutes.

Results

This section describes the results obtained from the execution of the promotion scenario without architectural elasticity.

In this context, Figure 7.14 shows the infrastructure income decrease (y-axis) over the time (x-axis). The income decrease is determined by the ratio of the income obtained from the run without promotion to the income obtained from each run with promotion. As expected, until 80 minutes of execution, there is no difference of income among all the runs, since there is no special event (such as a promotion created) to distinguish one run to the other. From 80 minutes on, however, there is a decrease in the incomes of 2% (for the run with discount rate 0.4) and 3% (for the run with discount rate 0.6). The decrease is due to the promotion on some VMs given by the infrastructure provider to some applications. In fact, as the workload of applications *App1* and *App3* decreases (by the time of 45 minutes), VMs are released and consequently free spots are left. In order to fill up those resource free spots, the IM creates some promotions after a couple of minutes. The promotions are taken by *App2*, when its workload increases (between 70 and 85 minutes). The rest of the runs stay constant until the end of the experiment. It is noteworthy that the run with promotion rate 0.4 behaves in a similar manner than the run without promotion. This is because a discount rate at 0.4 for few number of VMs may lead to a price that is close to the regular price. For example, two VMs of class m_1 will cost $0.18 * 2 * 0.6 = 2.16$, whereas a single VM of class m_2 costs 2.27.

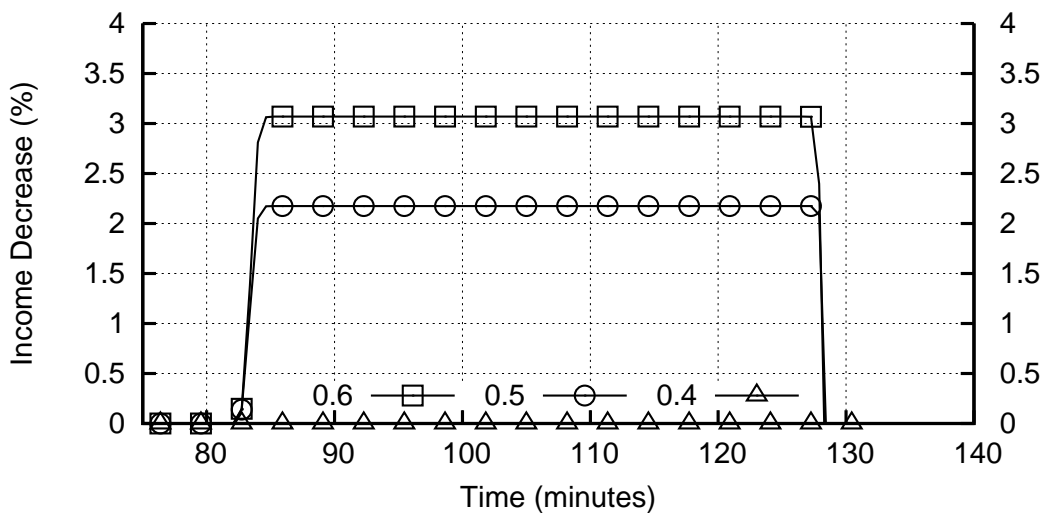


Figure 7.14: Infrastructure Income Decrease (%) with Respect to the Run without Promotion (Without Architectural Elasticity).

Unlike in previous scenario, where the promotions make the IM have an income increase, in this scenario, the IM have an income decrease (cf. Figure 7.14). In fact, in the previous scenario, a set of architecturally degraded applications take promotions in order to be upgraded. Without those applications, these promotions would never been taken. As a consequence, the IM leases more VMs, even though for a lower price. In the case of this scenario (without architectural elasticity), all applications have potential to get promotions due to their workload fluctuation. Instead of taking promotions because they are degraded, they get them after a workload increase and they would get VMs even if they were not in promotions (that is why we fixed the budget at 2.0). Therefore, the difference between VMs with and without promotion will make the IM have a decrease in the incomes (due to the given discounts). On the other hand, promotions can be useful to guide applications' decisions in terms of class of VMs, that is, the IM can influence on applications choice so as to have requests that are more appropriated for the infrastructure, that is, that improves the utilization rate.

Figure 7.15 shows the total power consumption (y-axis) over the time (x-axis). The power consumption is equal to zero at the beginning of the experiment, since all the PMs are previously shutdown. Due to the workload increase of all the applications (between 0 and 50 minutes) and the

decrease (at 60 minutes), the power consumption increases in the same proportion and few minutes later decreases. By the time of 80 minutes, the power consumption differs according to the run, which is to say that for the run without promotion and that with discount rate 0.4 (which does not affect the promotions), it increases by following the workload of application *App2* (Figure 7.13) and stays constant until the end of the experiment. For the rest of the runs, the power consumption stays constant until the end of the experiment, which is explained by PMs that are not switched on to accommodate the new requested VMs. Instead, applications are encouraged to request VMs whose dimensions fit in the existing free spots in the infrastructure, avoiding more PMs to be turned on. Figure 7.16 shows the difference between the total energy consumed by the infrastructure with and without promotion during the almost 45 minutes of promotion. Not surprisingly, there is almost no difference in the energy consumed by the run without promotion and the run with 0.4 of discount rate, whereas in comparison to the runs with discount rates of 0.5 and 0.6 there is an energy saving of as much as 550kJ in the 45 minutes of promotion duration.

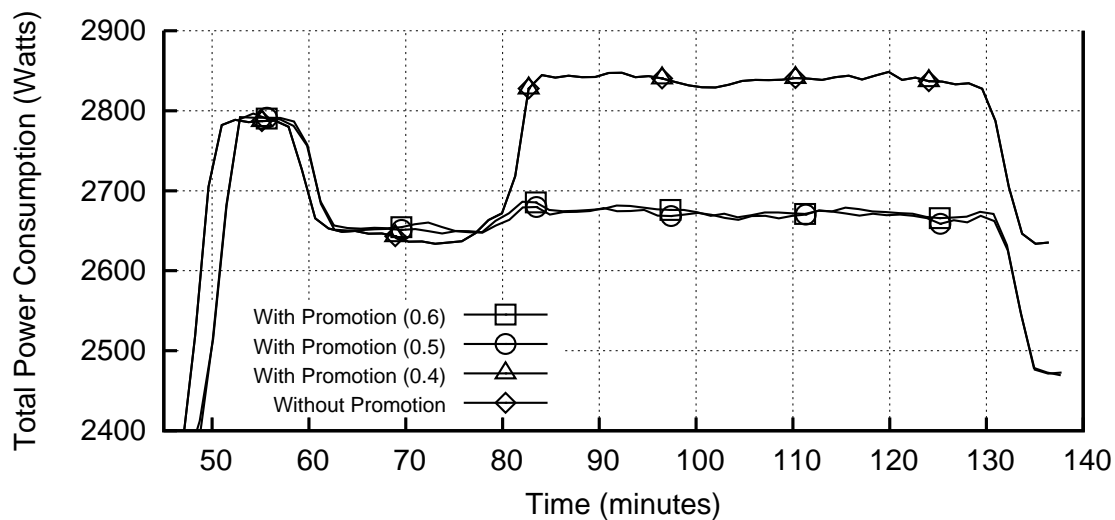


Figure 7.15: Infrastructure Power Consumption for the Promotion Scenario (without Architectural Elasticity).

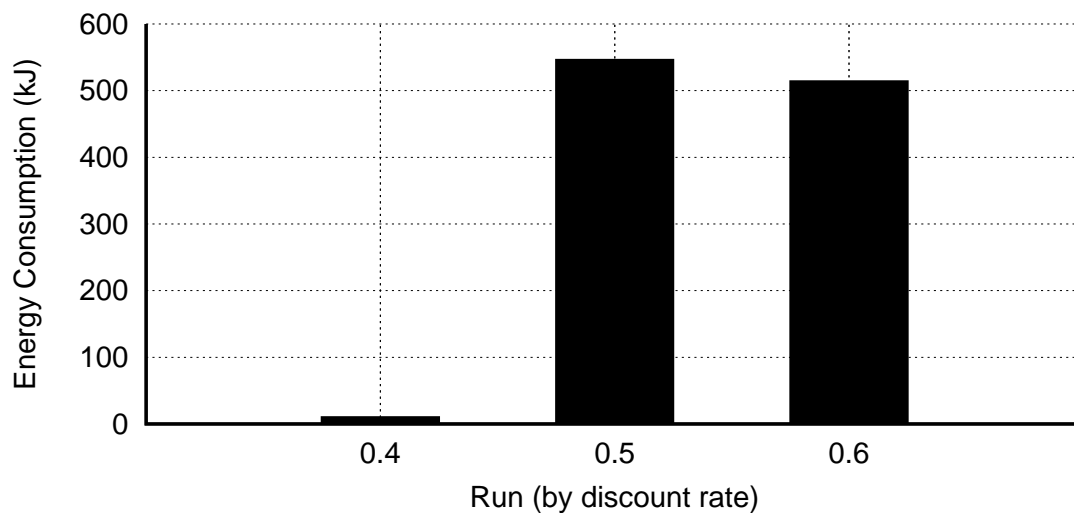


Figure 7.16: Energy Consumption Difference between Promotion-disabled and Promotion-enabled Runs.

It is straightforward that there is a trade-off between the best placement of VMs (and conse-

quently a reduction on the power consumption) and the discount given by promotions. For that reason, Figure 7.17 shows the normalized ratio of incomes (Figure 7.14) to the power consumption (Figure 7.16). What is important to notice in this chart is the difference between runs, even if it is small. More precisely the difference between the run without promotions and the ones with promotion. During a good part of the experiment, the ratios of all runs have almost no difference between them, except during the period of promotions (between 80 and 130 minutes). As it can be seen, the difference increases as the discount rate decreases, apart from the run with discount rate 0.4 in which applications behave exactly as if there were no promotions. It is also important to remark a small increase of the ratio in the runs with promotion, by the time of 130 minutes, which is due to the end of the promotions, when VMs on promotion become at full price again, increasing the incomes and thereby the ratio.

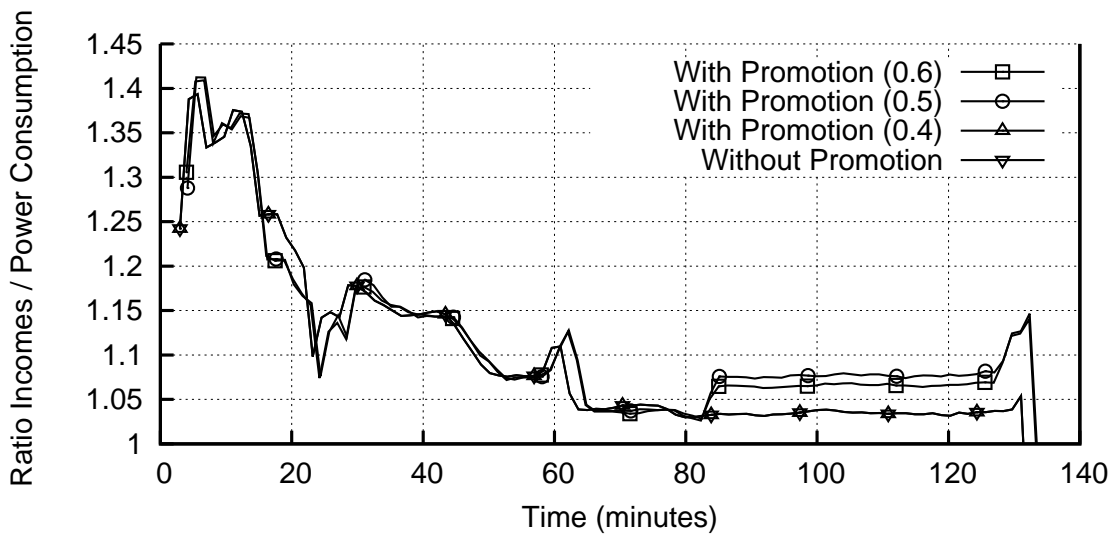


Figure 7.17: Ratio of the Normalized Income to the Power Consumption.

7.1.6 Discussion

This section provides some discussion regarding the results obtained from the experiments described in the previous sections, whose main objective is to evaluate the effectiveness of the approach proposed in this thesis work in terms of QoS at application levels and energy efficiency at the infrastructure level.

The budget-driven architectural elasticity scenario focused at the application level. More precisely, it was used to exploit the capability of adaptation of applications when faced to constrained situation (such as a budget constraint) to cope with the runtime context (increase in the workload). The results show that the architectural elasticity is very effective to make the application more flexible and therefore more able to adapt to that kind of situation. Moreover, it succeeds in keeping a better ratio of the QoS to the cost due to renting fees. Indeed, the architectural elasticity becomes an alternative of adaptation other than just a resource capacity elasticity (request and release of virtual resources).

Likewise, the scale down scenario also focused at the application level as well as at the infrastructure level. In fact, the idea was to observe the synergy between levels, which are promoted by the event-based autonomic protocol proposed in Chapter 5. As the results showed, the architectural elasticity plays an important role in this kind of scenario, where applications are faced to extremely constrained situations in terms of resources. In fact, the resource capacity elasticity alone is not able to cope with those restrictions so as to keep a certain level of performance QoS for all the existing application users. At the infrastructure level, the synergy enables the IM to

nicely communicate with AMs. As a result, the IM may cope with an energy shortage period while mitigating the impacts at the application level.

With respect to the existing work, some focus only at the infrastructure level [FRM12, HLM⁺09, LLH⁺09, BB10] by optimizing the placement of VMs, whereas in this thesis work we take into account both levels: application and infrastructure. Other work proposes resource elasticity approaches to cope with varying workload [AB10, Fer11, VTM10, APTZ12, PCL⁺11]. Some of them [APTZ12] try to establish a trade-off between response time and rejected requests (by using admission control techniques). Others [VTM10, PCL⁺11, Fer11], take into consideration the Service-Level Agreement to degrade the performance of applications whenever it is possible to improve the energy consumption. However, only admission control may not be enough when applications need to keep a desired level of performance, which is to say that it could be more interesting to answer all the requests but with a certain functionality or even a non-functional aspect of the application disabled. In this case, the architectural elasticity is imperative to accomplish such a level of flexibility. Existing works, such as [MFKP09, GDA10], consider applications as a composition of services that can be reconfigured at anytime so as to adapt to a certain situation. In both cases, applications and infrastructure are managed together as if they were part of the same system, which is not applicable to a cloud scenario, if we assume that cloud infrastructure and applications are managed by different actors, as we assume in this thesis work. That way, the benefits of the architectural elasticity along with the resource elasticity can be fully exploited thanks to the synergy provided by the event-based coordination protocol.

Regarding the promotion scenarios, the focus was more at the infrastructure level, since the promotion strategies aim to optimize the placement of new requested VMs so as to make the infrastructure more energy efficient. The results showed that this kind of strategy may be very interesting to have some control on applications' behaviour in terms of requests on resources. Promotions may give degraded applications the opportunity to upgrade their architectures, while making the infrastructure more energy efficient. Furthermore, the interleave of workload variation phases may lead to the creation and consumption of promotions, which in turn, leads to AM decisions in terms of resources that are more appropriated to the IM. Although the results showed that it is possible to gain in energy efficiency, while applying promotion strategies, this thesis work does not go further into an economical model for pricing VMs. Moreover, we do not rely in any specific model to determine when, for how long, and for how which price promotions should be created. We believe that this involves more advanced studies in the domain of revenue management such as yield management [TVR05] in which a service provider tries to maintain a certain level of control on consumers behaviour according to the demand and some pricing policies. We provide further discussion about this in Section 8. Recent work [CPMK12, ISBA12] deal with the cloud infrastructure from the perspective of a marketplace, in which prices may also fluctuate. Generally, the objective is to fairly determine the price of VMs according to some variable costs such as the energy consumption. In this thesis work, instead, the promotion strategy is adopted in order establish a synergy between applications and infrastructure and thus make applications to take better decisions for the infrastructure's current state. Hence, decisions taken at one level may impact positively at the other level.

7.2 Quantitative Evaluation

This section presents and discusses some results obtained from simulation-based experiments to show the feasibility of our approach in terms of scalability. First, we present the simulation results for the synchronization and coordination protocol of the multi-autonomic management approach proposed in this work (see Section 5.1). Then we present the results from experiments on the optimization problems modeled in constraint programming (CP)[RVBW06] in Sections 5.2.3 and 5.2.2.

7.2.1 Synchronization and Coordination Protocols

The main objective is to show the feasibility of our approach in terms of number of autonomic managers working simultaneously and interacting with each other. Firstly, we describe the experiment setup in terms of environment, autonomic manager configurations. Then, we present the results obtained from the simulations and finally we provide some discussion about them.

Experimental Setup

The experiments were performed on a machine with the following configuration: Intel Core 2 Duo processor, 4GB DRAM, Mac OS X Lion operating system. In order to simulate the autonomic manager, we implemented a simplified version of the prototype presented in Chapter 6. The idea was to build a lightweight piece of software able to simulate locally the coordination and synchronization of autonomic managers. In other words, the common autonomic framework presented in Section 6.1 was extended with the same events and actions defined for the AM and IM. Each autonomic task consists of a sleep command that simulates its execution time. The programming language used to implement it was Java 6.

The execution time for each task of each MAPE-K control loop j was fixed as follows: $T^{jA} = 2 * T^{jP}$ and $T^{jE} = 3 * T^{jP}$, that is, the execution time for the analysis task is twice that of the planning task, whereas the execution time of the execution task is three times the one of the planning task. For sake of simplicity we assigned the same execution time values for all the autonomic tasks. More precisely, for all AMs and IM $T^{amA} = T^{imA} = 200ms \pm \epsilon_A$, $T^{amP} = T^{imP} = 100ms \pm \epsilon_P$ and $T^{amE} = T^{imE} = 300ms \pm \epsilon_E$, where ϵ_A , ϵ_P and ϵ_E means a variation of more or less at most 20% of the value.

We chose those proportions because we understand that execution task is the most time consuming, since it requires to communicate with the managed system. Actions for deploying components, creating a VM, among others, may indeed take a lot of time to execute. The analysis task often involves combinatorial optimization and may also take a long time. The planning task may also involve some combinatorial optimization, but less costly than the analysis task, since it often works with a more restrained search space (already restrained in the analysis task). In fact, those values in isolation are not necessarily important. The most important is the timing configuration of the entire autonomic manager, since the timings can determine the rate a given manager is able to treat a given event.

We generate the arrival rates for the endogenous events based on a Poisson distribution. Table 7.6 shows two classes of arrival rates used in the experiments: high and low. As it can be seen, we assigned higher probability values for *Workload Increased*, *Workload Decreased* and *Low PM Utilization* in comparison to *Unused PM* and *Energy Shortage*. This is because the three first events are more likely to happen in comparison to the two last ones. The event *Promotion Expired* was omitted in these settings, since its behaviour is similar to a *Workload Decreased* event.

Class	Workload Increased	Workload Decreased	Low PM Util.	PM Unused	Energy Shortage
High	0.1	0.1	0.1	0.05	0.01
Low	0.05	0.05	0.05	0.025	0.01

Table 7.6: Arrival Rates for Endogenous Events.

When the AMs detect a *Workload Increased* event, the probability that the result of the analysis task requires a *Request VMs* interloop action is fixed to 70% (i.e. $\rho_{wi}(modify) = 1$ and $\rho_{wi}(interloop) = 0.7$). Idem for a *Workload Decreased* event, i.e. the probability that the analysis tasks results in a *Release VM* interloop action is fixed at 70% ($\rho_{wd}(modify) = 1$ and $\rho_{wd}(interloop) = 0.7$). That way, we stimulate more the communication inter autonomic managers, instead of simulating more actions on the managed system, which does not have a significant

impact on the global system. Similarly, when they receive a *Renting Fees Changed* event, the probability that the result of the analysis task requires a *Request VMs interloop* action was fixed at 30% (i.e. $\rho_{r_{fc}}(modif) = 0.3$ and $\rho_{r_{fc}}(interloop) = 1$), that is, in 70% of the cases the autonomic manager processes this kind of event and terminates in the analysis task, because AMs rejects the proposed promotions. The others events are treated in a deterministic way (i.e. $\rho_i(modif)$ and $\rho_i(interloop)$ are equal either to 0 or 1) according to the sequence of autonomic tasks and actions resulting from a given event (cf. Figures 5.8 and 5.6).

Regarding the public knowledge, we defined a single public knowledge referring to the renting fees, and a single critical section corresponding to all the promotions. Moreover, AMs request tokens in a blocking way before the analysis for processing *Workload Increased* and *Renting Fees Changed* events. We define a single critical section with the purpose to evaluate the cost of the synchronization on the overall autonomic managers performance and as a consequence the system scalability.

We perform several runs while varying the number of AMs: 10, 20, 30, 50 and 70. The idea is to observe how the variation of the arrival rates and number of AMs can affect the system performance (e.g. the token waiting time and events processing time). The objective was to perform simulation-based evaluations regarding the system stability and scalability when dealing with an increasing number of autonomic managers.

7.2.2 Results

Stability

Figure 7.18 shows the average token waiting time T_{lock} evolution in time. Each line corresponds to one run regarding a different token number of AMs in the system. When dealing with low arrival rates (Figure 7.18 (a)), for 50 and 70 AMs, T_{lock} increases until it reaches a peak and stabilizes afterwards. It means that the token manager waiting queue reaches its longest size, and keeps serving at a more or less constant rate. For 10, 20 and 30 AMs, T_{lock} remains always under 10000ms. When dealing with high arrival rates (Figure 7.18 (b)), we can observe a similar behavior for all the curves. Notice that T_{lock} rapidly increases for the highest numbers of AMs (i.e. 30, 50, 70) and stabilizes afterwards.

Scalability

Figure 7.19 presents the evolution of the event processing time (T_i^j , for autonomic manager j and event i) when varying the number of AMs in the system. Not surprisingly, *Workload Increased*, *Renting Fees Changed*, *Scale Down* and *Workload Decreased* trigger the most time consuming processes, since the two first ones might be followed by a token request, which may lead to a sharp increase of the token manager queue. The two last ones may stay stuck waiting until the others have finished.

Discussion

With respect to the stability, there might be high token waiting times as the arrival rates approach the service rate (frequency in which an autonomic manager can process an event). For instance, a high rate of the *Workload Increased* event along with a high number of concurrent AMs may produce a token arrival rate that might exceed the token manager service, leading to an infinite growth of the token manager queue. However, as long as the arrival and service rates are well balanced, the token waiting time always tend to stabilize. Alternatively, as mentioned in Section 5.2.3, another solution is to partition the public knowledge in several critical sections, each one requiring a different token, which are requested in an opportunistic (i.e. in a non-blocking way). Given that there is no blocking calls, there is no waiting time.

Conversely, the number of AMs along with high arrival rates may have a negative impact on the system scalability. For instance, for events that depend on a token, a long token waiting time

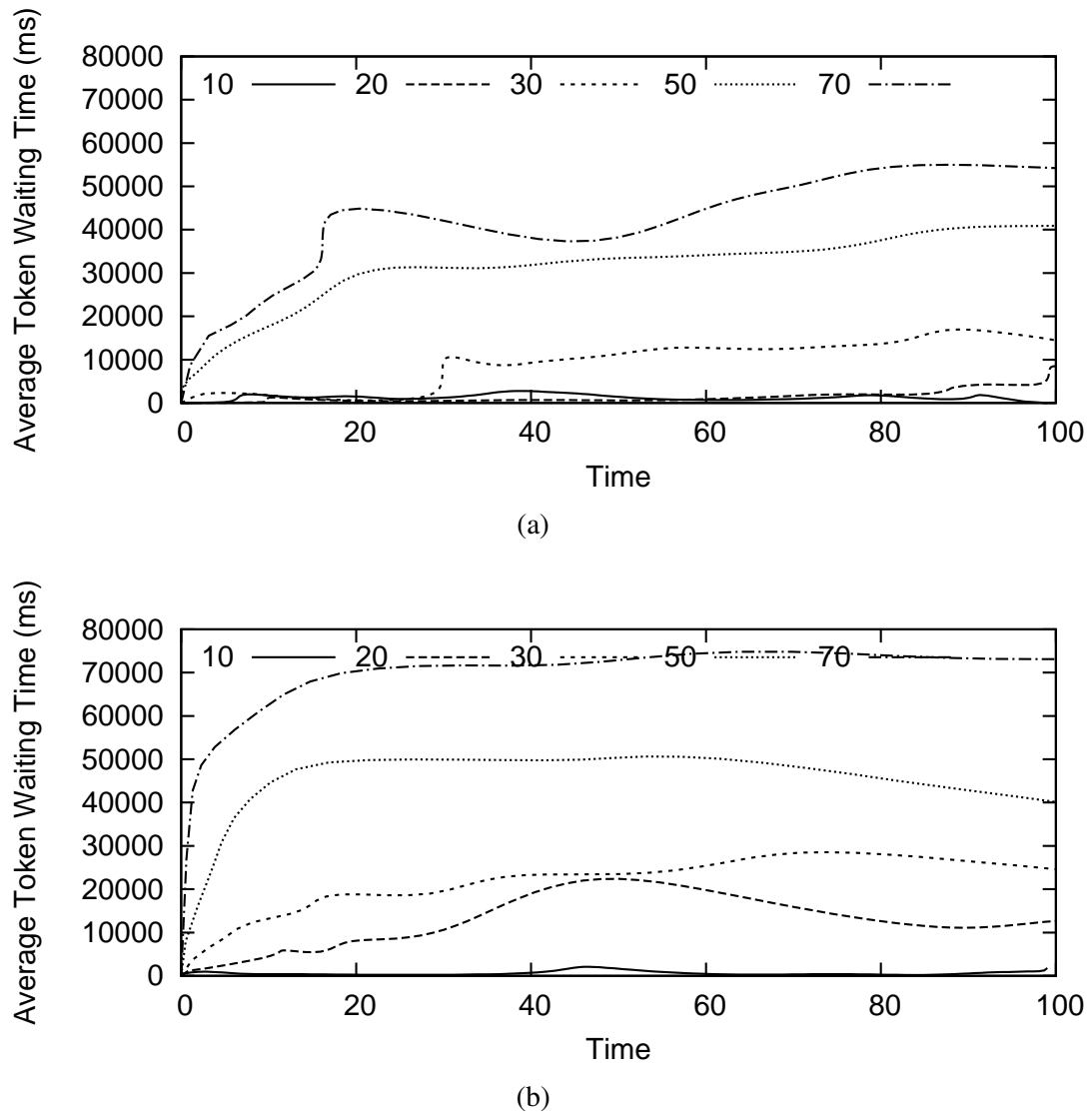


Figure 7.18: Average Token Waiting Time for (a) Low and (b) High Arrival Rates.

may lead to long event processing time. Thus, the more AMs the longer the token waiting times and consequently the event processing times. Again, by adjusting the arrival rates and the number of AMs in the system properly, the system can be scaled with respect to the number of AMs and the frequency of requests produced by them (resulting from the coordination). Although the control of exceeding arrival rates has not been tackled in this work, we believe that admission control techniques [WKGB12], along with event prioritization / preemption policies, can be effective to implement it (cf. Chapter 8).

