Gestion autonomique de l'élasticité multi-couche des applications dans le Cloud : vers une utilisation efficiente des ressources et des services du Cloud

Simon Dupont

Thèse de Doctorat

Simon DUPONT

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École des Mines 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 26 Avril 2016
Thèse n° : 2016 EMNA 0239

Gestion autonomique de l'élasticité multi-couche des applications dans le Cloud

Vers une utilisation efficiente des ressources et des services du Cloud

JURY

Président :	**M. Noël DE PALMA**, Professeur, Université de Grenoble 1
Rapporteurs :	**M. Olivier BARAIS**, Professeur, Université de Rennes 1
	M. Romain ROUVOY, Maître de conférences, Université de Lille 1
Examinateurs :	**M. Noël DE PALMA**, Professeur, Université de Grenoble 1
	M. Thierry COUPAYE, Responsable de la Recherche IoT, Orange Labs
Invité :	**M. Steven MORVAN**, Responsable Innovation, Sigma Informatique
Directeur de thèse :	**M. Jean-Marc MENAUD**, Professeur des grandes écoles, École des Mines de Nantes
Co-encadrant de thèse :	**M. Thomas LEDOUX**, Maître de conférences, École des Mines de Nantes

À mesure que les améliorations technologiques augmentent l'efficacité avec laquelle une ressource est employée, la consommation totale de cette ressource tend à augmenter au lieu de diminuer.

Paradoxe de Jevons
William Stanley Jevons
The Coal Question - 1865

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Liste des portions de code

Remerciements

Je tiens à exprimer mes remerciements à M. Olivier Barais, Professeur à l'Université de Rennes 1 et M. Romain Rouvoy, Maître de conférences à l'Université de Lille 1, pour m'avoir fait l'honneur de rapporter les travaux de cette thèse.

J'associe à ces remerciements M. Noël De Palma, Professeur à l'Université de Grenoble 1, d'avoir présidé mon jury de thèse.

Mes remerciements s'adressent également à M. Thierry Coupaye, Responsable de la Recherche IoT chez Orange Labs, pour avoir été examinateur de thèse ainsi qu'à Mme Patricia Serrano, Maître de conférences à l'Université de Nantes, pour avoir su se montrer disponible dans le suivi annuel de mes travaux.

Je tiens à exprimer ma profonde gratitude et mes remerciements les plus sincères à M. Jean-Marc Menaud, Professeur à l'École des Mines de Nantes et M. Thomas Ledoux, Maître de conférences à l'École des Mines de Nantes ainsi qu'à M. Steven Morvan, Responsable Innovation à Sigma Informatique, pour avoir dirigé mes travaux de thèse mais avant tout pour m'avoir accordé leur confiance durant tout le long de cette thèse.

Je remercie l'ensemble de mes collègues du laboratoire LINA, pour m'avoir permis de réaliser mes travaux dans un contexte scientifique riche. Mes remerciements vont également à tous mes collègues de Sigma, et en particulier aux membres de la Direction Technique, pour leur professionnalisme et les moments agréables partagés durant cette période.

Enfin, je remercie de tout mon cœur mes chers parents, ma sœur et mon frère pour leur soutien inconditionnel. Je terminerai par une mention spéciale à ma chère et tendre pour tout l'amour qu'elle me porte au quotidien ainsi qu'à mon chat, Monsieur Havane, pour son aide toute relative et sa présence apaisante.

1

Introduction

Ce chapitre présente le contexte de nos travaux qui portent sur la gestion de l'élasticité pour l'informatique en nuage (Cloud computing). Nous présentons brièvement le modèle du Cloud computing, puis les limites actuelles de l'élasticité ainsi que les verrous sous-jacents. Enfin, nous présentons le sujet de notre thèse et nos contributions principales qui s'articulent autour de la définition d'une implémentation avancée