Existing work proposed the use of multiple autonomic managers in the context of cloud computing [KCD⁺07, RRT⁺08, KLS⁺10, MKGdPR12]. In [RRT⁺08], the autonomic managers are restrained to the infrastructure management, whereas in our proposal the autonomic managers are associated to both applications and infrastructure. In [KCD⁺07], there is a tight-coupling information sharing which makes the problem management in each layer very dependent to each other. In this work, instead, we proposed an event-based coordination protocol and a synchronized public knowledge that shares only the essential information without giving any details about the infrastructure runtime context. In [KLS⁺10], a synchronization algorithm and a coordination protocol is proposed to make several autonomic managers (at both application and infrastructure levels) work together to reduce the power consumption at the infrastructure and improve the QoS of applica-

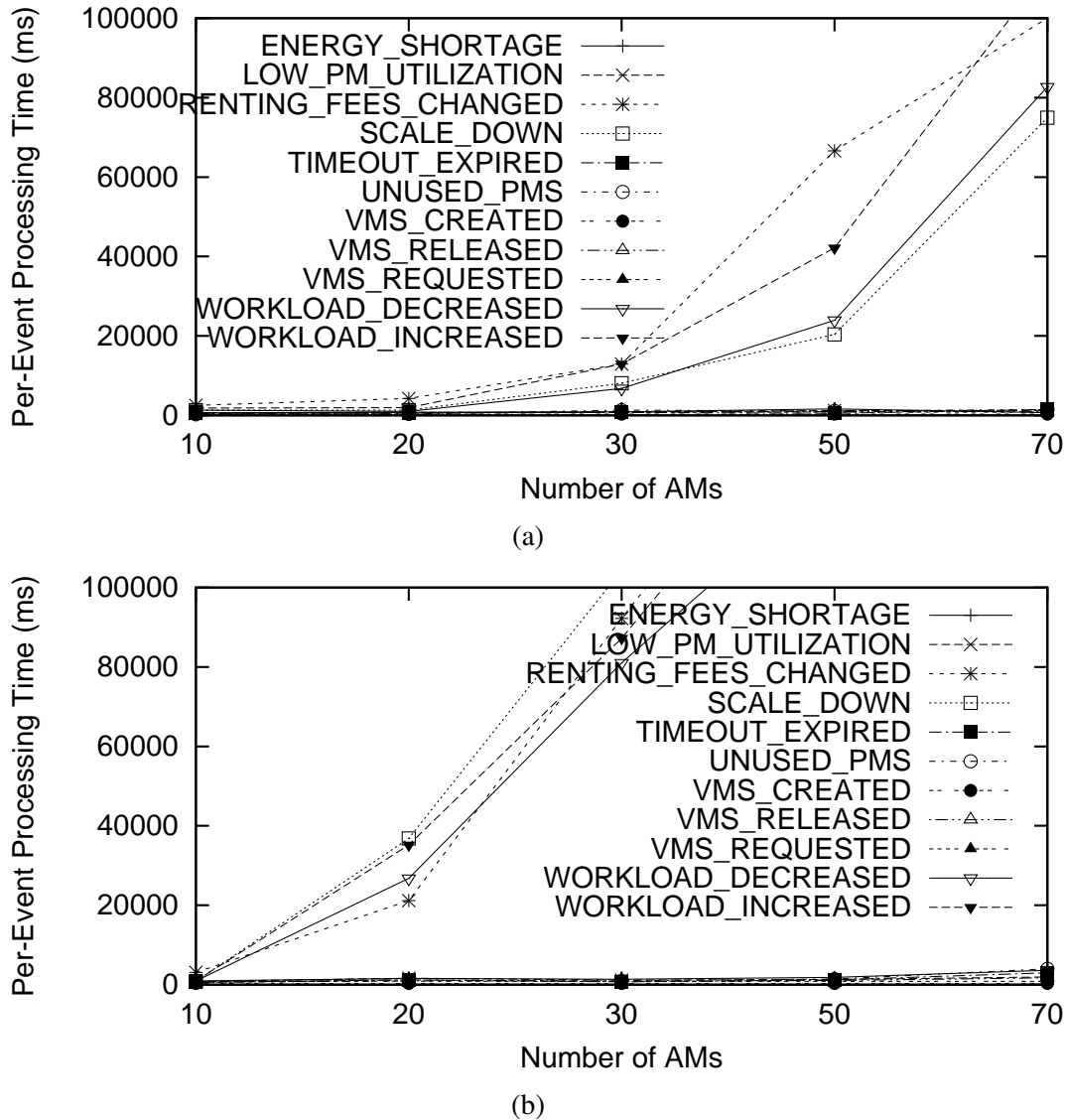


Figure 7.19: Per-Event Processing Time for (a) Low and (b) High Arrival Rates.

tions. The power management, as in other works [WW11], consists in adjusting the right level of Dynamic Voltage Frequency and Scaling (DVFS) of PMs, whereas in this work it is supposed that the entire PMs can be shutdown, which can be more effective in terms of energy savings. However, in our work, the idea of consensus among autonomic managers to decrease their QoS and then decrease the power consumption is not taken into account. Instead, we suppose that the power consumption is decreased either as a consequence of an unexpected event (*energy shortage*) or by the creation of promotions that guide applications decision. In [MKGdPR12] a synchronization model is proposed to avoid conflicting actions (such as performance and power consumption) in a multi autonomic manager environment. However, in that work, as in other works such as [CFF⁺11], the scenario presented consists in only one application and one infrastructure. In this thesis work, we consider a multi-application environment, each one equipped with its own autonomic manager. This is more suitable in multi-vendor/clients environments such as in cloud computing. Although conflicting actions or events can be sometimes avoided because of the synergy created either by the coordination protocol (e.g. energy shortage) or by the public knowledge (e.g. promotion strategies), in order for the proposed approach to be effective in resolving conflicts, it would be necessary more advanced scheduling policies and transactions mechanisms so as to allow to preempt current event treatment in MAPE-K control loops. These issues are left as

future work (cf. Chapter 8).

7.2.3 Optimization Problems

In this section, we present the performance evaluation of the solvers implemented to deal with the optimization problems of each autonomic managers' analysis task. The objective is to show how they scale by exposing the limitations of CP. First, we describe the experimental setup, then we present the results obtained from all executions and finally we provide some analytical discussion about them.

Setup

Execution Environment. We performed the experiments on a machine equipped with an Intel Core 2 Duo processor, 4GB DRAM and Mac OS X Lion operating system. As previously mentioned in Chapter 6, all the solvers were implemented in Java 6 and Choco 2.0 [Tea10] was used as CP solver engine.

Infrastructure. For this evaluation, we consider that the infrastructure is composed of a set of homogeneous PMs P such that $pm_i \in P$ $pm_i^{cpu} = 4$ and $pm_i^{ram} = 4$. With respect to the VM classes, we defined the set $M = \{m_1, m_2, m_3\}$, where $m_1^{cpu} = 1$, $m_1^{ram} = 1$, $m_1^{cost} = 0.18$, $m_2^{cpu} = 2$, $m_2^{ram} = 2$, $m_2^{cost} = 0.27$, $m_3^{cpu} = 4$, $m_3^{ram} = 4$ and $m_3^{cost} = 0.45$.

Application. Regarding the application definition, which is used for the *Architecture and Capacity Planning Problem*, we defined it several times by varying the number of components. We specified the needs of the application in terms of resources and the budget in a way that it could be possible to fix a number of VMs of each class that a component can be assigned to. We defined the number of architectural configurations at one, since it does not change the algorithm performance. On the contrary, the decision variable for the configuration propagates the constraints associated to the constituent components, which may restrain the search space. Idem for the promotions.

Problems. In this experiment, we focus on the optimization problems that were modeled and solved with CP (cf. Sections 5.2.3 and 5.2.2). Other analysis problems that does not rely on CP such as the planning algorithms were not evaluated since they are not related to optimization and can be solved (although not optimally) in polynomial time. More precisely, we focus on the following problems:

- *Virtual Machine Placement Problem.* The VM placement problem is implemented as a VM packing problem, in which a number of VMs should be placed in a number of PMs so that the number of PMs is minimized. The idea is to run the solver several times by varying the number of VMs to be placed and the number of hosting PMs. The VMs were created according to the following proportion: 50% of class m_1 , 35% of class m_2 and 15% of class m_3 . Hence, since the great majority of VMs are from class m_1 (smallest) and m_2 (middle), we stimulate a high consolidation rate of VMs in PMs while keeping the heterogeneity in terms of VM class (3, in this case). It is important to say that the consolidation rate interferes directly in the search space size and consequently the solving time.
- *Energy Shortage Problem.* The energy shortage problem consists in, given a number of PMs to shutdown, determining which ones should actually be shutdown. It is important to mention that the choice for the PMs to be shutdown and consequently their hosted VMs is done in a way all applications have their resources shrank while meeting their resource restriction rate (cf. Section 5.2.2). The experiment consists in running the solver several times by varying the number of PMs that need to be shutdown, the number of clients and the total number of PMs (500, 1000, 2000 and 4000). For this purpose, we define four

runs (Table 7.7) that specifies the value of each parameter (number of clients and number of PMs to shutdown) in function of the total number of PMs in the infrastructure. That means that for 500 PMs, the run $r1$ will evaluate the solver for 50 PMs to be shutdown and 5 applications. The infrastructure was created to be fully utilized with 50% of VMs of class m_1 , 35% of VMs of class m_2 and 15% of VMs of class m_3 . Lastly, the resource constraint rate are assigned equally to all the applications according to the percentage of PMs to be shutdown. In addition, we add a small positive margin (Table 7.8) to the resource restriction rate according to the number of applications so as to avoid over-constrained problems and consequently very long execution times. In other words, we leave a margin between the percentage of resource each application accepts to release and the amount of resources that has to actually be shutdown.

Run	Shortage Percentage	Applications Percentage
$r1$	0.05	0.01
$r2$	0.1	0.02
$r3$	0.2	0.05
$r4$	0.5	0.1

Table 7.7: Energy Shortage Parameters.

- *Architecture and Capacity Planing Problem.* This problem consists in determining the minimum amount of resource necessary to host the application's component so as to minimize its degradations in terms of both performance and architecture. (cf. Section 5.2.3). The experiment consists in running several times the solver by varying the number of components (from 2 to 30) as well as the number of VMs of each type that each component can be assigned to (10, 20 and 30).

Results

This section presents the results obtained from the execution of the experiment described in the previous section. In order to avoid inconsistent results, we performed ten executions for each experiment and calculated the average of the value resulting from each execution. The rest of the section presents the evaluation results for each problem mentioned in the previous section.

Virtual Machine Placement Problem. Figure 7.20(a) shows the execution time (y-axis) for the VM packing problem when varying the number of VMs to be placed and hosting PMs (x-axis). Not surprisingly, the execution time grows exponentially according to the number of VMs and PMs, although some useful optimizations techniques such as symmetry (cf. Section 2.4) are internally implemented and shipped within Choco. The results are expected because the VM packing problem is an instance of the NP-hard bin-packing problem [JGJ97b]. However, the VM packing problem is useful when migration is taken into account during the adaptation process, otherwise

Number of Applications	Margin
5	0.01
10	0.02
20	0.03
50	0.05
100	0.06
200	0.07

Table 7.8: Energy Shortage Constraint Margins.

the VMs can only be removed from their original hosting PMs to be destroyed. Thus, the VM placement problem can be seen as a more restrictive instance of the VM packing problem, in which the existing VMs cannot be removed from its original place. As consequence it reduces significantly the search space and thereby the solving time. Figure 7.20 (b) shows the solving time for the virtual machine placement problem when increasing by 50 the number of placed VMs in 1000, 2000 and 4000 PMs. As it can be noticed, the solving time decreases as the number of placed VMs increases. Although these delays can be accepted for small or medium-sized infrastructures, it becomes infeasible for huge (tens of thousands of PMs and VMs). One alternative can be to partition the infrastructure in a hierarchical way so as to reduce the number of PMs to be managed for each partition [FRM12, BB10]. Alternatively, a decentralized and collaborative approach [QLS12] seems to be very promising for this kind of problem.

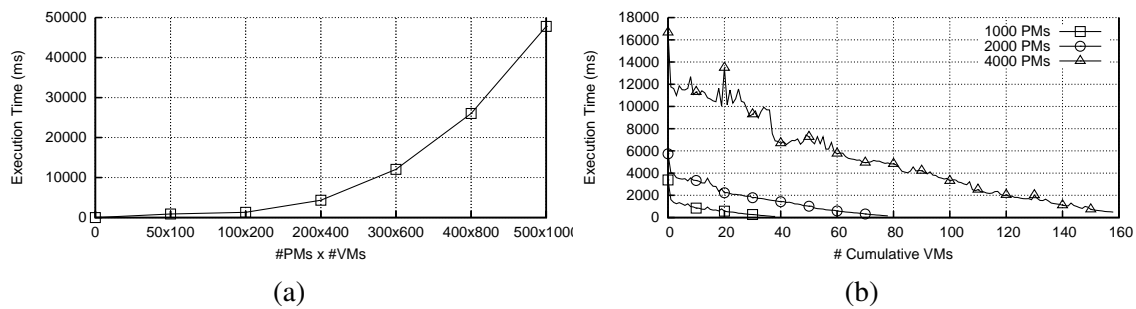


Figure 7.20: The Solving Time for (a) the Virtual Machine Packing Problem and (b) The Solving Time for the Virtual Machine Placement Problem.

Energy Shortage Problem. Figure 7.21 shows the solving time (y-axis) for the energy shortage problem, when executing the runs specified in Table 7.7 (x-axis). As it can be seen, the solver for the energy shortage problem sharply grows as the number of PMs involved also increases, which is logical since the number of PMs is proportional to the search space. Regarding the number of PMs to shutdown and the number of clients, we can notice that the solving time also grows when those parameters increase. In fact, the more clients more difficult it is to find a solution that meets the resource constraint rate of each client. Similar to the VM placement problem, the solving time can be considered acceptable for small and medium-sized infrastructures, even with a drastic shortage (e.g. that requires 50% of PMs to be shutdown). One alternative to outperform these results can be to loose (decrease) more and more the resource constraint rates associated to each application. In other words, to always try to have resource constraint rate that gives a good margin of difference (bigger) from the amount of resources that should be shutdown. As a consequence, more solutions will be available and thus reducing the solving time.

Architecture and Capacity Planning Problem. Figure 7.22 shows the solving time for the component and capacity planning problem (y-axis) when varying the number of components (x-axis) and the number of possible VMs of each type per component. Given that this problem is also an variant of the NP-hard bin-packing problem, the solving time grows exponentially as the number of components increases. In fact, every additional component corresponds to q (number of VM classes) more variables to take into account in the problem solving process. Moreover, when increasing the number of VMs (of each class) a component can be assigned to, there is an increase in the domains of each decision variable of the problem, which consequently leads to an explosion of the search space size. Although some of the delays may seem too long, for small and mid-sized applications (15 to 20 components), a solving time of 20 seconds or even more can be considered acceptable in some cases, depending on the reactivity frequency of the AM required by the managed application.

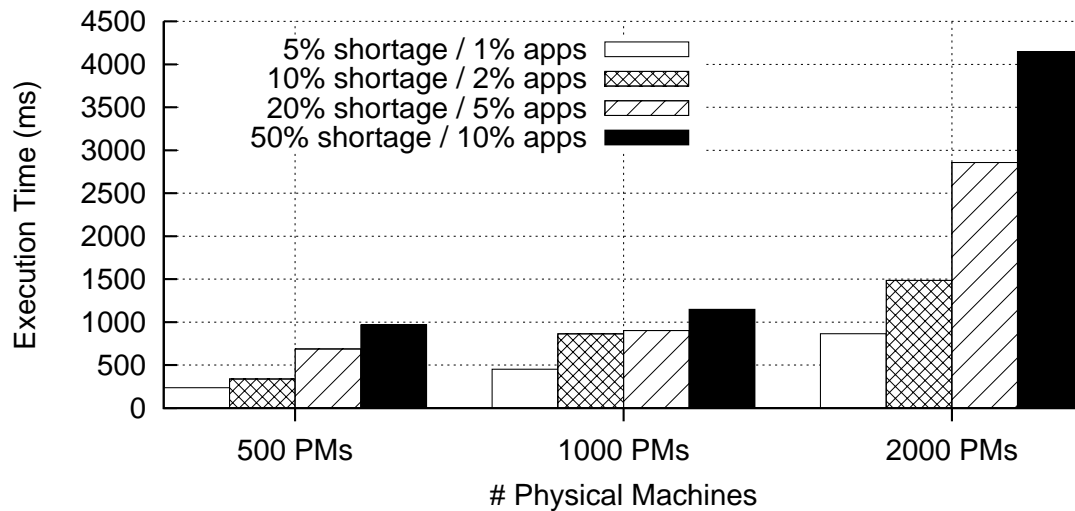


Figure 7.21: The Solving Time for the Energy Shortage Problem.

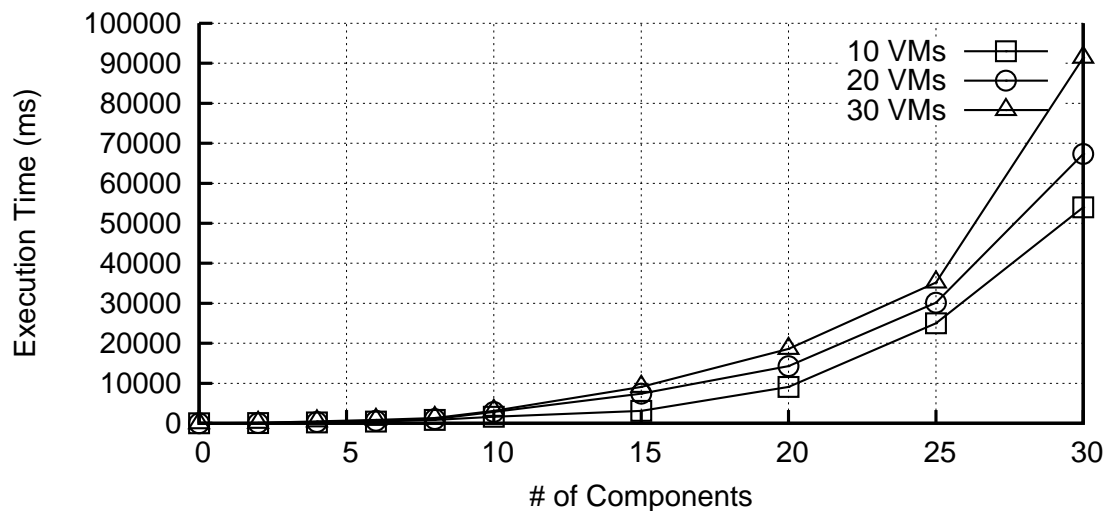


Figure 7.22: The Solving Time for the Architecture and Capacity Planning Problem.

Discussion

This section showed some performance results of the CP solvers in terms of performance. Many of problems dealt by CP are related to combinatorial optimization and thus are difficult to solve in a reasonable time for large inputs. Even though, for the three problems presented in this section, thanks to some optimizations implemented internally in the solver engine (e.g. symmetry and propagation) or externally (e.g. domain restrictions), we obtained acceptable results for small and medium-sized applications and infrastructure, while taking the advantage of not bothering about the solution process. Indeed, the declarative way in which the problems are expressed (in terms of variables and constraints) eases the maintainability and extensibility of the solvers. As a consequence, common problems in optimization solutions such as side-effects can be avoided. Other advantages of CP such as reuse and composition of constraints should also be highlighted.

With respect to existing work, [FRM12, HLM⁺09, LLH⁺09, BB10] proposed solutions to perform multi-dimensional bin-packing to perform server consolidation and therefore make the infrastructure more energy efficient. As showed in the virtual machine placement problem evalua-

tion, we perform VM placement, instead of a pure VM packing. Although a VM packing allows a higher degree of energy efficiency, we decided not performing it in order to avoid VMs migration overheads. Instead, we rely on promotion strategies (see Sections 7.1.4 and 7.1.5) so as to increase the synergy between applications and infrastructure and consequently improve the infrastructure utilization rate.

Concerning the architecture and capacity planning problem, we extended the constraint programming model proposed in [VTM10] so as to take into account several architectural configurations as well as promotions.

7.3 Summary

This chapter presented two experimental evaluations on the approach proposed in this thesis work.

Firstly, we presented a qualitative evaluation, whose objective was to show the effectiveness of the approach at both the application and infrastructure levels. To this end, we implemented a dynamic and reconfigurable service/component-based application equipped with architectural and resource capacity elasticity. The application was deployed on a real experimental infrastructure and exercised with a load injector to simulate the clients' behaviour. In order to evaluate the approach effectiveness in terms of quality of service and energy consumption, the experiments were driven by three scenarios. The first one aimed at showing the benefits of the architectural elasticity when dealing with constrained configurations, more precisely with a budget-constrained situation. The second scenario is also related with the capability of applications to adapt themselves to constrained environments. More precisely, applications are faced to energy shortage, in which part of the infrastructure should be switched off. For both scenarios, the results showed the importance of the architectural elasticity to cope with both budget and energy constraints, since applications equipped with architectural elasticity manages to maintain the performance quality of service for the same workload in comparison to applications without architectural elasticity. Hence, the coordination protocol along with the architectural elasticity may significantly contribute for an improvement in the synergy between applications and infrastructure providers, since the infrastructure providers succeeds in dealing with constrained situations (energy shortage) while mitigating the impacts at the application level.

The third scenario aimed to show the effectiveness of the public knowledge and promotions at the infrastructure manager in improving the synergy between applications and infrastructure. The results showed that this strategy can improve the synergy in the sense that applications' behaviour in terms of resource request may be driven by the creation of promotions. As a consequence, the decisions in terms of computing resources to request are more suitable for the infrastructure's current state with respect to the optimality of resource utilization and consequently energy efficiency.

At last, we presented a quantitative evaluation, in which we showed how some critical algorithms and protocols scale in terms of time. For that purpose, we performed two kinds of experiments: one to evaluate the multi-autonomic management model and its protocols and another to evaluate the optimization problems related to the analysis task of both application and infrastructure managers. Regarding the multi-autonomic management approach proposed in this work, we implemented a simulator in order to evaluate the system scalability and stability when varying the number of application managers. As in any synchronized system, there is a non-neglect cost implied by the control of the concurrent access to critical sections of the public knowledge. Even though, the system stabilizes after sometime and depending on the arrival rates of events (in both managers), the delays for acquiring a token or to treat an event can be considered acceptable. Finally, with respect to the optimization solvers, we executed several times the solvers implemented with constraint programming while varying the input data. Although the limitations related to scalability of NP-hard problems such as bin-packing problem as well as the limitations of generic and declarative approaches like constraint programming, the experiment yielded acceptable delays for small and medium-sized infrastructures (number of physical and virtual machines) and applications (number of components and virtual machines).

In conclusion, this chapter showed the feasibility of the approach proposed by this thesis work not only with respect to the improvement of the synergy between applications and infrastructure providers, and its consequences on the quality of service and energy efficiency, but also with respect to the scalability, although the complexity of the problems tackled.

Conclusion

This chapter concludes this thesis work by firstly revisiting the problem statement, then by summarizing the main contributions yielding from this research and finally by pointing out some perspectives emerged from these contributions.

8.1 Problem Statement Revisited

Over the past years it has been noticed a drastic increase of the energy consumption due to information technology (IT) infrastructures, which can be explained in part by the popularization of internet and the wide adoption of recent internet-based service models like cloud computing. On the one hand, cloud models relies on 24/7-available and limitless resource provisioning principle, which may lead to high levels of consumption, even if there is no need for that i.e. if the demand for a given service is way lower than its capacity. On the other hand, the flexibility of the on-demand provisioning made by cloud services can also contribute for a better service dimensioning and consequently the reduction of energy consumption. In fact, the mutualization of physical resources, which is possible thanks to techniques like virtualization, along with autonomic computing concepts enable not only infrastructures to self-manage so as to improve their utilization rate, but they also allow applications to self-manage by dynamically adjusting their needs in terms of resource according to their demands. In spite of that, autonomic decisions taken in isolation at one level may impact negatively systems running at other levels. For example, due to energy consumption purposes, an autonomic manager at the infrastructure level may decide (or be obliged) to shutdown part of the physical infrastructure. It is straightforward that this behaviour may affect the applications hosted on it. In fact, the lack of explicit synergy between managers along with the lack of flexibility of managed systems may compromise managed systems' QoS. For instance, in extreme cases such as *energy shortage* at the infrastructure level, applications may be asked or even forced to release part of the resources allocated to it. Scenarios like that may dramatically impact the running systems if there is no proper interface to promote a synergy between managers nor if the system is not capable of self-adapting in order to absorb that level of restriction.

8.2 Summary of Contributions

This thesis proposes a multi-autonomic management approach to improve QoS in cloud applications and the energy efficiency in cloud infrastructures. This section summarizes the main contributions made by this approach.

Architectural Elasticity for Cloud Application. Besides the resource capacity elasticity enabled by cloud infrastructure in which cloud applications are able to dynamically scale up and down, we also proposed another level of flexibility, namely architectural elasticity. The architectural elasticity refers to the capability of cloud applications to dynamically change its architectural structure in order to cope with a given constraint in terms of budget or computing resources. This flexibility was achieved due to two main factors: (i) the inherent modularity nature of component-based applications, in which one can define applications in terms of modular pieces of software that can be easily deployed, replaced and composed along with other pieces; and (ii) reflective techniques, which enable application to introspect (to navigate through) and intercede on (to modify) their own structure at runtime. That way, cloud applications are defined in terms of components that can be dynamically, deployed, bound or replaced with the purpose to face runtime changes.

We validated this contribution over a budget constraint and a resource/energy shortage experimental scenarios. The results showed the importance of the architectural elasticity to keep a certain level of performance QoS during such extreme situations.

Autonomic Model for Multiple Autonomic Managers. In order to get several autonomic managers working in a coordinated and synchronized fashion, we proposed a multiple autonomic management model, in which several MAKE-K-like control loops are able to communicate in a synchronized way. The model consists of a coordination and a synchronization protocols. The coordination protocol is a message-oriented protocol based on events and actions in the context of the managed system as well as exchanged among several autonomic managers. This protocol prevents managers to stay blocked waiting for other managers' responses. The synchronization protocol is necessary to keep data consistency in public knowledge critical sections. So, we relied on a token-based protocol in which one token per critical section is distributed at a given time in order to control the concurrent access to it.

Simulation-based results have shown that this model can be feasible (in terms of stability and performance) for a reasonable number of autonomic managers. However, the synchronization mechanism comes at a price: when the number of autonomic managers increases the token acquisition waiting time and consequently the management reaction time (time during which events are processed) also increases in the same proportion. This problem was tackled by adopting an opportunistic approach, that is, instead of waiting for a token, autonomic managers rather get it only if it is available. As a result, there is no more waiting times and the system becomes more scalable.

Application and Infrastructure Managers for the Cloud. Following the autonomic model described in the previous item, we proposed a multiple autonomic management architecture in the context of cloud computing. This architecture is composed of several application managers (AMs) and one infrastructure manager (IM). At the infrastructure level, the IM is in charge of managing the resource lease of a set of physical machines (PMs). It relies on virtualization techniques to deal with AMs requests on resources in a dynamic manner. Meaning that virtual machines (VMs) are requested and released on the fly while the IM takes care of placing them in a way the infrastructure utilization rate and consequently the energy efficiency are optimized. At the application level, a per-application AM is responsible for managing a set of existing components and the way they can be composed (architectural elasticity) as well as the required amount of resources (resource capacity elasticity) for them to maximize the application QoS. To this end, it relies on the flexibility made by both the architectural and resource elasticity so that to not only reconfigure the application's internal structure (e.g. components, bindings) but also to request and release VMs to/from the IM as needed.

We implemented a research prototype encompassing both managers. This research prototype has been used to qualitatively and quantitatively evaluate this thesis work.

Synergy between Applications and Infrastructure. The proposed autonomic model mentioned before relies on an information sharing approach in order to facilitate the synergy among autonomic managers. In the context of the AMs and IM, we defined a public knowledge that refers to the resource renting fees. More precisely, we proposed two kinds of renting fees, a static one, that is, they do not change over the time; and a dynamic one, that may vary according to the infrastructure context. The dynamic renting fees, in this thesis called promotions, are created with the purpose to stimulate AMs to consume this resource and consequently occupy a portion of the infrastructure that is not being used. The promotion strategy enables a certain level of influence of the IM over the decisions taken by the AMs, even if those autonomic managers manage systems from different vendors or located in different layers of the cloud stack. As a consequence, the actions performed by the AMs may be more suitable to the infrastructure.

Apart from the public knowledge based synergy, autonomic managers can also benefit of the coordination protocol to establish a more explicit synergy by means of communication interfaces (actions and events). For example, in periods of *energy shortage*, the IM asks some AMs to release a certain amount of resources, which, thanks to the architectural elasticity, may be manageable by the concerning AMs. As a result, the explicit synergy along with the architectural elasticity make the IM flexible enough to cope with extreme situation in a seamless way.

The proposed synergy approaches were experimentally evaluated. In the case of the public knowledge based synergy (promotion), it was noticed a gain in either the infrastructure utilization rate or the infrastructure total energy consumption. In the case of the synergy based on the communication interfaces, it was observed an improvement in the seamlessness of adaptations. In fact, because of this level of synergy between AMs and IM, applications involved in an extreme scenario (e.g. energy shortage) were able to cope with this scenario while minimizing the impacts on their QoS.

Constraint Programming Based Solvers for Optimization Problems. The analysis task of each autonomic manager is implemented by one or more optimization problems. We proposed a constraint programming (CP) model and solver for each problems above mentioned. CP is a technique to model Constraint Satisfaction and Optimization Problems, in which a problem is described in terms of variables, domains and constraints on these variables. Given a problem description, a general purpose solver engine takes care of solving it. The descriptive way in which problems are modeled frees modelers from the burden of developing a specific solver for each problem. Furthermore, this loose coupling between solver and description besides of allowing the re-use and composition of existing constraints, also improves maintainability and extensibility of existing models.

Nevertheless, that level of generality and descriptiveness to model and solve optimization problems comes at the expense of scalability limitations. For this reason, we performed scalability experiments on each one of the proposed solvers in order to expose their limits. The results showed that solvers are able to provide reasonable solving time for small and mid-sized applications and infrastructures.

8.3 Perspectives

This research work has succeed in addressing the issues already pointed out all over this document. Some of the previously presented contributions arose interesting and promising research perspectives. This section provides some discussion on them.

Message-oriented Middleware Implementation. In this thesis work, we implemented a message-oriented autonomic framework in order to allow the communication among several autonomic managers. Although it was conceptually defined to rely on a message-oriented middleware (MOM), the communication protocol was implemented in a ad-hoc manner over Hypertext Transport Protocol (HTTP). We believe that a MOM broker would ease the communication management in the

sense that it would make the communication details transparent for autonomic management peers. The implementation of a broker-based communication is an on-going work.

Adoption of Other Component-based Framework. This thesis work relied on Service Component Architecture (SCA) to implement component-based cloud applications and thus enable architectural elasticity. More specifically, we relied on FraSCAti [SMF⁺09], a SCA runtime developed on top of the Fractal component model [BCL⁺04]. Although the many advantages of FraSCAti such as the support for dynamic reconfigurations, there are still some limitations regarding the suitability for grid or even cloud application definitions. For instance, there is no support in FraSCAti for the definition of components in different domains (physical or virtual machines), even if they belong to the same application. This is disappointing since it is not possible to build distributed component-based applications while having a single model representation. For these reasons, it becomes necessary to investigate more grid-focused component models such as Grid Component Model (GCM) [BCD⁺09]. GCM is a component model with a focus on distributed and grid computing. It seems to be an interesting track to investigate, since it provides support for both dynamic reconfiguration (also based on the Fractal component model) and component distribution in the context of grid computing.

Fairness and Starvation-aware Token Management. We proposed a token protocol to manage the concurrent access to critical sections, in particular promotions, in public knowledges. Given that there is no control in the way those tokens are distributed, it might happen that the same requester gets always a token, whereas the others never have the chance to get it. This fairness/starvation issue should be addressed as a future work.

Economics-based Pricing Model. This thesis work proposes a public knowledge approach for improving the synergy between application and infrastructure managers which is based on promotion strategies. However, promotions are created in an ad-hoc manner, meaning that we do not precise when, for how long and how (with how much discount) those promotions should be created to optimize the infrastructure utilization and consequently the energy efficiency. Work on the management field such as in the domain of yield management [TvR05] have been studied in order to apply real economics models based on statistical data to determine prices dynamically. This kind of approach is particularly interesting for establishing a synergy between infrastructure and application providers since it focuses on the prediction of consumers behaviour. Alternatively, advanced theoretical models such as game theory [NM44] are also very promising to find an equilibrium in a win-win fashion (in terms of price) among conflicting participants (applications and infrastructure providers).

Application-level Reconfiguration Patterns. In this thesis work, we relied on a load balancing mechanisms to perform scale up/down and on component-based dynamic reconfiguration to perform architectural elasticity. However, there are other reconfiguration patterns that could be applied according to the application context and constraints [NBF⁺12]. For instance, an adaptation pattern could be to turn the application into a multi-tenant application so as to consume less resources and cope with a given resource/energy restriction. The challenge is to determine which patterns can be applied and whether they can be applied together. This is an on-going work in collaboration with a Ph.D. student from ASCOLA Research Group at the École des Mines de Nantes and Sigma, an IT French company.

Service Level Agreement (SLA) Support. In the optimization problems presented in this thesis work, we considered some degradation criteria (in terms of performance and architecture) that can be seen as objectives (or Service Level Objectives) or guides to avoid Service Level Agreements (SLAs) violations. Despite that, those criteria might not be enough to precisely express contracts (or agreements) as it is required in real world applications. We are currently working on the

incorporation of the SLAs in each layer of the cloud stack to our approach. To this end, work like [KL12] and [FPP12b] (at application and platform levels) and [BMDC⁺12] (at the infrastructure levels) are certainly good sources of inspiration for extending our approach to accommodate SLA definitions. This is an on-going work which is done in collaboration with another Ph.D student also from ASCOLA Research Group at the École des Mines de Nantes, in the context of the French Research Agency (Agence Nationale de la Recherche) funded project MyCloud.

Reconfiguration Reliability and Transactions. The reliability of operations becomes a critical issue when dealing with concurrent but interdependent autonomic managers. In particular, in the context of this thesis work, there is no guarantees that the actions performed by all autonomic managers involved in a given reconfiguration process actually take effect. Moreover, in case of failure, there is no roll back mechanisms that undo the already performed actions, which lead involved managed systems to inconsistent states. This issue has been already addressed in the context of dynamic reconfiguration of component-based architectures [LLC10] and seems to be a good start point for addressing this issue in the context of this thesis work.

Extension in the Interface between Application and Infrastructure Managers. Currently in this work, the interface between AMs and IM is limited to request/release VMs actions, without giving more details regarding the requested VMs such as the application-level constraints on them. We envision several situations in which only request/release operations are not enough and some requirements specific to each application should be translated into constraint at the infrastructure level. For instance, it would be reasonable to think that one VM should never be co-hosted (in the same PM) along with another VM (for replication purposes), or even the opposite (for network latency reasons). To this end, we believe that BtrPlace [HLM13] can be a good start point, since it proposes an approach for high level and flexible constraint definitions in terms of VM placement that can be done at application level and applied at the infrastructure level.

Stability and Reconfiguration Cost. In this thesis work, the cost implied by the execution of a reconfiguration was not taken into account for the decision making. While some actions, such as for binding or removing components, may have low cost in terms of execution time and overhead, others like for deploying a component or creating a VM can have non neglected cost and should vary according to the component's or VM's size. As a result, it may happen that by the time the reconfiguration takes effect it is no longer useful or that at the meantime another (conflicting) event arrives, requiring that the autonomic manager undo the previous reconfiguration. There are many tracks that could be investigated to mitigate this instability. First, a reconfiguration cost model should be taken into consideration in the optimization problem formulation. This is important because based on such a model one can analyze whether or not it is advantageous to perform the reconfiguration for a given runtime context. Also, a predictive approach would be necessary to profile the activity of managed systems (e.g. workload) in order to ease the identification of periods of instability. The event prioritization along with the expiration of out-dated events seems also to be needful, since it would allow the autonomic managers to choose for instance the newest one among two or several conflicting events (e.g. one detecting a workload increase and another workload decrease). Finally, autonomic managers preemption during an event processing could enable the countermand of a reconfiguration upon the arrival of a conflicting event, which can avoid undesirable future states or instability. For this purpose, a transactional approach, as previously discussed, would be of a great utility. All of these possible tracks should be investigated as future work.

Appendices



Resumé

A.1 Introduction

Au cours de ces dernières années, la consommation d'électricité liée à la technologie de l'information et de la communication (TIC) a remarquablement augmenté [Koo07, Koo11]. Il y a plusieurs raisons pour cette croissance. La démocratisation de l'Internet aussi bien que la popularisation de l'informatique en nuage [AFG⁺09, BYV⁺09, Mon12] ont motivé la migration des logiciels locaux vers les services accessibles à distance, disponible 24/24 et 7/7, et facturés à l'usage. Par conséquent, nous avons assisté à une croissance du nombre de nouveaux centres de données partout dans le monde.

Paradoxalement, les modèles de l'informatique en nuage [Pet11] peuvent être vus comme un outil très puissant pour faire face à la question de la consommation d'énergie. Du point de vue du fournisseur d'infrastructures, des innovations techniques telles que la virtualisation [SN05] permettent la mutualisation des ressources d'une machine physique (PM) en plusieurs machines virtuelles (VMs). En conséquence, la charge de travail de plusieurs applications peut être concentrée dans un nombre minimum de PMs, ce qui permet d'arrêter celles qui sont sous-utilisées et ainsi de réduire la consommation d'énergie [HLM⁺09]. Du point de vue du fournisseur d'applications, l'informatique en nuage permet de demander / relâcher des ressources à la volée. Cela signifie que le fournisseur d'infrastructures peut fournir des ressources de calcul et de stockage d'une manière souple et élastique, tandis que les applications sont facturées uniquement pour ce qu'elles consomment. Cela est particulièrement intéressant pour les applications qui doivent faire face à une charge de travail très variable, car elles peuvent ajuster dynamiquement la quantité de ressources nécessaires pour fournir le niveau désiré de qualité de service (QoS).

Le besoin de rendre les systèmes en nuage plus réactifs et autonomes aux modifications de l'environnement est l'une des principales raisons de l'adoption de *Autonomic Computing* [KC03]. En effet, l'*Autonomic Computing* rend les applications capables de réagir à une charge de travail très variables en ajustant dynamiquement la quantité de ressources nécessaires pour s'exécuter tout en gardant sa QoS [KCD⁺07, VTM10, APTZ12, CPMK12, YT12], alors que du point de vue du fournisseur d'infrastructures, l'*Autonomic Computing* permet à l'infrastructure de réagir rapidement aux changements de contexte (par exemple l'augmentation / diminution de l'utilisation des ressources physiques) en optimisant l'allocation des ressources et ainsi réduire les coûts liés à la consommation d'énergie [HLM⁺09, LLH⁺09, BB10, FRM12].

A.1.1 Problématique

Malgré les efforts récents pour la fabrication de matériels informatiques plus efficaces en termes de consommation énergétique [Bar05], au jour d'aujourd'hui les PMs consomment encore beaucoup d'énergie même quand elles sont inactives [BH07]. Par conséquent, la façon dont les applications sont déployées sur les VMs peuvent interférer indirectement sur la consommation d'énergie de l'infrastructure. En effet, les fournisseurs d'applications effectuent des actions telles que de demander/libérer de ressources informatiques/stockage dans le but d'optimiser le compromis entre la QoS et le coûts attribués à l'utilisation des ressources. Ces actions peuvent mettre l'infrastructure dans un état qui réduit le taux d'occupation des infrastructures et par conséquent de l'efficacité énergétique. En d'autres termes, les décisions guidées par la QoS, prises par les applications peuvent indirectement interférer dans la consommation d'énergie de l'infrastructure sous-jacente.

Pour que la synergie entre les applications et infrastructure soit efficace, il est nécessaire que les applications soient assez flexibles afin de réagir aux événements produits non seulement par l'application elle-même (par exemple, augmentation/diminution de la charge client), mais aussi par l'infrastructure (par exemple les changements de frais de location de ressources et/ou des contrainte de ressource). De ce fait, équiper les applications des mécanismes de reconfiguration leur permettrait d'être plus sensibles aux changements d'infrastructure et donc plus adaptable à un plus large éventail de situations. Dans l'autre sens, les applications peuvent être encouragées par le fournisseur d'infrastructure à occuper une certaine partie de l'infrastructure de manière à améliorer le taux d'occupation (et par conséquent l'efficacité énergétique).

A.1.2 Contributions

Cette thèse porte sur la gestion autonome des applications et des infrastructures pour améliorer les QoS et de l'efficacité énergétique dans l'informatique en nuage.

Nous croyons que la gestion de chaque application ainsi que l'infrastructure dans un seul système autonome est une tâche très difficile. Chaque application peut avoir ses propres objectifs et exigences, ce qui affecte par conséquent la façon dont leur gestionnaires doivent être conçus (par exemple en ce qui concerne l'échelle de temps de réactivité). Par ailleurs, il est courant dans l'informatique en nuage que les applications et l'infrastructure appartiennent à différentes organisations, ce qui implique que les détails des applications devraient pas nécessairement être visible au niveau de l'infrastructure. De la même manière, les applications ne devraient pas non plus être au courant des détails de l'infrastructure. De ce fait, nous proposons une approche pour la gestion de plusieurs gestionnaires autonomes, dans laquelle chaque application est équipée de son propre gestionnaire en charge de déterminer la quantité des ressources informatiques nécessaires pour offrir la meilleure QoS possible. Un autre gestionnaire, au niveau de l'infrastructure, gère les ressources physiques selon la consommation d'énergie.

Afin de gérer le manque de synergie entre les applications et l'infrastructure, nous proposons une approche pour la synchronisation et la coordination des gestionnaires autonomes. L'objectif est de promouvoir une synergie entre les applications et l'infrastructure pour que l'information d'un gestionnaire puisse être pris en compte plus facilement par d'autres gestionnaires de manière à éviter les interférences négatives.

Enfin, nous étendons les capacités de l'élasticité des ressources (demande/libération des ressources de calcul ou de données) en traitant les applications comme des *boîtes blanches*. Autrement dit, nous supposons que la façon dont les applications sont composés (c'est à dire les composants utilisés et comment ils sont connectés) peut conduire à différentes configurations architecturales ayant des besoins différents en termes de ressources. En conséquence, les applications deviennent plus sensibles et réactives à l'évolution de l'environnement d'exécution, peu importe si les changements se produisent dans l'application elle-même (par exemple, une augmentation considérable de la charge de travail avec un budget limité) ou dans l'infrastructure (par exemple, une restriction des ressources en raison de contraintes énergétiques).

A.2 État de l'art

Cette section présente une sélection des travaux pertinents liés au thème de cette thèse. Les travaux ont été classifiés en quatre catégories. Les approches liées à la couche applicative prennent en considération les caractéristiques des applications (QoS ou fonctionnalité) afin d'élaborer des stratégies pour réduire la consommation d'énergie (ou l'utilisation de ressources et par transitivité l'énergie) au niveau de l'infrastructure sous-jacente. Les approches liées à la couche d'infrastructure profitent de la flexibilité offerte par le réglage du matériel possible grâce à des techniques telles que la mise à l'échelle dynamique de la fréquence et tension (*Dynamic Voltage Frequency Scaling (DVFS)*) ou la virtualisation des serveurs (placement de la machine virtuelle, consolidation, etc) de manière à améliorer la consommation de ressources et par conséquent l'efficacité énergétique. Les approches transversales prennent en compte à la fois les aspects de l'infrastructure et ainsi que ceux des applications afin d'avoir des solutions globales en termes de qualité de service (QoS) (au niveau de l'application) et l'efficacité énergétique au niveau de l'infrastructure. Enfin, les approches de gestion multi-autonomes comptent sur plusieurs gestionnaires autonomes qui peuvent (ou non) être coordonnés pour atteindre leurs objectifs. Dans notre cas, les objectifs sont d'améliorer la QoS au niveau de l'application, tout en optimisant l'efficacité énergétique au niveau de l'infrastructure. Tous les domaines de travail connexes mentionnés précédemment sont résumés dans la figure A.1.

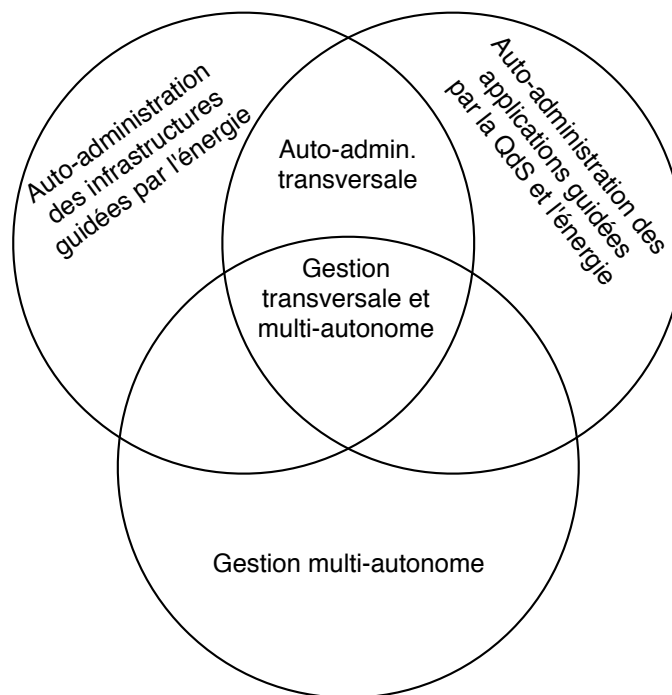


Figure A.1: Intersection des travaux connexes.

Afin d'évaluer de façon concise les travaux décrits précédemment, nous définissons certains attributs de comparaison:

- **Adaptation fonctionnelle :** cet attribut indique qu'au niveau applicatif, les applications sont capables de modifier leurs architectures de manière de s'adapter aux changements dans l'environnement d'exécution. Ces modifications architecturales peuvent avoir un impact sur leurs fonctionnalités. En conséquence, le degré d'utilité de chaque fonctionnalité possible doit être pris en compte.
- **Adaptation non-fonctionnelle :** il peut y avoir des changements (architectural ou non) au niveau applicatif qui peut avoir un impact sur les attributs de QoS tels que le temps de

réponse, débit, etc. Par exemple, les décisions prises par une application afin d'augmenter les ressources allouées pour elle peut conduire à l'amélioration des performances. Inversement, la QoS peut être dégradée si les ressources diminuent.

- **Multi-application** : cet attribut fait référence à la possibilité de traiter avec plusieurs applications *Software-as-a-Service (SaaS)* partageant la même infrastructure *Infrastructure-as-a-Service (IaaS)*. Comme nous nous concentrons sur les environnements en nuage, il est impératif que la solution tienne compte de la gestion des multiples applications.
- **Transversale** : cet attribut indique que l'approche prend en considération les aspects à la fois des applications et de l'infrastructure afin de traiter le problème de la QoS et la consommation de ressources / de l'énergie. Par exemple, une approche qui effectue à la fois l'allocation de ressources guidé par la QoS (au niveau applicatif) et la consolidation des serveurs (au niveau de l'infrastructure) est considéré comme étant transversale.
- **DVFS/Arrêt (On-off) de PM** : pour les approches au niveau de l'infrastructure, les stratégies d'économie d'énergie ou de ressources peut être matérialisée soit par DVFS, en éteignant des PMs ou une combinaison des deux. D'une part, DVFS offre la possibilité de réduire la consommation d'énergie avec une faible surcharge du système par ajuster la tension et la fréquence du processeur. D'autre part, en raison du manque de proportionnalité énergétique des systèmes matériels [BH07], cette approche reste encore limitée en l'absence de charge de travail dans un ensemble de PMs donné, ce qui est le cas dans de nombreux centres de données [Vog08]. Par conséquent, bien qu'il soit coûteux (par rapport à DVFS) d'arrêter / d'allumer une PM, les gains en énergie peuvent être beaucoup plus élevés. La solution idéale serait de combiner les deux options.
- **Multi-autonome** : cet attribut correspond à la capacité des systèmes de s'administrer en s'appuyant sur plusieurs gestionnaires autonomes, ce qui peut exiger une sorte de coordination entre eux pour atteindre le résultat souhaité.
- **Couplage lâche** : parfois, dans une approche transversale en couches, les détails d'une couche sont pris en compte par d'autres couches, peu importe si l'approche a un seul ou plusieurs gestionnaires autonomes. Comme cette thèse se concentre sur les environnements en nuage, dans lequel les acteurs interagissent d'une manière indirectement via des interfaces bien définies, nous considérons le couplage lâche comme une caractéristique importante pour la comparaison.

Le tableau A.1 résume le travaux connexes à l'égard des attributs de comparaison mentionnés ci-dessus, où X , $-$ et n/a correspondent respectivement à une caractéristique mise en œuvre, pas mis en œuvre, et qui ne s'applique pas à l'approche en question.

Parmi les travaux qui prennent en considération uniquement la couche applicative, tous sont en mesure d'adapter les aspects non-fonctionnels (QoS) en fonction des changements dans l'environnement. Cependant, seulement [PDPBG09] et [YT12] présentent une approche multi-application dans laquelle les applications sont capables de changer leur fonctionnalité via des configurations alternatives pour faire face aux changements dans l'environnement. Parmi les travaux qui font de l'optimisation de l'énergie seulement au niveau de l'infrastructure, uniquement [BB10] effectue à la fois DVFS et arrêt de PMs, le reste des travaux [LLH⁺09, HLM⁺09, HLM⁺09] s'appuient sur la consolidation des VMs et donc l'arrêt de PMs. Parmi les travaux qui tiennent compte à la fois les détails des applications et de l'infrastructure afin d'optimiser la QoS des applications et la consommation d'énergie des infrastructures, tous sont en mesure d'adapter la QoS (aspects non-fonctionnels) de façon à utiliser moins de ressources et par conséquent moins d'énergie. Seulement [GDA10, DAB11] fournissent des moyens d'adapter les applications afin de fournir d'autres configurations. En ce qui concerne les stratégies énergétiques, seulement [PCL⁺11] et [ISBA12] permettent d'effectuer les deux : DVFS et l'arrêt de PM, tandis que les autres ne comptent que sur l'arrêt de PM. Enfin, parmi les travaux qui sont transversaux et multi-autonome, [RRT⁺08]

est considéré comme une approche multi-autonome au sein d'un centre de données pour avoir un contrôle à plusieurs niveaux : des VM, des PM ou des tiers. Cela signifie que l'approche ne tient pas compte explicitement des aspects du niveau applicatif afin d'optimiser la consommation d'énergie dans le centre de données. [KLS⁺10] et [CFF⁺11] fournissent des solutions de gestion inter-couches et multi-autonome dans lequel les applications sont capables d'adapter leurs fonctionnalités ainsi que leurs aspects non-fonctionnels. Alors que [KLS⁺10] est aussi capable de gérer plusieurs applications, [CFF⁺11] considère des scénarii avec une seule application. Cependant, la stratégie d'économie de l'énergie dans [KLS⁺10] ainsi que comme dans d'autres approches [MKGdPR12, WW11, KCD⁺07] est uniquement basée sur DVFS, qui peut être relativement limitée. Enfin, concernant le couplage lâche, seulement [KLS⁺10], [CPMK12] et [WW11] fournissent des solutions dans lesquelles les gestionnaires autonomes travaillent en couplage lâche. C'est particulièrement intéressant parce que, dans le contexte de l'informatique en nuage, les détails d'un gestionnaire d'une couche donnée ne doit pas être visible à un autre gestionnaire situé à une couche différente.

Afin de surmonter les limitations mentionnées précédemment, cette thèse présente une approche inter-couches pour l'optimisation de la QoS des applications et l'efficacité énergétique de l'infrastructure dans le cadre de l'informatique en nuage. L'optimisation est obtenue grâce à la coordination de multiple gestionnaires autonomes qui communiquent en couplage lâche. Au niveau de l'applicatif, chaque application est équipée de son propre gestionnaire qui est capable d'adapter non seulement les aspects non-fonctionnels (QoS), mais aussi ses fonctionnalités de manière à être plus souple aux changements dans l'environnement d'exécution. Au niveau de l'infrastructure, un autre gestionnaire cherche à réduire le nombre de PMs (et par conséquent la consommation d'énergie) nécessaire pour répondre aux demandes de toutes les applications.

A.3 Un modèle multi-autonome pour l'informatique en nuage

Cette section présente un modèle de coordination de plusieurs gestionnaires autonomes, qui est composé d'un protocole de coordination et d'une base de connaissance publique qui nécessite un mécanisme de synchronisation. Tout d'abord, nous présentons le modèle architectural en mettant en évidence les principales exigences imposées par des modèles de services de l'informatique en nuage et comparons avec les modèles de l'état de l'art. Ensuite, nous décrivons le modèle de base de connaissance et le mécanisme de synchronisation sur lesquelles nous nous appuyons. Enfin, nous décrivons le protocole de coordination qui est basé sur un ensemble d'actions et d'événements inter-gestionnaires.

A.3.1 Modèle architectural

Les services en nuage permettent un accès facile aux ressources de calcul (par exemple des applications logicielles ou de l'infrastructure) d'une manière encapsulée et transparente, ce qui signifie que les détails internes sur l'état ou le comportement de ces ressources sont cachés aux consommateurs de services. C'est pourquoi, nous nous intéressons à des modèles architecturaux pour l'interaction entre gestionnaires autonomes qui répondent aux contraintes inhérentes des modèles de services en nuage à l'égard du partage d'information et le couplage lâche.

Selon l'état de l'art sur les architectures auto-adaptatives [Lem13], l'interaction entre gestionnaire autonomes peut suivre un ensemble de patrons, comme il est illustré dans la figure A.2.

Dans le patron *hiérarchique*, des boucles de contrôle MAPE-K [KC03] (de l'anglais *monitoring, analysis, planning, execution* et *knowledge*) complètes sont organisées dans une hiérarchie, dans laquelle les boucles (ou gestionnaires) à des niveaux inférieurs fonctionnent à des échelles de temps courts pour résoudre des problèmes locaux, alors que celles des niveaux supérieurs travaillent à des échelles de temps longs et possèdent une vision globale des problèmes grâce à des messages passés depuis les boucles aux niveaux inférieurs. Cette vision globale n'est pas souhaitable dans l'informatique en nuage en raison de limites organisationnelles, c'est à dire que les gestion-

	Travaux	Adaptation Fonctionnelle		DVFS / Arrêt de PM	Multi-application	Gestion Lâchement Couplée	Année	
Application	[MFKP09]	-	X	n/a	-	n/a	2009	
	[PDPBG09]	X	X	n/a	-	n/a	2009	
	MoKa [AB10]	-	X	n/a	-	n/a	2010	
	[PPMM11]	-	X	n/a	-	n/a	2011	
	[Fer11]	-	X	n/a	-	n/a	2011	
	[YT12]	X	X	n/a	X	n/a	2012	
Infra	EnaCloud [LLH ⁺ 09]	n/a	n/a	On-off	X	n/a	2009	
	Entropy [HLM ⁺ 09]	n/a	n/a	On-off	X	n/a	2009	
	[BB10]	n/a	n/a	On-off / DVFS	X	n/a	2010	
	Snooze [FRM12]	n/a	n/a	On-off	X	n/a	2012	
Transversale	Gest. unique	[VTM10]	-	X	On-off / DVFS	X	-	2010
		SAFDIS [GDA10, DAB11]	X	X	-	X	-	2010/2011
		[PCL ⁺ 11]	-	X	On-off / DVFS	X	-	2011
		[APTZ12]	-	X	On-off / DVFS	-	-	2012
		CloudPack [ISBA12]	-	X	On-off	X	-	2012
	Multi-gest.	[KCD ⁺ 07]	-	X	DVFS	-	-	2007
		[RRT ⁺ 08]	-	-	Both	-	-	2008
		[KLS ⁺ 10]	X	X	DVFS	X	X	2010
		[CPMK12]	-	X	-	X	X	2011
		Co-Con [WW11]	-	X	DVFS	X	X	2011
[CFF ⁺ 11]	X	X	On-off / DVFS	-	-	2011		
[MKGdPR12]	-	X	DVFS	-	-	2012		

Table A.1: Sommaire des travaux connexes.

naires risquent être déployés pour gérer les systèmes appartenant à différents fournisseurs (par exemple dans les nuages hybrides / public).

Les patrons *maître / esclave* ou *régionaux* nécessitent un module de décision unique (analyse et/ou planification) pour toutes les boucles, c'est à dire pour toutes les applications et l'infrastructure. Ce modèle implique un manque de flexibilité dans l'autonomie de chaque gestionnaire (par exemple, les applications qui sont gérées des échelles de temps différentes) car tous les gestionnaires sont créés par / déployé par le même fournisseur, ce qui n'est souvent pas le cas dans les environnements en nuage.

Le patron *décentralisé* exige que chaque tâche autonome d'un gestionnaire soit coordonnées avec son homologue dans le gestionnaire paire. Alors qu'il peut être très intéressant pour faire face avec des comportements contradictoires, ce modèle ne parvient pas à fournir un couplage lâche, car le nombre de points de communication entre les gestionnaires est plus grand que dans les autres patrons, ce qui rend par conséquent les gestionnaires plus enchevêtrés.

Le patron *partage d'information* améliore l'absence de dissimulation d'information observée dans le modèle décentralisé, car il permet la communication entre gestionnaires seulement via les

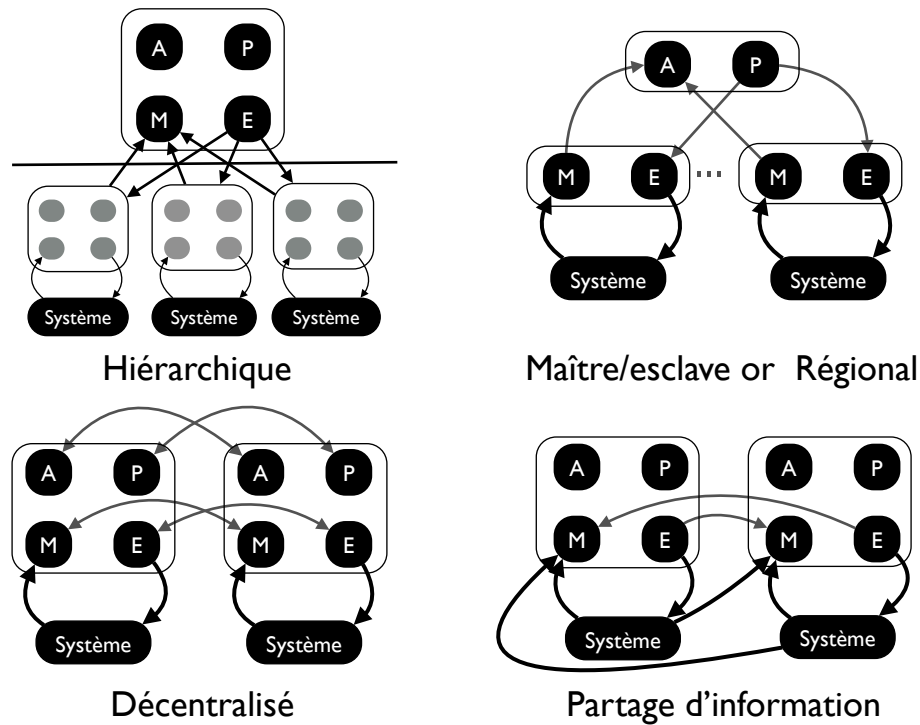


Figure A.2: Patrons de coordination de gestionnaires autonomes.

moniteurs et les exécuteurs. Cependant, il permet un partage des données de surveillance parmi toutes les tâches de surveillance, ce qui peut ne pas être applicable dans des environnements en nuage. Par exemple, il n'est pas concevable que les applications soient au courant des détails de la mise en place des VMs dans les PM.

Compte tenu de ces limitations, nous proposons un modèle d'architecture autonome (cf. figure A.3) pour la coordination des multiples gestionnaires qui répond aux contraintes architecturales de l'informatique en nuage en termes de couplage lâche et la dissimulation de l'information. Dans ce modèle, les gestionnaires communiquent strictement via leurs interfaces naturelles (tâches de surveillance et d'exécution), ce qui permet de restreindre les points d'interaction (inter-gestionnaires) et d'augmenter ainsi le couplage lâche entre les tâches autonomes. En ce qui concerne la dissimulation d'information, contrairement aux modèles précédents dans lesquels les données de surveillance sont partagées entre les moniteurs de tous les gestionnaires, dans le modèle proposé, la visibilité sur le contexte d'exécution d'un système administré quelconque est limitée au gestionnaire autonome correspondant. En effet, un gestionnaire peut partager une partie de sa connaissance avec d'autres gestionnaires afin qu'ils puissent prendre des décisions qui sont plus adéquates à l'état actuel des gestionnaires impliqués. Par exemple, les fournisseurs des IaaS peuvent s'appuyer sur une approche axée sur les prix afin d'influencer le comportement des consommateurs (par exemple, les fournisseurs de SaaS ou PaaS), c'est à dire que des promotions (soldes) peuvent être créées pour stimuler les consommateurs à occuper une certaine zone de l'infrastructure et ainsi améliorer le taux d'utilisation. Cette information dynamique quant aux changements de prix de locations de ressources doit être partagée entre les gestionnaires impliqués, notamment ceux qui contrôlent les IaaS/PaaS et ceux qui contrôlent les SaaS/PaaS.

A.3.2 Protocole de synchronisation pour l'accès à la base de connaissance publique

La dissimulation d'information est une caractéristique importante des modèles à base de services, et plus particulièrement des modèles de services dans le nuage. Néanmoins, le manque de visibilité globale du contexte d'exécution des services peut conduire les gestionnaires à prendre des

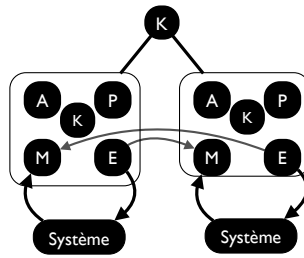


Figure A.3: Modèle architectural proposé pour la coordination des multiples gestionnaires autonomes.

décisions qui en quelque sorte impactent négativement les systèmes administrés par d'autres gestionnaires.

Afin d'avoir une analyse mieux contextualisée et à respecter le principe de dissimulation d'information, la base de connaissance de chaque gestionnaire est divisé en deux parties: la connaissance privé (*private knowledge*), qui stocke les informations internes nécessaires pour les tâches autonomes, et la connaissance du publique (*public knowledge*), qui est partagée avec d'autres gestionnaires. La base de connaissance publique peut être synchronisée si les actions exécutées par les gestionnaires nécessitent de modifier (directement ou indirectement) l'information partagée. En effet, les actions simultanées de multiples gestionnaires peuvent conduire à des changements dans la base de connaissance en même temps (problème de concurrence) et peut conduire à des résultats globaux non-logiques (problème de cohérence). Dans notre approche, nous considérons que le gestionnaire ayant une base de connaissance publique est le seul qui a le droit de la modifier directement, ce qui évite les problèmes de concurrence.

La base de connaissance publique est composée de sections critiques et non critiques. Les premiers correspondent à des éléments d'informations qui sont rarement modifiées et ne qui ne compromettent pas la cohérence du système, tandis que l'autre se réfère à des éléments d'information qui peuvent être changés fréquemment, donc qui peuvent interférer dans la cohérence du système. Ainsi, alors que la synchronisation est facultative pour accéder aux sections non critiques, elle est obligatoire pour accéder à celles qui sont critiques.

Pour cette raison, nous introduisons un protocole à base de jetons afin de synchroniser l'accès aux sections critiques entre plusieurs gestionnaires. Comme pour les transactions dans les bases de données, ce protocole de synchronisation assure que seul un gestionnaire peut accéder à une section critique à la fois. Chaque section critique est associé à un jeton, ce qui signifie que, pour accéder à une section critique donnée, les gestionnaires doivent demander le jeton correspondant. Un gestionnaire peut obtenir le jeton soit en demandant explicitement via un message *TOKEN REQUEST* (demande active de jeton), soit via un message de transfert de jeton (*TOKEN TRANSFERT*) d'un gestionnaire vers un autre (réception passive de jeton). Chaque fois qu'un gestionnaire n'a plus besoin de jeton, il le libère via un message *TOKEN RELEASE*. Chaque fois qu'un jeton est demandé, le demandeur doit attendre un message de jeton acquis *TOKEN ACQUIRED*. Chaque gestionnaire autonome ayant une connaissance publique avec des sections critiques met en œuvre un gestionnaire de jetons qui est en charge de la gestion du protocole de jeton.

A.3.3 Protocole de coordination dirigé par des évènements

L'objectif du protocole de coordination est de fournir des moyens pour les gestionnaires autonomes de communiquer afin que les actions soient exécutées de façon coordonnée et de mettre en place une synergie entre les gestionnaires. Dans le modèle architectural proposée, la communication entre les gestionnaires est achevée à travers des tâches de surveillance et d'exécution, qui sont déjà utilisées comme interfaces avec le système administré. Par conséquent, ces gestionnaires sont moins enchevêtrés les uns avec les autres et donc plus faiblement couplés.

Afin de clarifier les interactions entre plusieurs gestionnaires, il est important de différencier les actions et les événements qui font partie du système administré et ceux qui font partie du modèle architectural multi gestionnaires. La figure A.4 illustre la hiérarchie des différents types d'événements et d'actions.

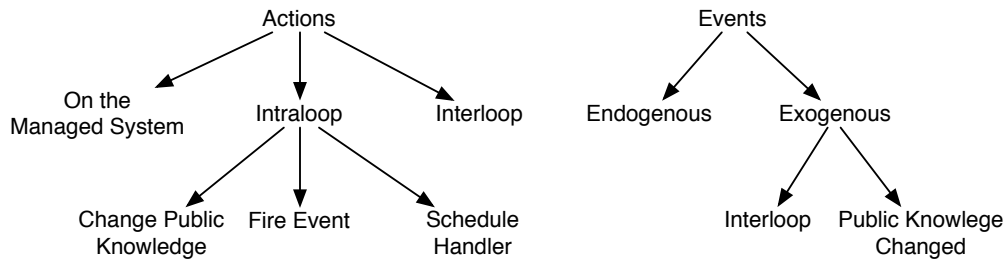


Figure A.4: Hiérarchie d'actions et d'évènements autonomes.

Les actions peuvent être exécutées sur le système administré (*Action on the managed system*), dans la boucle de contrôle MAPE-K (*intraloop*) ou sur une autre boucle de contrôle MAPE-K (*interloop*). Les actions sur le système administré et les actions *interloop* sont toujours exécutées par la tâche d'exécution de la boucle de contrôle MAPE-K. Une action *interloop* peut notifier un autre gestionnaire autonome comme s'il demandait un service et attendait la réponse. Toutefois, afin d'éviter les états de blocage, les gestionnaires doivent communiquer de manière asynchrone. Dans ce cas, la tâche de planification crée un conteneur (*handler*) avec toutes les autres actions qui doivent être exécutées en réponse à cette action *interloop*. Cette action *interloop* est donc une action de notification, qui est considéré comme un événement inter-boucle (*interloop*) pour la boucle de contrôle destinataire.

Enfin, les actions *intraloop* peuvent être pour modifier le contenu de la base de connaissance publique (*Changer Public Knowledge*), pour programmer les actions d'un conteneur (*Schedule Handler*) qui dépend d'un ou plusieurs objets futurs [EFGK03], ou pour déclencher un événement endogène (*Fire Event*) sur le même gestionnaire autonome. Cette action peut être instantanée ou temporisée (soit déclenchée après un intervalle de temps donné).

Comme il est illustré dans la figure A.4, un événement peut être soit endogène ou exogène. La source d'événements endogènes est toujours le système administré. Alors que la source d'événements exogènes est un autre gestionnaire autonome. Pour les événements exogènes, il y a les événements inter-boucle (*interloop*) - créés par les actions inter-boucle (*interloop*) - et les événements qui indique qu'une base de connaissance publique a été modifiée (*Public Knowledge Changed*) - déclenché suite à une action *Change Public Knowledge*.

A.4 Une approche pour la gestion multi-autonome pour l'informatique en nuage

Cette section détaille l'approche proposée pour l'auto-adaptation des services en nuage. Nous présentons d'abord un aperçu de l'architecture globale, qui englobe les couches SaaS et IaaS de la pile classique des service en nuage. Ensuite, nous fournissons plus de détails sur la façon dont l'auto-adaptation a été réalisée dans chaque couche.

A.4.1 Vue architecturale globale

Basé sur le modèle proposé dans la section précédente, nous introduisons une approche composé de multiple gestionnaires autonomes mis en œuvre avec la boucle de contrôle MAPE-K pour permettre l'auto-adaptation des applications et des infrastructures en nuage. Comme on peut le voir sur la figure A.5, chaque application est équipée de son propre gestionnaire (*Application Manager*

- AM), qui est responsable de la gestion de l'élasticité architecturale et des ressources, ce qui signifie qu'il détermine la meilleure configuration architecturale (composition des composants (C) parmi d'autres configurations) et le montant minimum de ressources informatiques (en terme de machines virtuelles - VM) nécessaires pour offrir la meilleure qualité de service (QoS) possible pour une certaine charge de travail. Au niveau de l'infrastructure, un autre gestionnaire (Infrastructure Manager - IM) gère les ressources (machines physiques - PM et VM) d'une manière consciente en termes de consommation de ressources / énergie. Plus précisément, ce gestionnaire est responsable du placement optimal des VMs (avoir le nombre maximum de VMs sur le nombre minimum de PMs).

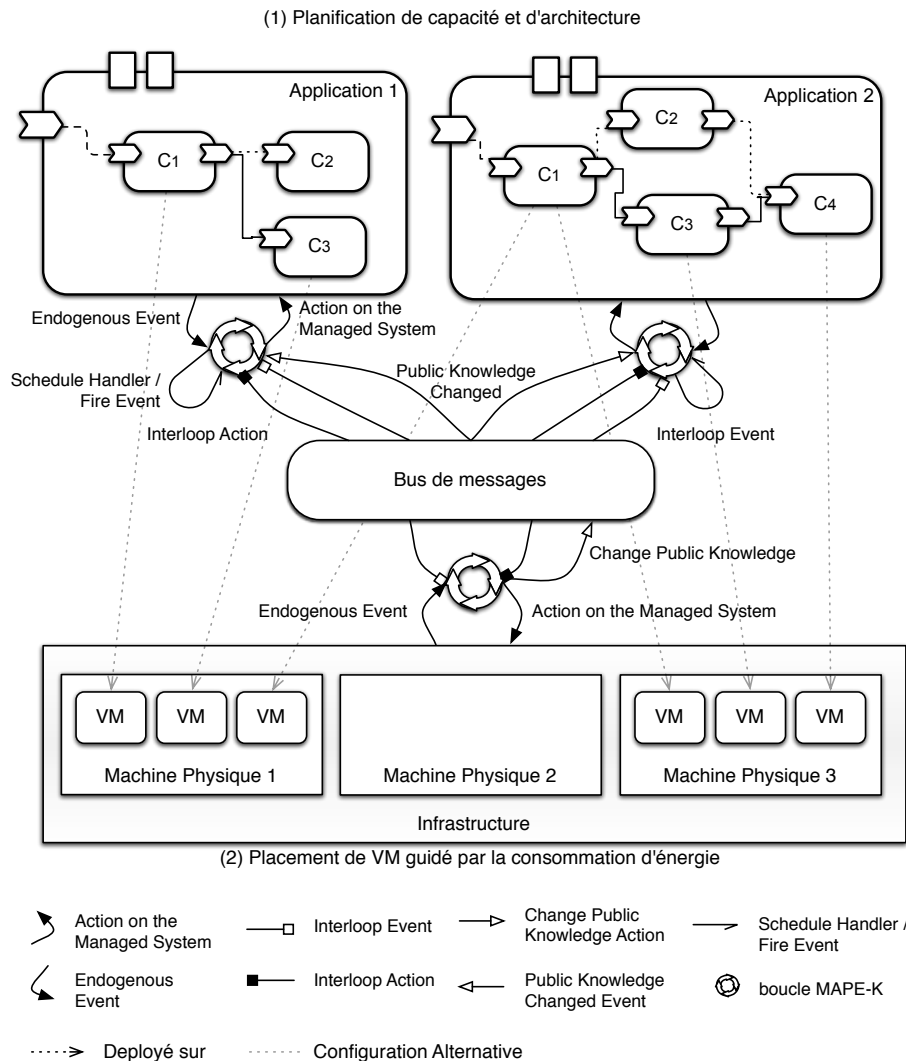


Figure A.5: Vue globale de l'architecture.

Le problème de la dépendance entre les systèmes est clairement définie : les composants logiciels (C) sont déployés sur des VMs, qui à leur tour, sont déployés sur l'infrastructure physique. Ainsi, les décisions prises par les gestionnaires AMs peuvent interférer avec l'infrastructure sous-jacente et par conséquent les décisions prises par le gestionnaire IM.

Pour résoudre ces problèmes, les gestionnaires doivent être capables de communiquer les uns avec les autres via des actions / événements inter-boucle (*interloop*). Ces actions / événements sont envoyés aux gestionnaires destinataires grâce à un bus de messages (*message-oriented middleware - MOM*). Les détails sur ces événements / actions ainsi que les modèles d'analyse utilisés pour déterminer les actions qui doivent être exécutées en réponse à un événement donné sont présentés dans les sections suivantes.

A.4.2 Gestionnaire d'infrastructure

L'IM est construit sur le modèle de référence MAPE-K dans le but de rendre l'infrastructure en nuage auto-administrable. En plus de la structure MAPE-K classique, l'IM se compose également d'une base de connaissance publique, qui partage certaines informations telles que les prix de location de VMs. Le reste de cette section détaille le modèle de connaissance, les interface en termes d'événements et d'actions (détecté et exécutés par les tâches de surveillance et exécution) et la tâche d'analyse qui est modélisée comme un ensemble de problèmes de satisfaction de contraintes (*Constraint Satisfaction Problem - CSP*).

Modèle de base de connaissance

La base de connaissance de l'IM est défini comme suit:

- $P = \{pm_i \mid i \in [1, p]\}$: l'ensemble de PMs.
- $pm_i = (pm_i^{cpu}, pm_i^{ram})$: les capacités de CPU et RAM de $pm_i \in P$.
- $A = \{a_i \mid i \in [1, \psi]\}$: l'ensemble d'applications.
- a_i^{rr} : le taux de réduction de ressource que l'application a_i accepte.
- $V = \{vm_i \mid i \in [1, v]\}$: l'ensemble de VMs en train d'exécuter.
- $vm_i = (vm_i^{cpu}, vm_i^{ram}, vm_i^{app})$: les capacités de CPU et RAM de vm_i .
- $vm_i^{app} \in A$: l'application propriétaire de la VM.
- $H_{p \times v}$ la matrice d'incidence, où $H_{ij} = 1$ signifie pm_i héberge vm_j , $H_{ij} = 0$ sinon.
- $M = \{m_i \mid i \in [1, q]\}$: l'ensemble de types/classes de VM offertes.
- $m_i = (m_i^{cpu}, m_i^{ram}, m_i^{cost})$: les capacités de CPU et RAM, et les prix de location pour la classe de VM $m_i \in M$.

L'IM peut décider de promouvoir des soldes (promotions Pr dans un laps de temps donné) pour les PMs dont le taux d'occupation est inférieur à un seuil donné de manière à stimuler les demandes pour occuper une partie des ressources sous-utilisées. Comme une promotion doit être appliquée à une seule application, elle doit être traitée comme une section critique. Par conséquent, la base de connaissance publique est composé d'une section non-critique contenant le prix de location de VM M et plusieurs sections critiques, chacun correspondant à une promotion Pr , comme suit:

- $Pr = \{pr_i \mid i \in [1, \xi]\}$: l'ensemble des promotions.
- $pr_i \in Pr = (pr_i^{pack}, pr_i^{price}, pr_i^{token})$
- $pr_i^{pack} = \{pack_{ij} \mid j \in [1, q]\}$: un pack de VMs pour la promotion pr_i .
- $pack_{ij} \in \mathbb{N}^*$: le nombre de VMs de classe m_j dans le pack.
- $pr_i^{price} = (1 - \delta) * \sum_{j=1}^q m_j^{cost} * pack_{ij} : \forall i \in [1, \xi]$: le prix global pr_i basé sur un taux fixe de réduction δ du prix normal de chaque VM du pack.

Événements et actions

La figure A.6 illustre les patrons événements-actions possibles pour le gestionnaire IM.

VMs Requested : c'est un événement du type *interloop* survenant quand un gestionnaire AM effectue une action du type *interloop Request VM*. Cet événement comprend un ensemble de VMs demandées au prix normal et éventuellement les promotions. Pour chaque promotion demandé, le jeton correspondant est transféré (TOKEN TRANSFERT) dans le message de telle sorte qu'un accès exclusif est garanti, ce qui signifie que l'AM avait déjà demandé et acquis un jeton (TOKEN REQUEST / TOKEN ACQUIRED) pour chaque promotion. Cet événement déclenche une tâche d'analyse qui évalue le placement des VMs demandées sur les PMs de manière à minimiser le nombre de PMs allumées nécessaires. La tâche de planification prend le résultat de l'analyse et traduit en un ensemble d'actions *Switch On PM*, *Create VM* et *Active Promotion* sur l'infrastructure. Enfin, il informe en retour l'AM que les VMs demandées sont prêtes par l'exécution de l'action *Notify VMs Created*, du type *interloop*.

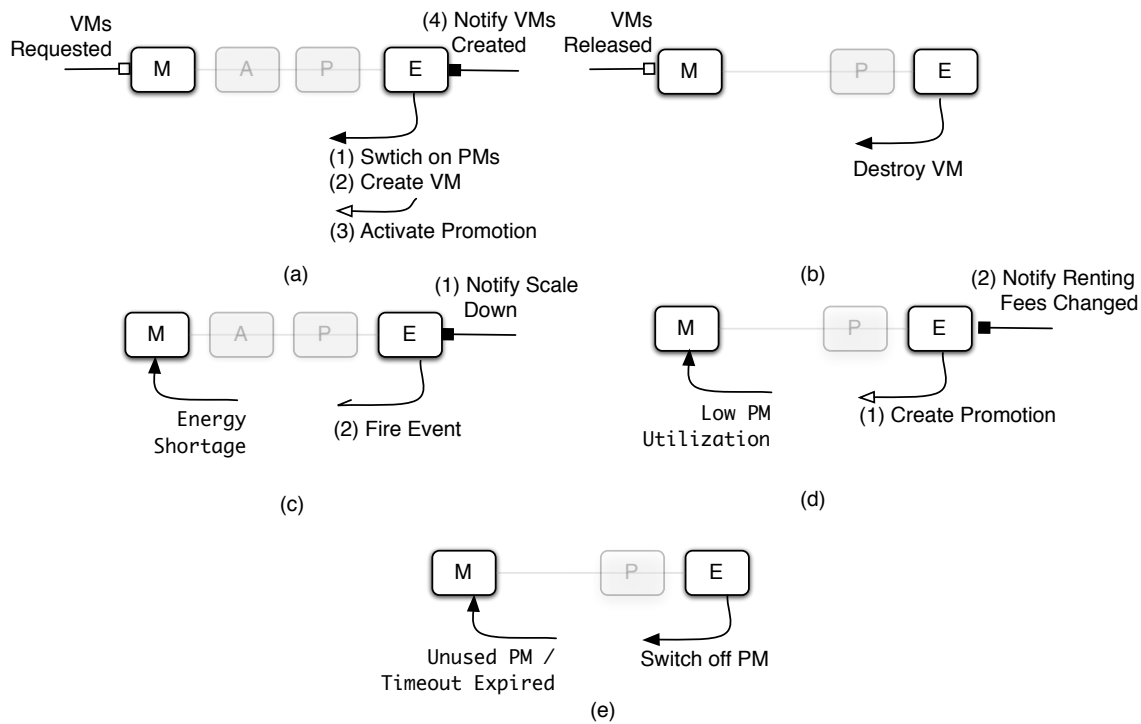


Figure A.6: Actions et événements du gestionnaire IM.

Release VMs : est aussi un événement du type *interloop* qui se produit lorsque un AM effectue une action *Release VM*, du type *interloop* (cf. section A.4.3). Ce événement contient la liste des VMs qui doivent être détruites. Comme il n'y a pas d'analyse particulière pour ce genre d'événement, la tâche de planification est déclenchée directement à partir de la surveillance. La tâche de planification traduit la liste des VMs à être libérées à travers d'un ensemble d'actions *Destroy VM* sur l'infrastructure.

Energy Shortage : c'est un événement du type *endogène* qui vient des données du contexte d'exécution de l'infrastructure (par exemple des sondes de consommation électrique). Cet événement comprend le nombre de PMs à arrêter $P^\#$. Suite à cet événement, l'IM déclenche la tâche d'analyse pour déterminer quelles PMs doivent être éteintes. La tâche de planification prend le résultat de la tâche d'analyse et le traduit en un ensemble d'actions *Notify Scale Down*, du type *interloop* afin d'informer les AMs concernés sur les contraintes qui pèsent sur l'infrastructure. Enfin, la tâche de planification crée une action *Fire Event* dont l'objectif est de déclencher l'événement endogène *Timeout Expired* de manière temporisée, c'est à dire après un délai donné. Cet événement contient l'ensemble de PMs à arrêter ($P^{off} \subseteq P$). Une fois que cet événement est détecté,

c'est à dire après le délai expiré, la tâche de planification prend la liste des PMs à arrêter et la traduit en un ensemble de d'actions *Switch Off PM* sur l'infrastructure (e).

Low PM Utilization : c'est un événement endogène de l'infrastructure qui est détecté chaque fois qu'une PM pm_i a été sous-utilisé (lorsque le taux d'occupation est inférieur à U_{min}) au cours d'une certaine période de temps t_l , comme il est formalisé dans l'équation A.1. Comme on peut le voir sur la figure A.6, la détection de cet événement déclenche directement la tâche de planification qui crée, pour chaque heure concernée, un ensemble d'actions *Create Promotion* qui crée des promotions (du type *Change Public Knowledge*), suivie par une action *NotifyRentingFeesChanged* (du type *interloop*, qui notifie les AMs souscrites que la base connaissance publique a été changée.

$$\min\left(\frac{\sum_{j=1}^v H_{ij} * vm_j^{ram}}{pm_i^{ram}}, \frac{\sum_{j=1}^v H_{ij} * vm_j^{cpu}}{pm_i^{cpu}}\right) < U_{min} \quad (A.1)$$

est vrai pendant t_l

PM Unused : c'est un événement endogène de l'infrastructure qui est détecté quand une PM n'a hébergé aucune VM pendant un certain temps t_u , tel qu'il est exprimé par l'équation A.2. Lors de la détection de cet événement, les PMs concernés ($P^{off} \subseteq P$) sont directement arrêtées. Comme il n'est pas nécessaire d'avoir une tâche d'analyse, la tâche de planification est directement déclenchée. Elle crée un ensemble d'actions *Switch PM Off* sur les infrastructures.

$$\min\left(\frac{\sum_{j=1}^v H_{ij} * vm_j^{ram}}{pm_i^{ram}}, \frac{\sum_{j=1}^v H_{ij} * vm_j^{cpu}}{pm_i^{cpu}}\right) = 0 \quad (A.2)$$

est vrai pendant t_u

Analyse

Les tâches d'analyse sont constitués des CSPs et sont modélisés et résolus avec de la programmation par contrainte (Constraint Programming - CP)[RVBW06]. Un modèle CP est composé d'un ensemble de variables (de décision), un ensemble de domaines (valeurs possibles pour chaque variable), et des contraintes sur ces variables. Dans le cas des problèmes d'optimisation, une fonction objectif est également définie.

Lors de la détection d'un événement dans la tâche de surveillance, l'IM peut déclencher la tâche d'analyse ou directement la tâche de planification ou de l'exécution, si aucune analyse n'est requise. Comme on peut le voir sur la figure A.6, la tâche d'analyse est déclenchée soit par un événement *VMs Requested* ou un événement *Energy Shortage*. Les tâches d'analyse utilisent soit l'analyseur *VM Placement Analyzer* ou *Energy Shortage Analyzer*.

Energy Shortage Analyzer : cet analyseur est utilisé lors de la détection d'un événement *Energy Shortage* et vise à trouver un ensemble de PMs à arrêter pour faire face à une situation de pénurie d'énergie. La solution doit répondre aux contraintes en termes de taux de réduction des ressources (A_k^{rr}) déclaré par chaque application (a_k) et le nombre minimum de PMs à arrêter $P^\#$, tout en minimisant l'impact de la restriction des ressources imposée aux applications.

Premièrement, nous définissons la variable de décision z_i , dont le domaine $D(z_i) \in \{0, 1\} \forall i \in [1, p]$. Cette variable indique si une PM $pm_i \in P$ doit être arrêté. La deuxième partie est constituée d'un ensemble de contraintes sur les variables de décision z_i . CR_k et RR_k sont des expressions auxiliaires correspondant respectivement à la quantité actuelle de ressources allouée à une application a_k et le montant des ressources à être libéré après arrêter les PMs spécifiées dans les variables z . L'équation A.3 indique que pour chaque application $a_k \in A$ le montant des ressources à libérer (RR_k) ne doit pas dépasser le montant des ressources actuellement allouées à cette application (CR_k) réduit de la ressource taux de restriction (a_k^{rr}). L'équation A.4 indique qu'au moins $P^\#$ PMs doivent être arrêtées afin de faire face à la pénurie d'énergie. Enfin, l'objectif est de minimiser

la restriction de ressources imposée aux applications (cf. équation A.5).

$$CR_k * (1 - a_k^{rr}) \geq RR_k, \forall i \in [1, \psi] \quad (\text{A.3})$$

$$\sum_{i=1}^p z_i \geq P^\# \quad (\text{A.4})$$

$$\text{minimize} \left(\sum_{i=1}^{\psi} RR_i \right) \quad (\text{A.5})$$

L'analyseur *VM Placement Analyzer* est déclenché suite à la détection d'un événement *VMs Requested* et son objectif est de placer les VMs demandées (à prix normaux ou réduits) dans les PMs de sorte que le nombre de PMs allumées nécessaires soit minimisé. Pour des raisons de place, nous ne détaillons pas le fonctionnement de cet analyseur dans ce résumé.

Planification

La tâche de planification peut être déclenchée après l'exécution de la tâche d'analyse ou directement après la détection d'un événement lors de la surveillance. La planification prend en considération l'état actuel de l'infrastructure et l'état désiré résultant de la surveillance ou de l'analyse pour créer actions de reconfigurations (voir la section A.4.2) de manière structurée.

Selon la figure A.6, il existe cinq planificateurs différents, chacun pour faire face à un événement différent. Tous prennent aussi en entrée un ensemble de variables et en sortie un plan (objet de type *Plan*). L'objet *Plan* agrège un ensemble d'activités, chacune contenant un ou plusieurs autres activités ou actions autonomes. Ces activités peuvent être exécutées en séquence (objet de type *Sequence*) ou en parallèle (un objet de type *Fork*).

L'algorithme 13 décrit le planificateur *Energy Shortage Planner*, qui est déclenché après la l'analyse effectuée par le *Energy Shortage Analyzer*, qui, à son tour, est déclenchée lors de la détection d'un événement *Energy Shortage*. Il prend en entrée l'ensemble des PMs (P^{off}) à arrêter, le délai au terme duquel celles-ci doivent être arrêtés (*timeout*), et le contexte d'exécution \mathcal{C} . Il commence par créer les objets *plan* et *fork* (lignes 1 et 2). Il parcourt ensuite l'ensemble des applications A tout en recueillant l'ensemble des VMs impliquées appartenant à la application en question (ligne 4). Cet ensemble est utilisé par l'action *NotifyScaleDown* (ligne 5) afin d'informer les applications qui devront libérer des VMs. Puis l'algorithme ajoute au plan des actions de notification (ligne 6), crée une action (*FireEvent*) qui va déclencher l'événement *TimeoutExpired* contenant les PMs à arrêter après le délai donné (ligne 7) et l'ajoute au plan. Lors de la détection de cette événement, le planificateur *Shutdown PM Planner* est déclenché. est décrit dans l'algorithme

Algorithm 13: *Energy Shortage Planner*.

Input: P^{off} : les PMs à arrêter.

Input: *timeout*: le temps après lequel les PMs devront être arrêtées.

Input: \mathcal{C} : le contexte d'exécution de l'infrastructure.

Result: *plan* : le plan résultant.

1 *plan* \leftarrow new instance of *Plan*

2 *notificationFork* \leftarrow new instance of *Fork*

3 **foreach** $app \in A$ **do**

4 // les VMs hébergées dans les PMs à arrêter.
 $vms \leftarrow \{vm_j \mid vm_j^{app} = app \wedge \forall i \in [1, p](\forall j \in [1, v](H_{ij} = 1 \wedge pm_i \in P^{off}))\}$
 5 $add(NotifyScaleDown(app, vms), notificationFork)$

6 $add(notificationFork, plan)$

7 $add(FireEvent(TimeoutExpired(P^{off}), timeout), plan)$

Virtual Machine Release Planner ce planificateur est déclenché lors de la détection d'un événement *VMs Released*. Il prend en entrée l'ensemble de VMs qui doivent être libérées (R^{vms}) et produit en sortie un ensemble d'actions qui vont effectivement détruire ces VMs.

Virtual Machine Placement Planner ce planificateur est utilisé pour faire face à des événements *VMs Requested* et déclenché après l'analyse *Virtual Machine Placement Analyzer*. Il prend en entrée l'ensemble de nouvelles VMs (V'), l'ensemble promotions demandées (R^{pr}) et la nouvelle matrice d'incidence VMs / PMs (h). Le planificateur produit comme sortie un ensemble d'actions qui vont créer les nouvelles VMs sur les PMs respectives et notifier l'AM demandeuse.

Promotion Planner ce planificateur est déclenché directement lors de la détection d'un événement *Low PM Utilization*. Il prend en entrée l'ensemble des PMs (P^{pr}) dans lesquelles les promotions doivent être créées, les prix de location de VMs (M) et le taux de réduction des promotions (δ). Le planificateur produit en sortie un ensemble d'actions qui créent les promotions dans la base de connaissance publique.

Physical Machine Shutdown Planner ce planificateur est déclenché directement lors de la détection d'un événement *Unused PM* ou d'un événement *TimeouExpired*. Cela prend en entrée le vecteur des PMs à arrêter P^{off} et produit en sortie un ensemble d'actions sur l'infrastructure pour éteindre les PMs en question.

Pour des raisons de simplicité, les algorithmes de ces planificateurs ne sont pas détaillés dans ce résumé.

A.4.3 Gestionnaire d'application

Semblable à l'IM, l'AM est également basé sur une boucle de contrôle MAPE-K. Mais, contrairement à l'IM, l'AM ne comprend qu'une base de connaissance privée et aucune base de connaissance publique. Cette section détaille cette base de connaissance, qui est composée d'un modèle d'application et le contexte d'exécution. Ensuite, nous décrivons les patrons événements-actions, les modèles CSP utilisés par les tâches d'analyse. Enfin, nous présentons les tâches de planifications utilisées dans ce gestionnaire.

Modèle de base de connaissance

La base de connaissance privé de l'AM est définie comme suit:

- $C = \{c_i \mid i \in [1, n]\}$: l'ensemble des composants applicatifs
- $c_i = (c_i^r, c_i^s)$. c_i^r : les dépendances fonctionnelles (*references*) et les services offerts du composant c_i
- $K = \{k_i \mid i \in [1, m]\}$: l'ensemble des configurations architecturales
- $k_i = (k_i^c, k_i^{deg}, k_i^{bind})$
- $k_i^c \subseteq C$: les composants figurant dans une configuration k_i
- $k_i^{bind} = \{(r, s) \mid r \in \bigcup_{i \in [1, n]} c_i^r, s \in \bigcup_{i \in [1, n]} c_i^s\}$: l'ensemble des connections (*bindings*) qui lient la référence d'un composant à un service offert par un autre composant.
- $k_i^{deg} \in [0, 1]$: la dégradation architecturale, c'est à dire la dégradation de la QoS due à la configuration architecturale actuelle (les composants et connections utilisés)
- $perf_i : \Lambda \times \mathbb{N}^* \times \mathbb{N}^* \mapsto \mathbb{N}^*$: la fonction performance pour un composant c_i sous une charge client donnée, et un montant de ressources CPU et RAM allouées au composant.

- $\Lambda = \{\lambda_1, \dots, \lambda_u\}$: l'ensemble de seuil de charge client
- $V = \{vm_i \mid i \in [1, v]\}$: l'ensemble de VMs allouées à l'application administrée
- rt^{ideal}, rt^{acc} : les seuils désirable et acceptable de temps de réponse
- $rt^{deg} \in [0, 1]$: la dégradation de la performance, i.e. la dégradation de la QoS liée au critère de performance quand le temps de réponse actuel vaut entre rt^{ideal} et rt^{acc}
- $CPR = \{cpr_i \mid i \in [1, t]\}$: l'ensemble des promotions pris par l'application
- $cpr_i = (cpr_i^{price}, cpr_i^{pack})$: le prix total (avec réduction) de la promotion et le pack de VMs faisant partie de cette promotion.
- $budget$: le budget de l'application (ou taux de dépense)

Événements et actions

La figure A.7 montre comment chaque événement est relié aux actions à la fois sur le système administré ou sur un autre gestionnaire tel que l'IM.

Workload Increased/Decreased : ces sont des événements endogènes (*endogenous event*) qui se produisent lorsque la charge de travail moyenne actuelle λ_{avg} (la moyenne charge de travail dans un intervalle de temps donné) atteint soit un seuil inférieur λ_L (*Workload Decreased*) ou un seuil supérieur λ_U (*Workload Increased*). Cet événement contient les nouvelles limites (supérieure et inférieure) de charge de travail ($\lambda_{next}^U, \lambda_{next}^L \in \Lambda$). Avant le déclenchement de la tâche d'analyse, l'AM demande les jetons des promotions disponibles afin d'y obtenir un accès exclusif. Afin d'éviter des états de blocage (*deadlock*), nous avons adopté une stratégie opportuniste pour exploiter les promotions. Un AM tente d'obtenir le jeton, si ceci n'est pas disponible, il abandonne et va à l'analyse comme s'il n'y avait pas de promotions disponibles. La tâche d'analyse détermine la meilleure configuration architecturale et le minimum quantité de ressources (en termes de VMs) pour les nouveaux seuils de charge de travail et libère jetons inutiles, i.e. des promotions qui n'intéressent pas aux applications. La tâche de planification génère un plan différent en fonction du résultat produit par l'analyse, à savoir s'il y a un besoin pour plus de VMs ou non. A la fin, la tâche de planification produit une action *Request VM (interloop)* avec un transfert de jeton (si c'est le cas), suivie d'une action *Schedule Handler* pour programmer un conteneur d'actions (objet futur). Le conteneur contient toutes les actions sur le système administré qui seront exécutées quand les VMs demandées seront disponibles (cf. figure A.7 (d)). Alternativement, la tâche de planification génère un ensemble d'actions de manipulation des composants relatives à l'application (système administré) (*Stop / Start Component, Bind Composant, emph Deploy / Remove Component*), suivie ou non d'une action *VM Created (interloop)* afin de notifié l'AM que les VMs sont prêtes.

Renting Fees Changed : c'est un événement exogène du type *Public Knowledge Changed*. Il se produit chaque fois que l'IM crée une nouvelle promotion. Ce genre d'événement déclenche la tâche d'analyse qui se comporte comme quand il y a des événements *Workload Increased/Decreased*. Dans ce cas, l'analyseur peut seulement demander de nouvelles VMs (celles des promotions).

VMs Created : il s'agit d'un événement *interloop* dont l'objectif est d'informer l'AM (depuis l'IM) que les VMs demandées sont prêtes à l'emploi. Cela déclenche directement la tâche d'exécution qui ne fait rien qu'exécuter les actions dans le conteneur programmé précédemment en passant comme argument l'objet futur (liste des VMs demandées) nécessaire à l'exécution des actions du conteneur. L'exécuteur déploie les composants sur les VMs créées, connecte ces composants à

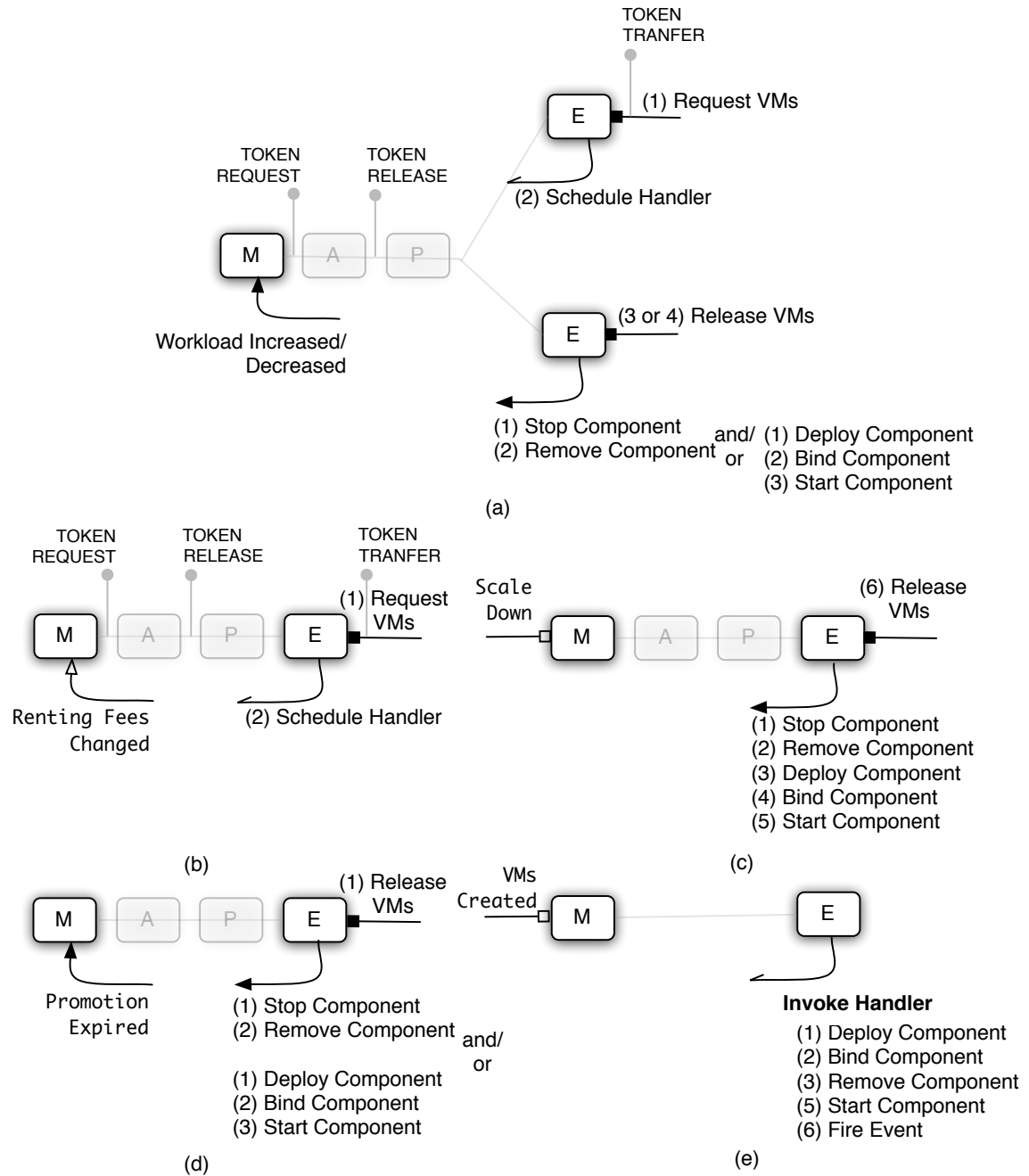


Figure A.7: Événements et actions du gestionnaire applicatif.

d'autres (existants ou nouveaux) et démarre les composants juste déployés. Il convient de rappeler que chaque promotion a une durée déterminée, après laquelle ces VMs redeviennent à prix régulier. Ainsi, dans le cas de nouvelles promotions, le conteneur contient aussi une action *FireEvent* afin de déclencher un événement *Promotion Expired* quand la promotion est expirée (i.e. après un délai donné).

Scale Down : c'est un événement du type *interloop* dont l'objectif est d'informer l'AM (depuis l'IM) qu'il doit répondre à certaines contraintes de ressources. Concrètement, l'événement contient la liste des VMs \check{V} qui devrait être immédiatement libérée par l'application en question. Cet événement déclenche une tâche d'analyse pour réaffecter les composants sur un plus petit nombre de VMs. À cette fin, il peut être nécessaire de modifier la configuration architecturale de l'application (par exemple en remplaçant les composants qui consomment davantage de ressources). En conséquence, un ensemble d'actions de manipulation de composants (ex: *Stop Component*, *Remove Component*, *Deploy Component*, *Bind Component* et *Start Component*) sont effectuées sur le système administré (l'application en question), suivie d'une action inter-boucle (*interloop*) *Release VM* pour libérer les VMs.

Promotion Expired : c'est un événement endogène déclenché par une action *FireEvent*, qui à son tour, est exécuté lors du traitement de l'événement *VMs Created*. Il indique qu'une promotion existante arrive à terme. Cet événement déclenche la tâche d'analyse afin de décider si l'AM maintient les VMs, même à prix régulier. A cet effet, l'analyse se comporte de manière similaire à quand il y a des événements *Workload Increased / Decreased*.

Analyse

La tâche d'analyse AM est composée de deux analyseurs différents, qui permettent de résoudre séparément deux problèmes en fonction de la nature de l'événement détecté. Ainsi comme pour les analyseurs de l'IM, les analyseurs de l'AM sont modélisés en CSP et basés sur la programmation par contrainte pour les résoudre.

Architecture and Capacity Planning Analyzer : cet analyseur est déclenché lors d'un événement *Workload Increased / Decreased*, *Renting Fees Changed* ou *Promotion Expired*. Ses objectifs sont de déterminer: (i) la configuration architecturale optimale (c'est à dire les composants et connexions) de manière à minimiser la dégradation architecturale, et (ii) le montant des ressources (nombre de VMs de chaque type alloué à chaque composant) nécessaire de façon à minimiser la dégradation de la performance et les coûts liés aux frais de location des ressources.

La première partie du modèle CP est définie par un ensemble de variables de décision: x , dont le domaine $D(x) \in [1, m]$, est une variable indiquant configuration architecturale de l'application; y_{ij} , où $D(y_{ij}) \in [1, \frac{budget}{m_j^{cost}}]$, $\forall i \in [1, n], \forall j \in [1, q]$, est une variable indiquant le nombre de VMs de classe m_j alloué au composant c_i ; u_i , où $D(u_i) \in \{0, 1\}$, $\forall i \in [1, n]$, est une variable indiquant si le composant est utilisé par la configuration architecturale x ; et p_i , où $D(p_i) \in \{0, 1\}$, $\forall i \in [1, \xi]$, est une variable indiquant si la promotion pr_i est utilisée.

La deuxième partie de ce modèle CP se compose d'un ensemble de contraintes auxquelles les variables sont soumises. Nous affirmons que le coût total de la demande en raison des prix de location de ressources ne peuvent pas dépasser le budget ($budget \geq \sum_{i=1}^n \sum_{j=1}^q y_{ij} * m_j^{cost}$).

L'équation A.6 définit une expression qui stocke la réduction obtenue à partir des promotions de rabais disponibles. L'équation A.7 indique le coût en tenant compte des réductions (l'équation A.6) et la variable de décision y_{ij} . Le coût est normalisé ($cost \in [0, 1]$) par rapport au budget de l'application. L'équation A.7 correspond à une contrainte qui indique que le coût total ne peut pas dépasser le budget.

$$disc = \sum_{i=1}^{\xi} \left(\sum_{j=1}^q (m_j^{cost} * pack_{ij}) - pr_i^{price} \right) * p_i + \sum_{i=1}^t \left(\sum_{\forall vm \in cpr_i^{pack}} vm^{cost} \right) - cpr_i^{price} \quad (A.6)$$

$$cost = \frac{\sum_{i=1}^n \sum_{j=1}^q y_{ij} * m_j^{cost} - disc}{budget} budget \geq \sum_{i=1}^n \sum_{j=1}^q y_{ij} * m_j^{cost} - disc \quad (A.7)$$

$$(A.8)$$

L'équation A.9 définit une contrainte indiquant que si une configuration est choisie, les composants correspondants devraient être marqués "utilisé" (variable u_i) et la variable auxiliaire deg^{arch} doit être égale à la dégradation architecturale correspondante. En ce qui concerne la contrainte exprimée par l'équation A.10, cela signifie que si un composant n'est pas utilisé par la configuration, il n'y a pas de VMs attribuées. L'équation A.11 exprime une contrainte indiquant l'impact des promotions choisies sur les ressources alloués (y_{ij}). Pour chaque nouvelle promotion p_l , si elle est prise par l'application en question, le nombre de VMs de chaque classe m_j doit être supérieure ou égale au nombre actuel de VMs de même classe affectés à l'application (e_j) plus le nombre de VMs ($pack_{lj}$) de la même classe offerte par la promotion pr_l .

$$x = l \implies (u_i = 1 \iff c_i \in k_i^c) \wedge deg^{arch} = k_i^{deg} \quad \text{où } l \in [1, m] \quad (A.9)$$

$$u_i = 0 \implies \sum_{j=1}^q y_{ij} = 0 : \forall i \in [1, n] \quad (A.10)$$

$$\forall j \in [1, q] (\sum_{i=1}^n y_{ij} \geq e_j + \sum_{l=1}^{\xi} p_l * pack_{lj}) \text{ où } pack_{lj} \in pr_l^{pack} : \forall l \in [1, \xi] \quad (A.11)$$

Il y a aussi une contrainte précisant que si le temps de réponse ($perf_i(\lambda_U^{next}, cpu_i, ram_i)$) est au-dessus du seuil acceptable, la dégradation (deg^{perf}) est égale à 1 (c'est à dire 100% de dégradation de performance). S'il est compris entre les seuils acceptable et idéal, la dégradation est égale à rt^{deg} . Dans le cas contraire, elle est égal à zéro, c'est à dire il n'y a pas de dégradation si le temps de réponse est inférieur ou égal au seuil idéal. Où $cpu_i = \sum_{j=1}^q y_{ij} * m_j^{cpu}$ et $ram_i = \sum_{j=1}^q y_{ij} * m_j^{ram}$ sont les quantités de CPU et RAM allouées au composant c_i .

La dernière partie du modèle CP est la fonction objectif. L'objectif de cette analyseur et par conséquent du modèle CP est de trouver la meilleure configuration architecturale tout en minimisant les coûts liés aux prix de location des ressources. L'équation A.12 définit une fonction objectif qui minimise les dégradations architecturale et de performance ainsi que le coût dû à la location des ressources.

$$minimize(\alpha_{perf} * deg^{perf} + \alpha_{arch} * deg^{arch} + \alpha_{cost} * cost) \quad (A.12)$$

Où $\alpha_{arch} + \alpha_{perf} + \alpha_{cost} = 1$ et $\alpha_{arch}, \alpha_{perf}, \alpha_{cost} \in [0, 1]$. $\alpha_{arch}, \alpha_{perf}$ correspondent aux poids des dégradations architecturale et de performance, et α_{cost} correspond au poids relatif à la location de ressources.

Scale Down Analyzer : cet analyseur est très similaire à l'analyseur *Architecture and Capacity Planning Analyzer*, puisque les objectifs sont les mêmes. La seule différence est que les ressources disponibles sont plus limitées dans le sens que certaines VMs doivent être supprimées. Soit $\bar{V} = V - \check{V}$, c'est la différence entre le vecteur de VMs en cours d'exécution et le vecteur de VMs à libérer. La contrainte exprimée par l'équation A.13 doit être vraie, où $e_i = 1$ si vm_i est une VM de classe m_i . Sinon, $e_i = 0$.

$$\forall j \in [1, q]. \sum_{i=1}^n y_{ij} \leq \sum_{i=1}^{|\check{V}|} e_i \quad (A.13)$$

Planification

La tâche de planification est déclenché après l'exécution de la tâche d'analyse. Elle prend en compte l'état actuel de l'application administrée (par exemple configuration architecturale et l'affectation composants / VMs) et l'état souhaité (par exemple, une nouvelle configuration architecturale et / ou une nouvelle affectation composants / VMs) résultant de la tâche d'analyse, pour créer les actions autonomes (cf. section A.4.3) de manière logique.

Selon la figure A.7, il y a quatre planificateurs possible, chacun est censé faire face à un événement différent. Cependant, comme ils ont les mêmes objectifs et les mêmes entrées et sorties, ils peuvent être mis en œuvre dans un seul planificateur (ci-appelé *Architecture and Capacity Planner*). Afin de faciliter la compréhension, nous divisons la description dans deux parties: Le planificateur se comporte différemment selon la sortie de l'analyseur : (i) lorsqu'il y a un besoin de nouvelles VMs, et (ii) lorsqu'il n'y a pas besoin de VM. La différence entre les deux cas est que dans le premier, il est nécessaire d'encapsuler un ensemble d'actions dans un conteneur *handler* qui est exécuté à l'arrivée d'un objet futur (les VMs demandées), alors que dans le second cas, l'ensemble du plan est exécutée sans attendre quoi que ce soit. Comme résultat, à part d'une action (*interloop*) pour demander / libérer de ressources (*Request / Release VMs*), le planificateur produit des actions sur l'application administrée (telles que *Stop / Start Component*, *Remove / Deploy Component* et *Bind Component*) pour manipuler les composants sur les VMs allouées à l'application en question.

Afin de rester le plus concis possible, nous ne détaillons pas l'algorithme qui met en œuvre ce planificateur.

A.5 Évaluation

Cette section présente les résultats obtenus à partir d'un scénario expérimental utilisé pour évaluer le protocole de coordination proposé. Le scénario consiste à déclencher un événement *Energy Shortage* au niveau de l'infrastructure et en observant ses effets sur la consommation d'énergie ainsi que sur la QoS des applications (SaaS hébergés). L'objectif est de montrer un cas réel d'utilisation du protocole de coordination (avec des événements et actions *interloop*) ainsi que d'évaluer comment se comportent les applications (lorsque elles sont équipées de l'élasticité architecturale) et l'infrastructure, capable de réagir à des situations extrêmes telles que la pénurie d'énergie. Il est à noter que nous avons évalué l'impact de la base de connaissance publique dans le cadre d'un scénario de promotion. De plus, nous avons également évalué l'évolutivité des modèles analytiques. Pour des raisons de place, nous omettons les analyses d'évaluation qui résultent de ces expériences.

A.5.1 Banc d'essai

Les expériences ont été effectuées sur un environnement de banc d'essai composé des applications basées sur le modèle architectural à base de composants (*Service Component Architecture - SCA*)¹ et une infrastructure matérielle réelle, dans lequel un ou plusieurs instances d'application sont déployées.

Application : L'application SCA est composée de trois composants $C = \{c_1, c_2, c_3\}$ et dispose de deux configurations architecturales $K = \{k_1, k_2\}$, où $k_1^c = \{c_1, c_2\}$, $k_1^{deg} = 0$, $k_2^c = \{c_1, c_3\}$, $k_2^{deg} = 0, 50$. Cela signifie que la configuration k_1 n'a pas de dégradation architecturale et c_1 est lié à c_2 , alors que dans la configuration k_2 il y a une dégradation de 50% et c_1 est lié à c_3 .

Chaque composant effectue des opérations intensives sur la CPU et a été calibré afin de profiler les exigences de performance de l'application. Les composants c_1 et c_2 consomment davantage des ressources que le composant c_3 . En conséquence, bien que la dégradation soit imposée, la configuration architecturale k_2 peut être une alternative afin de consommer moins de ressources, si

¹<http://oasis-opensca.org/sca>

nécessaire. Basé sur un profil de performance, nous définissons l'ensemble des seuils de charge de travail $\Lambda = \{0, 30, 55, 80, 170\}$. Comme dit précédemment, les seuils sont utilisés pour détecter événements de variation de charge de travail, ce qui peut conduire à des reconfigurations dans l'application.

Infrastructure : Pour l'infrastructure physique, nous nous sommes appuyés sur Grid'5000, qui est un réseau français pour banc d'essai expérimental. Il est composé de dix sites géographiquement distribué fournissant un ou plusieurs grappes, qui sont caractérisés par un ensemble de PMs équipées de dispositifs de mesure de puissance. Nous avons utilisé la grappe *sagittaire*,] qui est composé de 79 PMs *Sun Fire V20z*, une seule CPU / dual core (*AMD Opteron 250*) avec 2048 Mo de *DRAM* et 65GB d'espace de stockage. Nous avons également défini deux catégories de VMs $M = \{m_1, m_2\}$ (par exemple, petits et grands), où $m_1^{cpu} = 1, m_1^{ram} = 1, m_1^{cost} = 0.18$ and $m_2^{cpu} = 2, m_2^{ram} = 2, m_3^{cost} = 0.27$.

Client : Enfin, afin de simuler les clients de manière à produire une charge réelle, nous nous sommes appuyés sur Clif ², un framework de test de charge.

A.5.2 Protocole d'expérience

Pour ce scénario, nous avons déployé deux instances de l'application décrite dans la section précédente. Pour une instance, nous avons autorisé les deux configurations architecturales (k_1 et k_2) et pour l'autre, nous n'avons autorisé qu'une seule (k_1). Pour les deux instances, le budget a été fixé à 1, 5. La durée de l'expérience était d'une heure, pendant laquelle la charge de travail augmente de 0 à 140 requêtes par secondes. La charge de travail de l'application est décrite dans la figure A.9.

En ce qui concerne l'infrastructure, nous avons utilisé un sous-ensemble de 13 PMs de la grappe *sagittaire*. Nous avons effectué plusieurs essais en changeant le degré de pénurie d'énergie, c'est-à-dire le nombre de PMs à arrêter: 1, 3 et 5. L'événement *Energy Shortage* est programmé pour être déclenché à 45 minutes d'exécution pour toutes les trois exécutions. Le délai d'attente (*timeout*) avant l'arrêt des PMs par l'IM a été fixé à 10 minutes.

A.5.3 Résultats

La figure A.8 montre la consommation d'énergie totale de l'infrastructure, avant et après l'événement *Energy Shortage*, pour les trois exécutions (arrêt de 1, 3 et 5 PMs). Il convient de noter que l'intervalle entre la détection de l'événement (ligne verticale) et la diminution de la consommation d'énergie est dû au délai entre l'exécution de l'action *Notify Scale Down* (notification) sur les AM et l'arrêt effective des PMs. Ce délai correspond à 10 minutes. En ce qui concerne la consommation d'énergie, il est évident que plus les nombre PMs arrêtées est important moins grande est la consommation d'énergie.

La figure A.9 présente les résultats des trois exécutions en termes d'impact sur la QoS des applications. Plus précisément, le graphique montre la moyenne de temps de réponse pour les deux instances de l'application décrite précédemment (avec et sans élasticité de l'architecture (EA dans la figure). Sans surprise, l'arrêt de seulement 1 PM (cf. figure A.9 (a)) n'affecte pas les applications, car la PM choisie n'héberge aucune VM et par conséquent, aucun des composants des applications.

Lorsque trois PMs doivent être arrêtées (cf. figure A.9 (b)), il est déjà possible de remarquer les impacts sur la QoS des applications. En effet, l'application avec élasticité architecturale est capable de s'adapter afin de supporter la restriction de ressources et maintenir le niveau idéal de temps de réponse. Toutefois, pour l'application sans élasticité architecturale, son gestionnaire autonome ne fait que libérer les VMs requises par l'IM. En conséquence, le temps de réponse

²<http://clif.ow2.org>

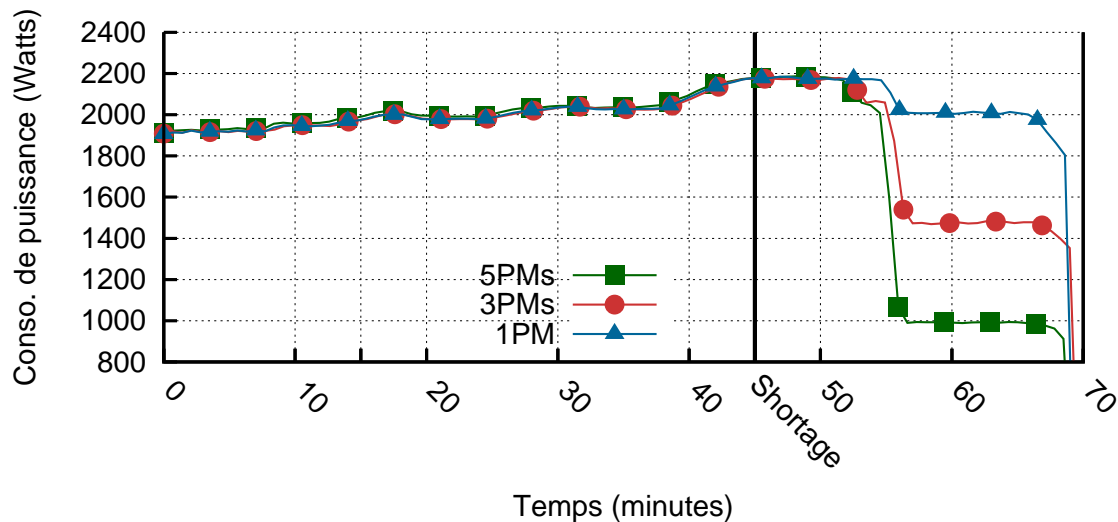


Figure A.8: Consommation de puissance lors d'une pénurie d'énergie.

augmente fortement à partir du moment où l'événement *Scale Down* est détecté, même s'il reste entre les niveaux idéal et acceptable.

Pour l'arrêt de cinq PMs (cf. figure A.9 (c)), l'impact sur la QoS des applications est plus visible. Lors de la détection d'un événement *Scale Down*, l'application sans élasticité architecturale dépasse de près de 100% le niveau acceptable de temps de réponse, tandis que l'application avec l'élasticité d'architecturale reste dans les limites de l'acceptable.

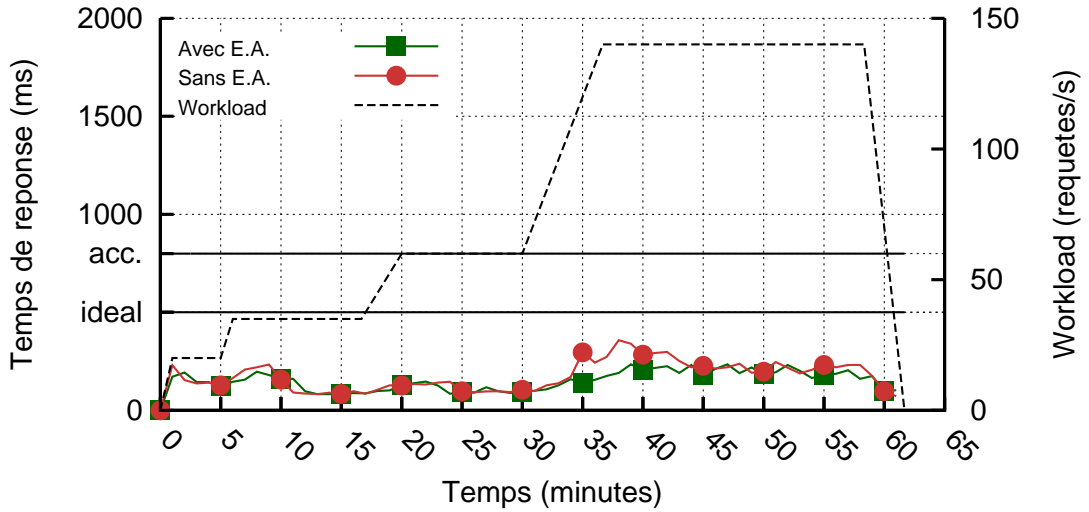
Pour résumer, ce scénario est important pour montrer comment élasticité architecturale ainsi que le protocole de coordination peuvent améliorer la synergie entre les gestionnaires AMs et IM. Comme c'est montré dans les figures A.9 et A.8, l'IM est capable de faire face à une situation extrême de façon transparente au niveau de l'infrastructure (contraintes d'énergie) en minimisant les effets secondaires négatifs sur les applications (par exemple, une dégradation de QoS).

A.6 Conclusion

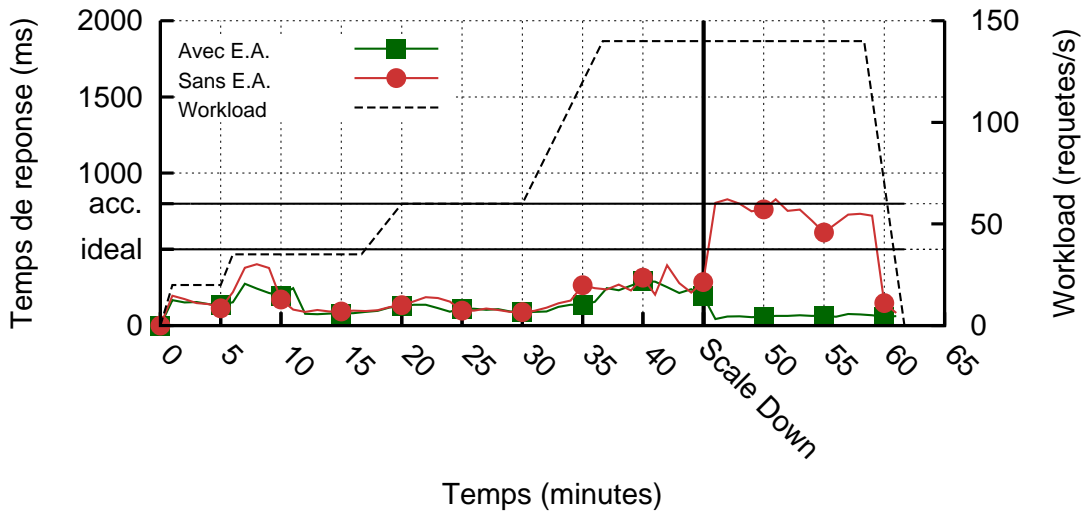
Au cours des dernières années, il a été constaté une augmentation drastique de la consommation d'énergie des infrastructures de technologie de l'information, ce qui peut être expliqué en partie par la dissémination de l'Internet et l'adoption à grande échelle des modèles de services récents basés sur Internet tels que l'informatique en nuage. D'une part, les modèles de l'informatique en nuage s'appuient sur le principe d'approvisionnement des ressources illimitées et disponible 24/24 7/7, ce qui peut conduire à des niveaux élevés de consommation, même si il n'est pas nécessaire. D'autre part, la flexibilité de l'approvisionnement à la demande caractéristique des services dans le nuage peut également contribuer à un meilleur dimensionnement de capacité et par conséquent à la réduction de la consommation d'énergie. En fait, la mutualisation des ressources physiques, ce qui est possible grâce à techniques telles que la virtualisation, ainsi que les concepts d'informatique autonome permettent non seulement aux infrastructures de se gérer elles-mêmes afin d'améliorer leur taux d'utilisation, mais ils permettent également aux applications de gérer elles-mêmes en ajustant dynamiquement leurs besoins en termes de ressources en fonction de leurs exigences.

A.6.1 Récapitulatif de la problématique

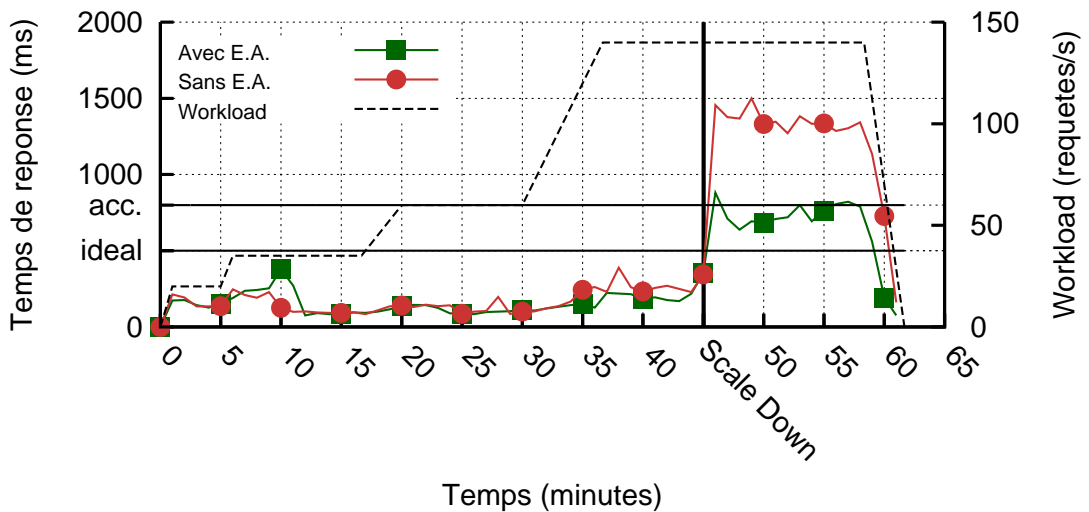
Malgré cela, les décisions autonomes prises isolément à un niveau peuvent avoir un impact négatif sur les systèmes fonctionnant à d'autres niveaux. Par exemple, due à des restrictions de consommation d'énergie, un gestionnaire autonome au niveau de l'infrastructure peut décider (ou



(a)



(b)



(c)

Figure A.9: Le temps de réponse moyen lors d'un événement *Scale Down* pour l'arrêt de: (a) 1, (b) 3, et (c) 5 PMs.

être obligé à) arrêter une partie de l'infrastructure physique. Il est évident que cette opération peut affecter les applications hébergées sur l'infrastructure en question. En effet, le manque de synergie entre les gestionnaires ainsi que le manque de flexibilité des systèmes administrés peuvent compromettre la QoS des ceux-ci. Par exemple, dans les cas extrêmes tels qu'une *pénurie d'énergie* au niveau de l'infrastructure, les applications peuvent être sollicitées ou même contraintes de libérer une partie des ressources qui leur sont attribuées. Les scénarios de ce genre peuvent considérablement influencer les systèmes exécutant s'il n'existe pas d'interface propre pour favoriser une synergie entre les gestionnaires, ni si le système n'est pas capable d'auto-adaptation, afin d'absorber ce niveau de restriction.

A.6.2 Sommaire des contributions

Cette thèse propose une approche de gestion multi-autonome dans le but d'améliorer la QoS des applications et l'efficacité énergétique des infrastructures dans le nuage. Cette section résume les contributions principales de cette approche.

Élasticité architecturale pour des applications dans le nuage. Au delà de l'élasticité de ressource typique des infrastructures en nuage, dans lequel les applications sont en mesure de s'agrandir ou de se contracter (en termes de ressource) selon la demande, nous avons également proposé un autre niveau de flexibilité : l'élasticité architecturale.

L'élasticité architecturale adresse la capacité des applications de changer dynamiquement leur structure architecturale afin de faire face à une contrainte donnée en termes de budget ou de ressources informatique. Cette flexibilité est obtenue grâce à deux facteurs principaux: (i) la modularité inhérente aux applications à base de composants, dans lesquelles on peut définir des applications en termes de pièces modulaires de logiciel qui peuvent être facilement déployées, remplacées et composées avec d'autres morceaux, et (ii) les techniques de réflexion qui permettent aux applications d'inspecter (naviguer) et intercéder (modifier) leur propre structure à l'exécution. De cette façon, les applications dans le nuage sont définies en termes de composants qui peuvent être dynamiquement, déployés, connectés ou remplacés dans le but de faire face aux changements à l'exécution.

Nous avons validé cette contribution via un scénario de contrainte budgétaire et un scénario de pénurie d'énergie. Les résultats ont montré l'importance de l'élasticité architecturale pour maintenir un certain niveau de QoS face à des situations extrêmes.

Modèle autonome pour plusieurs boucles de contrôle. Afin de faire fonctionner ensemble plusieurs gestionnaires autonomes, nous avons proposé un modèle de gestion autonome dans lequel plusieurs boucles de contrôle sont capables de communiquer de manière synchronisée. Le modèle est composé d'un protocole de coordination et d'un protocoles de synchronisation. Le protocole de coordination est un protocole orienté message basé sur des événements et des actions échangés dans le cadre du système administré, ainsi que entre plusieurs gestionnaires autonomes. Ce protocole empêche les gestionnaires de rester bloqués en attendant des réponses d'autres gestionnaires. Le protocole de synchronisation est nécessaire pour maintenir la cohérence des données dans des sections critiques des bases de connaissance publique. À cet effet, nous nous sommes appuyés sur un protocole à base de jetons afin de contrôler les accès simultanés aux sections critiques.

Des résultats obtenus via des simulations ont montré que ce modèle peut être réalisable (en termes de stabilité et de performance) pour un nombre raisonnable de gestionnaires autonomes concurrents. Cependant, le mécanisme de synchronisation a un prix: lorsque le nombre de gestionnaires autonomes augmente, le temps d'attente pour l'acquisition d'un jeton et par conséquent le temps de réaction (temps pendant lequel des événements sont traités) augmentent dans la même proportion. Ce problème a été résolu par l'adoption d'une approche opportuniste, c'est-à-dire qu'au lieu d'attendre jusqu'à que tous les jetons soient disponibles, les gestionnaires autonomes

n'essaie d'obtenir que ceux qui sont disponibles en moment donné. En conséquence, il n'y a pas de temps d'attente et ainsi le système peut passer à l'échelle.

Un gestionnaire autonome applicatif et un gestionnaire autonome pour l'infrastructure dans le nuage. En se basant sur le modèle autonome décrit dans le paragraphe précédent, nous avons proposé une architecture pour la gestion de plusieurs boucles de contrôle dans le contexte de l'informatique en nuage. Cette architecture est composée de plusieurs gestionnaires d'applications (AMs) et un gestionnaire de l'infrastructure (IM). Au niveau de l'infrastructure, l'IM est en charge de la gestion d'allocation de ressources d'un ensemble de machines physiques (PM). Il s'appuie sur des techniques de virtualisation pour traiter les demandes des ressources en provenance des AMs de manière dynamique, ce qui signifie que les machines virtuelles (VMs) sont demandées et libérées à la volée pendant que l'IM prend soin de les placer de sorte que le taux d'utilisation des infrastructures et par conséquent l'efficacité énergétique soient optimisés.

Au niveau applicatif, un AM par application est responsable de la gestion de l'ensemble des composants constituant l'application en question et la façon dont ceux-ci peuvent être connectés (élasticité architecturale) ainsi que la quantité de ressources nécessaires (l'élasticité des ressources) pour maintenir la QoS. À cette fin, les AMs s'appuient sur la flexibilité apportée par à la fois l'élasticité architecturale et l'élasticité des ressources de sorte à non seulement reconfigurer la structure interne de l'application (par exemple les composants, les connecteurs), mais aussi demander et libérer les machines virtuelles auprès de l'IM au besoin.

Nous avons mis en œuvre un prototype de recherche englobant à la fois les deux types de gestionnaires. Ce prototype a été utilisé pour évaluer qualitativement et quantitativement ce travail de thèse.

Synergie entre les applications et l'infrastructure. Le modèle autonome mentionné précédemment repose sur le partage d'information afin de faciliter la synergie entre gestionnaires autonomes. Dans le contexte des AMs et l'IM, nous avons défini une base de connaissance publique qui fait référence aux prix de location de ressource. Plus précisément, nous avons proposé deux types de prix de location, un statique, c'est à dire qu'il ne change pas avec le temps, et un dynamique, qui peut varier selon le contexte de l'infrastructure. Les prix de location dynamiques, dans cette thèse appelé promotions, sont créés dans le but de stimuler les AMs à occuper une partie de l'infrastructure qui n'est pas utilisée. La stratégie de promotion permet à l'IM à un certain niveau d'influencer les décisions prises par l'AMs, même si ces gestionnaires autonomes gèrent des systèmes appartenant à différents fournisseurs ou situés dans différentes couches de la pile de l'informatique de nuage. En conséquence, les actions réalisées par les AMs peuvent être plus appropriées pour l'infrastructure.

En plus de la synergie fondée sur la base de connaissance publique, les gestionnaires autonomes peuvent également bénéficier du protocole de coordination pour établir une synergie plus explicite par l'intermédiaire d'interfaces de communication (actions et événements). Par exemple, dans les périodes de *pénurie d'énergie*, l'IM demande à quelques AMs de libérer une certaine quantité de ressource, ce qui, grâce à l'élasticité architecturale, peut être réalisable par les AMs concernées. Ainsi, la synergie explicite avec l'élasticité architecturale rendent l'IM suffisamment flexible pour faire face à des situations extrêmes d'une manière transparente.

Les approches de synergie proposées ont été évaluées expérimentalement. Dans le cas de la synergie via une base de connaissance publique (promotion), il a été constaté un gain soit dans le taux d'utilisation ou dans la consommation totale d'énergie de l'infrastructure. Dans le cas de la synergie basée sur les interfaces de communication, il a été observé une amélioration de la souplesse des adaptations. En effet, en raison de ce niveau de synergie entre les AMs et l'IM, les applications impliquées dans un scénario extrême (par exemple, lors d'une pénurie d'énergie) ont été capable d'y faire face, tout en minimisant les impacts sur leur QoS.

Modèles basés sur la programmation par contraintes pour les problèmes d'optimisation.

La tâche de l'analyse de chaque gestionnaire autonome est mis en œuvre par un ou plusieurs problèmes d'optimisation. Nous avons proposé un modèle de programmation par contraintes (CP) pour chaque problème mentionné ci-dessus. CP est une technique de modélisation de problèmes de satisfaction de contraintes et d'optimisation, dans lequel un problème est décrit en termes de variables, domaines et des contraintes sur ces variables.

Compte tenu de la description du problème, un moteur général prend soin de le résoudre. La façon descriptive où les problèmes sont modélisés libère les développeurs du fardeau de l'élaboration d'un solveur spécifique pour chaque problème. Par ailleurs, ce couplage lâche entre le solveur et la description permet la réutilisation et la composition des contraintes existantes, et améliore également la maintenabilité et l'extensibilité des modèles existants.

Néanmoins, ce niveau de généralité et de description pour modéliser et résoudre les problèmes d'optimisation se fait au détriment des limitations liées au passage à l'échelle. Pour cette raison, nous avons réalisé des expériences de passage à l'échelle sur chacun des modèles afin d'exposer leurs limites. Les résultats ont montré que les solveurs sont en mesure de résoudre les problèmes respectifs dans un temps raisonnable pour les petites et moyennes applications et infrastructures.

A.6.3 Perspectives

Ce travail de recherche a réussi à régler les problèmes déjà soulignés au long de ce document. Une partie des contributions présentées précédemment ont fait émerger des perspectives de recherche intéressantes et prometteuses.

Mise en œuvre d'un *middleware* orienté message. Dans cette thèse, nous avons mis en place un *framework* autonome orienté message afin de permettre la communication entre plusieurs gestionnaires autonomes. Bien qu'il était conceptuellement défini pour s'appuyer sur un *middleware* orienté message (MOM), le protocole de communication a été appliqué d'une manière *ad-hoc* sur Hypertext Transport Protocol (HTTP). Nous croyons qu'un courtier MOM faciliterait la gestion de la communication dans le sens où il rendrait les détails de communication transparente pour les gestionnaires autonomes. La mise en œuvre d'une telle communication est un travail en cours.

Adoption d'autres *frameworks* à base de composants. Ce travail de thèse s'est appuyé sur *Service Component Architecture* (SCA) afin de mettre en œuvre des applications (dans le nuage) à base de composants et de permettre ainsi l'élasticité architecturale. Plus précisément, nous nous sommes appuyés sur FraSCAti [SMF⁺09], un environnement d'exécution SCA développé au-dessus du modèle de composant Fractal [BCL⁺04]. Malgré des nombreux avantages de FraSCAti tel que le support aux reconfigurations dynamiques, il y a encore quelques limitations en ce qui concerne l'aptitude aux grilles de calcul ou les définitions d'applications dans le nuage. Ainsi, il n'y a pas de support à Frascati pour la définition des composants dans différents domaines (machines physiques ou virtuelles), même si elles appartiennent à la même application. C'est décevant car il n'est pas possible de construire des applications à base de composants distribués tout en ayant un modèle unique de représentation. Pour ces raisons, il devient nécessaire d'étudier d'autres modèles de composants plus axés sur la grille comme *Grid Component Model* (GCM) [BCD⁺09]. GCM est un modèle de composant avec un accent sur l'informatique répartie. Il semble être une piste intéressante à étudier, car elle offre le support à la fois à la reconfiguration dynamique (également basé sur le modèle de composants Fractal) et à la distribution de composants dans le cadre de la grille de calcul.

Gestion de jetons sensible à l'équité et famine. Nous avons proposé un protocole à base de jetons pour gérer l'accès simultané aux sections critiques dans les bases de connaissance publiques, en particulier les promotions. Étant donné qu'il n'y a pas de contrôle dans la façon dont ces jetons

sont distribués, il peut arriver que le même demandeur obtient toujours un jeton, alors que les autres n'ont jamais la chance de l'obtenir. Cette question doit être traitée en travaux futurs.

Modèle de prix basé sur l'économie. Cette thèse propose une approche qui s'appuie sur une base de connaissance publique (plus précisément sur la forme de promotions) dans le but d'améliorer la synergie entre les application et l'infrastructure. Toutefois, les promotions sont créés d'une manière *ad-hoc*, ce qui signifie que nous ne précisons pas où, pour combien de temps et comment (avec combien de remise) les promotions devraient être créées pour optimiser l'utilisation des infrastructures et par conséquent l'efficacité énergétique. Les travaux sur le domaine d'*Yield Management* (gestion fine) [TVR05] ont été étudié afin d'appliquer de véritables modèles économiques basés sur des données statistiques pour pouvoir déterminer des prix dynamiquement. Ce type d'approche est particulièrement intéressante pour établir une synergie entre le fournisseur d'infrastructure et les fournisseurs d'applications, car il met l'accent sur la prévision de comportement des consommateurs. Alternativement, les modèles théoriques avancées telles que la théorie des jeux [NM44] sont également très prometteurs afin de trouver un équilibre dans un mode gagnant-gagnant (en termes de prix) entre les participants contradictoires (les applications et l'infrastructure fournisseurs).

Patrons de reconfiguration au niveau applicatif. Dans ce travail de thèse, nous nous sommes appuyés sur un mécanisme de répartition de charge pour effectuer le passage à l'échelle et sur une reconfiguration dynamique basée sur des composants pour effectuer élasticité architectural. Cependant, il existe d'autres modèles de reconfiguration qui pourraient être appliquées selon le contexte de l'application et des contraintes [NBF⁺12]. Par exemple, un modèle d'adaptation pourrait être de transformer une application dans une application multi-locataire afin de consommer moins de ressources ainsi que faire face à une contrainte de ressources/d'énergie donnée. Le défi consiste à déterminer quels modèles peuvent être appliquées et s'ils peuvent être appliqués ensemble.

Support au *Service Level Agreements (SLA)*. Dans les problèmes d'optimisation présentés dans ce travail de thèse, nous avons pris en compte des critères de dégradation (en termes de performance et architecture) qui peuvent être considérés comme des objectifs (ou *Service Level Objectives*) ou les guides afin d'éviter des violation de *Service Level Agreements (SLA)*. Malgré cela, ces critères ne pourraient pas être suffisant pour exprimer précisément les contrats comme il faut dans les applications du monde réel. Nous travaillons actuellement sur l'incorporation des SLA dans chaque couche de la pile logicielle de l'informatique en nuage à notre approche. À cette fin, des travaux comme [KL12, FPP12b] (au niveau de l'application et de la plateforme) et [BMDC⁺12] (au niveau des infrastructures) sont certainement de bonnes sources d'inspiration pour étendre notre approche pour intégrer la définition de SLA.

Fiabilité des reconfigurations et transactions. La fiabilité des opérations devient une question cruciale lorsqu'il s'agit de gestionnaires autonomes à la fois concurrents et interdépendants. En particulier, dans le contexte de cette thèse, il n'y a aucune garantie que les actions effectuées par l'ensemble des gestionnaires autonomes impliqués dans un processus de reconfiguration donné sont effectivement exécutée. En outre, en cas de échec, il n'y a pas de mécanismes de *rollback* (reprise) qui annulent les actions déjà effectuées, ce qui peut conduire les systèmes gérés impliqués à des états incompatibles. Cette question a déjà été traitée dans le cadre de la reconfiguration dynamique des architectures à base de composants [LLC10] et semble être un bon point de départ pour aborder cette question dans le cadre de ce travail de thèse.

Extension de l'interface entre les applications et l'infrastructure. Actuellement, dans ce travail, l'interface entre les AMs et l'IM est limitée aux actions pour demander/libérer de VMs, sans

donner plus de détails concernant les VM demandées tels que les contraintes au niveau des applications. Nous envisageons plusieurs situations dans lesquelles seules les opérations de libération/demande ne suffisent pas et des exigences spécifiques à chaque application doivent être traduites en des contraintes au niveau de l'infrastructure. Par exemple, il serait raisonnable de penser qu'une VM ne devrait jamais être hébergée dans la même PM avec une autre VM (pour des raisons de réplication), ou même le contraire (à fin de réduire la latence réseau). À cette fin, nous croyons que BtrPlace [HLM13] peut être un bon point de départ, car il propose une approche flexible pour la définition haut niveau des contraintes en termes de placement des VMs qui peut être fait à la demande au niveau applicatif et appliquée au niveau de l'infrastructure.

Stabilité et coût de reconfiguration. Dans ce travail de thèse, le coût associé à l'exécution d'une reconfiguration n'est pas pris en compte au moment de la prise de décision. Bien que certaines mesures, comme connecter ou supprimer de composants, peuvent avoir un faible coût en termes de temps d'exécution, d'autres comme le déploiement d'un composant ou la création d'une machine virtuelle peut avoir un coût non négligeable et devrait varier en fonction du composant ou de la taille des VMs. Par conséquent, il peut arriver qu'au moment où la reconfiguration prend effet elle ne soit plus utile ou qu'entre temps un autre événement (contradictoire) arrive, exigeant du gestionnaire autonome l'annulation de la reconfiguration précédente. Il y a des nombreuses pistes qui pourraient être étudiées pour atténuer cette instabilité. Tout d'abord, un modèle de coûts de reconfiguration doivent être pris en considération dans la formulation des problèmes d'optimisation. Ceci est important parce qu'en se basant sur un tel modèle, on peut analyser quand il est avantageux d'effectuer la reconfiguration étant donné un contexte d'exécution. En outre, une approche prédictive serait nécessaire pour établir le profil d'activité des systèmes administrés (par exemple charge de travail) afin de faciliter l'identification des périodes d'instabilité. La hiérarchisation des événements avec l'expiration d'événements périmés semblent également être nécessaire, car cela permettrait aux gestionnaires autonomes de choisir par exemple le plus récent entre deux ou plusieurs événements contradictoires (par exemple une détection d'une augmentation suivie d'une diminution de la charge de travail). Enfin, la préemption des gestionnaires autonomes lors du traitement d'un événement pourrait permettre la révocation d'une reconfiguration lors de l'arrivée d'un événement contradictoire, ce qui peut éviter des états futurs indésirables ou l'instabilité. A cet effet, une approche transactionnelle, comme indiqué précédemment, serait d'une grande utilité.

Bibliography

- [AB10] Jean Arnaud and Sara Bouchenak. Adaptive internet services through performance and availability control. In *Proceedings of the ACM Symposium on Applied Computing, SAC '10*, pages 444–451, New York, NY, USA, 2010. ACM. [7](#), [34](#), [53](#), [148](#), [172](#)
- [ACL⁺08] Danilo Ardagna, Cinzia Cappiello, Marco Lovera, Barbara Pernici, and Mara Tanelli. Active energy-aware management of business-process based applications. In *Proceedings of the 1st European Conference on Towards a Service-Based Internet (ServiceWave'08)*, pages 183–195, Berlin, Heidelberg, 2008. Springer-Verlag. [41](#)
- [AFG⁺09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. [11](#), [20](#), [167](#)
- [AFG⁺10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010. [20](#)
- [AP07] Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, 33(6):369–384, 2007. [35](#)
- [APTZ12] Danilo Ardagna, Barbara Panicucci, Marco Trubian, and Li Zhang. Energy-aware autonomic resource allocation in multitier virtualized environments. *IEEE Transactions on Services Computing*, 5(1):2–19, 2012. [11](#), [41](#), [53](#), [70](#), [148](#), [167](#), [172](#)
- [Bar05] Luiz André Barroso. The Price of Performance: An Economic Case for Chip Multiprocessing. *ACM Queue*, pages 48–53, September 2005. [12](#), [168](#)
- [BB10] Anton Beloglazov and Rajkumar Buyya. Energy efficient resource management in virtualized cloud data centers. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10*, pages 826–831, Washington, DC, USA, 2010. IEEE Computer Society. [12](#), [38](#), [40](#), [52](#), [53](#), [148](#), [155](#), [156](#), [167](#), [170](#), [172](#)
- [BCD⁺09] Françoise Baude, Denis Caromel, Cédric Dalmaso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. GCM: a grid extension to Fractal for autonomous distributed components. *Annales des Télécommunications*, 64:5–24, 2009. [28](#), [162](#), [192](#)
- [BCL⁺04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2003)*, Edinburgh, Scotland, May 2004. [28](#), [30](#), [105](#), [162](#), [192](#)

- [BDNDM10] Donato Barbagallo, Elisabetta Di Nitto, Daniel J. Dubois, and Raffaella Mirandola. A bio-inspired algorithm for energy optimization in a self-organizing data center. In *Proceedings of the First international conference on Self-organizing architectures*, SOAR'09, pages 127–151, Berlin, Heidelberg, 2010. Springer-Verlag. 40
- [BGC11] Rajkumar Buyya, Saurabh Kumar Garg, and Rodrigo N. Calheiros. Sla-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions. In *Proceedings of the 2011 International Conference on Cloud and Service Computing*, CSC '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society. 21
- [BH07] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, December 2007. 11, 12, 37, 51, 168, 170
- [BKB07] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing SLA violations. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128, May 2007. 40
- [BMDC⁺12] Damien Borgetto, Michael Maurer, Georges Da-Costa, Jean-Marc Pierson, and Ivona Brandic. Energy-efficient and sla-aware management of iaas clouds. In *Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet*, e-Energy '12, pages 25:1–25:10, New York, NY, USA, 2012. ACM. 163, 193
- [BPG⁺09] Fabienne Boyer, Noel Palma, Olivier Gruber, Sylvain Sicard, and Jean-Bernard Stefani. A self-repair architecture for cluster systems. In Rogério Lemos, Jean-Charles Fabre, Cristina Gacek, Fabio Gadducci, and Maurice Beek, editors, *Architecting Dependable Systems VI*, volume 5835 of *Lecture Notes in Computer Science*, pages 124–147. Springer Berlin Heidelberg, 2009. 28
- [BYV⁺09] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009. 11, 20, 167
- [CFF⁺11] Cinzia Cappiello, Alexandre Mello Ferreira, Maria Grazia Fugini, Pierluigi Plebani, and Monica Vitali. Business process co-design for energy-aware adaptation. In *Proceedings of the 7th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 463–470, aug. 2011. 48, 52, 53, 152, 171, 172
- [CGLR96] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the 5th International Conference on Knowledge Representation and Reasoning, (KR'96)*, pages 148–159. Morgan Kaufmann, 1996. 32
- [CHO] Choco solver documentation. <http://choco.svn.sourceforge.net/viewvc/choco/trunk/src/site/resources/tex/documentation/choco-doc.pdf>. Last access on 21/02/2013. 31
- [CMKS09] Trieu C. Chieu, Ajay Mohindra, Alexei A. Karve, and Alla Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *Proceedings of the 2009 IEEE International Conference on e-Business Engineering, ICEBE '09*, pages 281–286, Washington, DC, USA, 2009. IEEE Computer Society. 34

- [CPMK12] Stefania Costache, Nikos Parlavantzas, Christine Morin, and Samuel Kortas. An economic approach for application qos management in clouds. In *Proceedings of the 2011 international conference on Parallel Processing - Volume 2*, Euro-Par'11, pages 426–435, Berlin, Heidelberg, 2012. Springer-Verlag. [11](#), [47](#), [52](#), [53](#), [148](#), [167](#), [171](#), [172](#)
- [CRMDM12] Eddy Caron, Luis Roderio-Merino, Frédéric Desprez, and Adrian Muresan. Auto-Scaling, Load Balancing and Monitoring in Commercial and Open-Source Clouds. Research Report RR-7857, INRIA, February 2012. [61](#)
- [CSST98] Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Architectural reflection: Bridging the gap between a running system and its architectural specification. In *Proceedings of 6th Reengineering Forum (REF'98)*, pages 8–11. IEEE, 1998. [28](#)
- [DAB11] Erwan Daubert, Françoise André, and Olivier Barais. Adaptation multi-niveaux : l'infrastructure au service des applications. In *Proceedings of Conférence Française en Systèmes d'Exploitation (CFSE)*, St Malo, France, May 2011. [42](#), [52](#), [53](#), [170](#), [172](#)
- [DL05] Pierre-Charles David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, MPAC '05, pages 1–7, New York, NY, USA, 2005. ACM. [107](#)
- [DLLC08] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. FPath & FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annals of Telecommunications: Special Issue on Software Components – The Fractal Initiative*, 2008. [105](#)
- [DMR10] Gwenaël Delaval, Hervé Marchand, and Eric Rutten. Contracts for modular discrete controller synthesis. *SIGPLAN Notices*, 45(4):57–66, April 2010. [49](#)
- [DPM12] Djawida Dib, Nikos Parlavantzas, and Christine Morin. Towards multi-level adaptation for distributed operating systems and applications. In *Proceedings of the 12th international conference on Algorithms and Architectures for Parallel Processing - Volume Part II*, ICA3PP'12, pages 100–109, Berlin, Heidelberg, 2012. Springer-Verlag. [42](#)
- [DSV⁺11] Gargi Dasgupta, Amit Sharma, Akshat Verma, Anindya Neogi, and Ravi Kothari. Workload management for power efficiency in virtualized data centers. *Communications of the ACM*, 54(7):131–141, July 2011. [40](#)
- [EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003. [73](#), [74](#), [175](#)
- [Fac02] Alexander Factor. *Analyzing Application Service Providers*. Sun Microsystems Press Series. Sun Microsystems, 2002. [20](#)
- [FDD12] Sylvain Frey, Ada Diaconescu, and Isabelle Demeure. Architectural integration patterns for autonomic management systems. In *Proceedings of the 9th IEEE International Conference and Workshops on the Engineering of Autonomic and Autonomous Systems (EASe 2012)*. IEEE, April 2012. [50](#)
- [Fer11] Alexandre M. Ferreira. Energy-aware scheduling of service requests in multi-tier environment: Analysis and evaluation. In *Proceedings of the World Congress on Sustainable Technologies (WCST)*, pages 129–134, nov. 2011. [34](#), [53](#), [148](#), [172](#)

- [FFH⁺02] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming, CP '02*, pages 462–476, London, UK, UK, 2002. Springer-Verlag. 32
- [FK04] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Elsevier, 2004. 20
- [FPP12a] Alexandre Mello Ferreira, Barbara Pernici, and Pierluigi Plebani. Green performance indicators aggregation through composed weighting system. In *Proceedings of the Second international conference on ICT as Key Technology against Global Warming, ICT-GLOW'12*, pages 79–93, Berlin, Heidelberg, 2012. Springer-Verlag. 49
- [FPP12b] Andre Lage Freitas, Nikos Parlavantzas, and Jean-Louis Pazat. An integrated approach for specifying and enforcing slas for cloud services. In *Proceedings of the IEEE Fifth International Conference on Cloud Computing, CLOUD '12*, pages 376–383, Washington, DC, USA, 2012. IEEE Computer Society. 163, 193
- [FRM11] Eugen Feller, Louis Rilling, and Christine Morin. Energy-aware ant colony based workload placement in clouds. In *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing, GRID '11*, pages 26–33, Washington, DC, USA, 2011. IEEE Computer Society. 40
- [FRM12] Eugen Feller, Louis Rilling, and Christine Morin. Snooze: A scalable and autonomic virtual machine management framework for private clouds. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID '12*, pages 482–489, Washington, DC, USA, 2012. IEEE Computer Society. 12, 40, 53, 59, 148, 155, 156, 167, 172
- [GBF⁺12] íñigo Goiri, Josep Ll. Berral, J. Oriol Fitó, Ferran Juliá, Ramon Nou, Jordi Guitart, Ricard Gavaldí, and Jordi Torres. Energy-efficient and multifaceted resource management for profit-driven virtualized data centers. *Future Generation Computer Systems*, 28(5):718–731, May 2012. 40
- [GDA10] Guillaume Gauvrit, Erwan Daubert, and Françoise Andre. Safdis: A framework to bring self-adaptability to service-based distributed applications. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA '10*, pages 211–218, Washington, DC, USA, 2010. IEEE Computer Society. 42, 52, 53, 70, 148, 170, 172
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 30, 102
- [GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction, SIGPLAN '82*, pages 120–126, New York, NY, USA, 1982. ACM. 36
- [GMP⁺96] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, CP'96*, pages 179–193. Springer Berlin Heidelberg, 1996. 32

- [GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000. 27, 28
- [GTAB12] Ali Ghaddar, Dalila Tamzalit, Ali Assaf, and Abdalla Bitar. Variability as a Service: Outsourcing Variability Management in Multi-tenant SaaS Applications. In *Proceedings of the 24th International Conference on Advanced Information Systems Engineering*, volume 7328 of *CAiSE'12*, pages 175–189. Springer Berlin Heidelberg, 2012. 24
- [GWBB07] Lakshmi Ganesh, Hakim Weatherspoon, Mahesh Balakrishnan, and Ken Birman. Optimizing power consumption in large scale storage systems. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association. 38
- [Haa02] Peter J. Haas. *Stochastic Petri Nets: Modelling, Stability, Simulation*. Springer Series in Operations Research. Springer, 2002. 36
- [Has03] Service-oriented architecture explained, 2003. http://www.ondotnet.com/pub/a/dotnet/2003/08/18/soa_explained.html. Last access on 15 /12/2012. 29
- [HC01] George T. Heineman and William T. Council. *Component-based software engineering: putting the pieces together*. ACM Press Series. Addison-Wesley, 2001. 26, 27, 58
- [HLM⁺09] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 41–50, New York, NY, USA, 2009. ACM. 11, 12, 22, 38, 39, 40, 52, 53, 59, 81, 148, 156, 167, 170, 172
- [HLM13] Fabien Hermenier, Julia Lawall, and Gilles Muller. Btrplace: A flexible consolidation manager for highly available applications. *IEEE Transactions on Dependable and Secure Computing*, 99(PrePrints):1, 2013. 163, 194
- [HM08] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing-degrees, models, and applications. *ACM Computing Surveys*, 40(3):7:1–7:28, August 2008. 25
- [Hoa78] Charles A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. 88
- [Hor01] Paul Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical report, 2001. 24, 70
- [ISBA12] Vatche Ishakian, Raymond Sweha, Azer Bestavros, and Jonathan Appavoo. Cloud-pack* exploiting workload flexibility through rational pricing. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 374–393, New York, NY, USA, 2012. Springer-Verlag New York, Inc. 43, 52, 53, 148, 170, 172
- [JGJ97a] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software reuse: architecture process and organization for business success*. ACM Press books. ACM Press, 1997. 27

- [JGJ97b] Edward G. Coffman Jr., Michael Randolph Garey, and David S. Johnson. Approximation algorithms for np-hard problems. chapter Approximation algorithms for bin packing: a survey, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997. [38](#), [154](#)
- [KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003. [7](#), [11](#), [24](#), [25](#), [64](#), [70](#), [102](#), [167](#), [171](#)
- [KCD⁺07] Jeffrey O. Kephart, Hoi Chan, Rajarshi Das, David W. Levine, Gerald Tesaro, Freeman Rawson, and Charles Lefurgy. Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs. In *Proceedings of the Fourth International Conference on Autonomic Computing*, Washington, DC, USA, 2007. IEEE Computer Society. [11](#), [44](#), [52](#), [53](#), [151](#), [167](#), [171](#), [172](#)
- [KL12] Yousri Kouki and Thomas Ledoux. SLA-driven Capacity Planning for Cloud Applications. In *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science*, CloudCom 2012, Taipei, Taiwan, Province De Chine, December 2012. [21](#), [34](#), [163](#), [193](#)
- [KLS⁺10] Aman Kansal, Jie Liu, Abhishek Singh, Ripal Nathuji, and Tarek Abdelzaher. Semantic-less coordination of power management and application performance. *SIGOPS Operating Systems Review*, 44(1):66–70, March 2010. [7](#), [45](#), [46](#), [52](#), [53](#), [151](#), [171](#), [172](#)
- [Koo07] Jonathan G. Koomey. Estimating total power consumption by servers in the U.S. and the world. Technical report, Analytics Press, Oakland, CA, February 2007. [11](#), [167](#)
- [Koo11] Jonathan G. Koomey. Growth in Data center electricity use 2005 to 2010. Technical report, Analytics Press, Oakland, CA, August 2011. [11](#), [167](#)
- [KW00] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International System Administration and Network Engineering Conference*, SANE'2000, Maastricht, The Netherlands, 2000. [22](#)
- [LBMN09] Kien Le, Ricardo Bianchini, Margaret Martonosi, and Thu D. Nguyen. Cost-and energy-aware load distribution across data centers. In *Proceedings of the SOSP Workshop on Power Aware Computing and Systems*, HotPower '09, Big Sky, MT, USA, 2009. [40](#)
- [Lem13] Rogério et al. Lemos. Software engineering for self-adaptive systems: A second research roadmap. In Rogério Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *LNCS*. 2013. [50](#), [70](#), [171](#)
- [LL05] Yawei Li and Zhiling Lan. A survey of load balancing in grid computing. In Jun Zhang, Ji-Huan He, and Yuxi Fu, editors, *Computational and Information Science*, volume 3314 of *Lecture Notes in Computer Science*, pages 280–285. Springer Berlin Heidelberg, 2005. [60](#)
- [LLC10] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable dynamic reconfigurations in a reflective component model. In *Proceedings of the 13th international conference on Component-Based Software Engineering*, CBSE'10, pages 74–92, Berlin, Heidelberg, 2010. Springer-Verlag. [28](#), [163](#), [193](#)

- [LLH⁺09] Bo Li, Jianxin Li, Jinpeng Huai, Tianyu Wo, Qin Li, and Liang Zhong. Enacloud: An energy-saving application live placement approach for cloud computing environments. In *Proceedings of the IEEE International Conference on Cloud Computing, CLOUD '09*, pages 17–24, Washington, DC, USA, 2009. IEEE Computer Society. [12](#), [38](#), [52](#), [53](#), [59](#), [148](#), [156](#), [167](#), [170](#), [172](#)
- [LRA⁺05] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Systems*, 1(3):169–182, August 2005. [47](#)
- [LWYH09] Gregor von Laszewski, Lizhe Wang, Andrew J. Younge, and Xi He. Power-aware scheduling of virtual machines in dvfs-enabled clusters. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops, CLUSTER '09*, pages 1–10, New Orleans, LA, 2009. IEEE Computer Society. [37](#)
- [Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the Conference on Object-oriented programming systems, languages and applications, OOPSLA '87*, pages 147–155, New York, NY, USA, 1987. ACM. [28](#)
- [Mah04] Qusay H. Mahmoud. *Middleware for communications*. J. Wiley & Sons, 2004. [73](#)
- [MBC01] Rui S. Moreira, Gordon S. Blair, and Eurico Carrapatoso. A reflective component-based & architecture aware framework to manage architecture composition. In *Proceedings of the Third International Symposium on Distributed Objects and Applications, DOA '01*, pages 187–, Washington, DC, USA, 2001. IEEE Computer Society. [28](#)
- [MBC03] Rui Moreira, Gordon Blair, and Eurico Carrapatoso. Formaware: framework of reflective components for managing architecture adaptation. In *Proceedings of the 3rd international conference on Software engineering and middleware, SEM'02*, pages 115–129, Berlin, Heidelberg, 2003. Springer-Verlag. [28](#)
- [MFKP09] Alexandre Mello Ferreira, Kyriakos Kritikos, and Barbara Pernici. Energy-aware design of service-based applications. In *Proceedings of the 7th International Joint Conference on Service-Oriented Computing, ICSOC-ServiceWave '09*, pages 99–114, Berlin, Heidelberg, 2009. Springer-Verlag. [35](#), [53](#), [148](#), [172](#)
- [Mic08] Randal K. Michael. *Mastering Unix Shell Scripting: Bash, Bourne, and Korn Shell Scripting for Programmers, System Administrators, and UNIX Gurus*. Wiley, 2008. [114](#)
- [MKGdPR12] Soguy Mak-Karé Gueye, Noël de Palma, and Eric Rutten. Coordinating energy-aware administration loops using discrete control. In *Proceedings of the 8th International Conference on Autonomic and Autonomous Systems (ICAS 2012)*, March 2012. [49](#), [52](#), [53](#), [70](#), [151](#), [152](#), [171](#), [172](#)
- [Mon12] Ania Monaco. A view inside the cloud. *The Institute*, 2012. <http://theinstitute.ieee.org/technology-focus/technology-topic/a-view-inside-the-cloud>. Last access on 15/12/2012. [11](#), [20](#), [167](#)
- [MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000. [27](#)

- [NB10] Vivek Nallur and Rami Bahsoon. Design of a market-based mechanism for quality attribute tradeoff of services in the cloud. In *Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10*, pages 367–371, New York, NY, USA, 2010. ACM. [47](#)
- [NBF⁺12] Alexander Nowak, Tobias Binz, Christoph Fehling, Oliver Kopp, Frank Leymann, and Sebastian Wagner. Pattern-driven green adaptation of process-based applications and their runtime infrastructure. *Computing*, 94(6):463–487, June 2012. [162](#), [193](#)
- [NM44] John Von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944. [162](#), [193](#)
- [NRTV07] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA, 2007. [43](#)
- [Pap03] Mike P. Papazoglou. Service -oriented computing: Concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering, WISE '03*, pages 3–, Washington, DC, USA, 2003. IEEE Computer Society. [29](#)
- [PCL⁺11] Vinicius Petrucci, Enrique V. Carrera, Orlando Loques, Julius C. B. Leite, and Daniel Mosse. Optimized management of power and performance for virtualized heterogeneous server clusters. In *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID '11*, pages 23–32, Washington, DC, USA, 2011. IEEE Computer Society. [42](#), [52](#), [53](#), [148](#), [170](#), [172](#)
- [PDPBG09] Jérémy Philippe, Noël De Palma, Fabienne Boyer, and Olivier Gruber. Self-adapting service level in java enterprise edition. In *Proceedings of the 10th ACM/I-FIP/USENIX International Conference on Middleware, Middleware '09*, pages 8:1–8:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc. [7](#), [35](#), [36](#), [51](#), [53](#), [170](#), [172](#)
- [Pet11] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing, 2011. [11](#), [20](#), [167](#)
- [PPMM11] Diego Perez-Palacin, Raffaella Mirandola, and José Merseguer. Enhancing a qos-based self-adaptive framework with energy management capabilities. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS, QoSA-ISARCS '11*, pages 165–170, New York, NY, USA, 2011. ACM. [36](#), [53](#), [172](#)
- [QLS12] Flavien Quesnel, Adrien Lèbre, and Mario Südholt. Cooperative and Reactive Scheduling in Large-Scale Virtualized Platforms with DVMS. *Concurrency and Computation: Practice and Experience*, 2012. [40](#), [155](#)
- [RJQ⁺10] Ivan Roderó, Juan Jaramillo, Andres Quiroz, Manish P. Parashar, Francesc Guim, and Stephen W. Poole. Energy-efficient application-aware online provisioning for virtualized clouds and data centers. In *Proceedings of the International Conference on Green Computing, GREENCOMP '10*, pages 31–45, Washington, DC, USA, 2010. IEEE Computer Society. [40](#)
- [RL80] Martin Reiser and Stephen S. Lavenberg. Mean-value analysis of closed multi-chain queuing networks. *Journal of the ACM*, 27(2):313–322, April 1980. [34](#)

- [RRT⁺08] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No "power" struggles: coordinated multi-level power management for the data center. *SIGARCH Computer Architecture News*, 36(1):48–59, March 2008. [7](#), [44](#), [45](#), [52](#), [53](#), [151](#), [170](#), [172](#)
- [RVBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006. [14](#), [31](#), [32](#), [38](#), [81](#), [96](#), [110](#), [148](#), [179](#)
- [RvBW07] Francesca Rossi, Peter van Beek, and Toby Walsh. *Constraint Programming*, pages 189–211. Elsevier Science, San Diego, USA, 2007. [31](#), [32](#), [81](#), [96](#)
- [Sch92] Thomas Schiex. Possibilistic constraint satisfaction problems or "how to handle soft constraints?". In *Proceedings of the eighth conference on Uncertainty in Artificial Intelligence*, UAI'92, pages 268–275, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. [32](#)
- [SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Component Software Series. ACM Press, 2002. [27](#)
- [Sin04] Amit Singh. An introduction to virtualization, 2004. <http://www.kernelthread.com/publications/virtualization/>. Last access on 15/12/2012. [22](#)
- [SK11] S. Subashini and V. Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1):1 – 11, 2011. [23](#)
- [SKZ08] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. Energy aware consolidation for cloud computing. In *Proceedings of the Conference on Power-aware Computing and Systems*, HotPower'08, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association. [40](#)
- [SMF⁺09] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable sca applications with the frascati platform. In *Proceedings of the IEEE International Conference on Services Computing*, SCC '09, pages 268–275, Washington, DC, USA, 2009. IEEE Computer Society. [29](#), [30](#), [105](#), [132](#), [162](#), [192](#)
- [Smi84] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 23–35, New York, NY, USA, 1984. ACM. [28](#), [58](#)
- [SN05] James E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005. [11](#), [22](#), [59](#), [167](#)
- [SP97] Clemens Szyperski and Cuno Pfsister. Proceedings of the workshop on component-oriented programming. In Max Muhlhuaser, editor, *Special Issues in Object-Oriented Programming - ECOOP'96 Workshop Reader.*, Lecture Notes in Computer Science. dpunkt verlag, Heidelberg, 1997. [26](#)
- [Str12] Anja Strunk. Costs of virtual machine live migration: A survey. In *Proceedings of the IEEE Eighth World Congress on Services*, SERVICES '12, pages 323–329, Washington, DC, USA, 2012. IEEE Computer Society. [22](#), [38](#), [59](#), [65](#)
- [SZwL⁺11] Vivek Shrivastava, Petros Zerfos, Kang won Lee, Hani Jamjoom, Yew-Huey Liu, and Suman Banerjee. Application-aware virtual machine migration in data centers.

- In *Proceedings of the IEEE International Conference on Computer Communications*, INFOCOM'11, pages 66–70, april 2011. [40](#)
- [Tea10] CHOCO Team. Choco: an open source java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010. [31](#), [110](#), [153](#)
- [TvR05] Kalyan T. Talluri and Garrett J. van Ryzin. *The Theory and Practice of Revenue Management*. International Series in Operations Research & Management Science. Springer, 2005. [148](#), [162](#), [193](#)
- [VAN08a] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pMapper: power and migration cost aware application placement in virtualized systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 243–264, New York, NY, USA, 2008. Springer-Verlag New York, Inc. [22](#)
- [VAN08b] Akshat Verma, Puneet Ahuja, and Anindya Neogi. Power-aware dynamic placement of hpc applications. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 175–184, New York, NY, USA, 2008. ACM. [40](#)
- [VKKS11] Akshat Verma, Gautam Kumar, Ricardo Koller, and Aritra Sen. Cosmig: Modeling the impact of reconfiguration in a cloud. In *Proceedings of the IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 3–11, Washington, DC, USA, 2011. IEEE Computer Society. [22](#), [38](#), [59](#), [65](#)
- [Vog08] Werner Vogels. Beyond server consolidation. *Queue*, 6(1):20–26, January 2008. [46](#), [51](#), [170](#)
- [VTM10] Hien Nguyen Van, Frédéric Dang Tran, and Jean-Marc Menaud. Performance and power management for cloud infrastructures. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, pages 329–336, Washington, DC, USA, 2010. IEEE Computer Society. [11](#), [41](#), [53](#), [81](#), [148](#), [157](#), [167](#), [172](#)
- [VWMA11] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. On interacting control loops in self-adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 202–207, New York, NY, USA, 2011. ACM. [50](#)
- [WB11] Linlin Wu and Rajkumar Buyya. Service Level Agreement (SLA) in Utility Computing Systems. In Valeria Cardellini, Emiliano Casalicchio, Kalinka Regina Lucas Jaquie Castelo Branco, Júlio Cezar Estrella, and Francisco José Monaco, editors, *Performance and Dependability in Service Computing*. IGI Global, July 2011. [21](#), [34](#)
- [WKGB12] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. Sla-based admission control for a software-as-a-service provider in cloud computing environments. *Journal of Computer and System Sciences*, 78(5):1280–1299, September 2012. [151](#)
- [WN99] Laurance A. Wolsey and George L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 1999. [43](#)
- [WSKW06] Ian Warren, Jing Sun, Sanjev Krishnamohan, and Thiranjith Weerasinghe. An automated formal approach to managing dynamic reconfiguration. In *Proceedings*

of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE '06, pages 37–46, Washington, DC, USA, 2006. IEEE Computer Society.
28

- [WW11] Xiaorui Wang and Yefu Wang. Coordinating power control and performance management for virtualized server clusters. *IEEE Transactions on Parallel and Distributed Systems*, 22(2):245–259, February 2011. [7](#), [47](#), [48](#), [52](#), [53](#), [152](#), [171](#), [172](#)
- [YT12] Zeratul Izzah Mohd Yusoh and Maolin Tang. Composite SaaS Placement and Resource Optimization in Cloud Computing Using Evolutionary Algorithms. In *Proceedings of the IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 590–597, Washington, DC, USA, 2012. IEEE Computer Society. [7](#), [11](#), [37](#), [52](#), [53](#), [167](#), [170](#), [172](#)

Thèse de Doctorat

Frederico Guilherme ÁLVARES DE OLIVEIRA JÚNIOR

Gestion multi autonome pour l'optimisation de la consommation énergétique sur les infrastructures en nuage

Multi Autonomic Management for Optimizing Energy Consumption in Cloud Infrastructures

Résumé

Conséquence directe de la popularité croissante des services informatique en nuage, les centres de données se développent à une vitesse vertigineuse et doivent rapidement faire face à des problèmes de consommation d'énergie. Paradoxalement, l'informatique en nuage permet aux infrastructure et applications de s'ajuster dynamiquement afin de rendre l'infrastructure plus efficace en termes d'énergie et les applications plus conformes en termes de qualité de service (QoS). Toutefois, les décisions d'optimisation prises isolément à un certain niveau peuvent indirectement interférer avec (voire neutraliser) les décisions prises à un autre niveau, par exemple, une application demande plus de ressources pour garder sa QoS alors qu'une partie de l'infrastructure est en cours d'arrêt pour des raisons énergétiques. Par conséquent, il devient nécessaire non seulement d'établir une synergie entre les couches du nuage, mais aussi de rendre ces couches suffisamment souples et sensibles pour être en mesure de réagir aux changements d'exécution et ainsi profiter pleinement de cette synergie. Cette thèse propose une approche d'auto-adaptation qui prend en considération les composants applicatifs (élasticité architecturale) ainsi que d'infrastructure (élasticité des ressources) pour réduire l'empreinte énergétique. Chaque application et l'infrastructure sont équipées d'une boucle de contrôle autonome qui leur permet d'optimiser indépendamment leur fonctionnement. Afin de créer une synergie entre boucles de contrôle autour d'un objectif commun, nous proposons un modèle pour la coordination et la synchronisation de plusieurs boucles de contrôle. L'approche est validée expérimentalement à la fois qualitativement (amélioration de QoS et des gains d'énergie) et quantitativement (passage à l'échelle).

Mots clés

Autonomic Computing, Informatique en Nuage, Informatique Verte, Architecture Logicielle à Base de Composants

Abstract

As a direct consequence of the increasing popularity of Internet and Cloud Computing services, data centers are amazingly growing and hence have to urgently face energy consumption issues. Paradoxically, Cloud Computing allows infrastructure and applications to dynamically adjust the provision of both physical resources and software services in a pay-per-use manner so as to make the infrastructure more energy efficient and applications more Quality of Service (QoS) compliant. However, optimization decisions taken in isolation at a certain level may indirectly interfere in (or even neutralize) decisions taken at another level, e.g. an application requests more resources to keep its QoS while part of the infrastructure is being shutdown for energy reasons. Hence, it becomes necessary not only to establish a synergy between cloud layers but also to make these layers flexible and sensitive enough to be able to react to runtime changes and thereby fully benefit from that synergy. This thesis proposes a self-adaptation approach that considers both application internals (architectural elasticity) and infrastructure (resource elasticity) to reduce the energy footprint in cloud infrastructures. Each application and the infrastructure are equipped with their own autonomic manager, which allows them to autonomously optimize their execution. In order to get several autonomic managers working together, we propose an autonomic model for coordination and synchronization of multiple autonomic managers. The approach is experimentally validated through two studies: a qualitative (QoS improvements and energy gains) and a quantitative one (scalability).

Key Words

Autonomic Computing, Cloud Computing, GreenIT, Component-based Software Engineering.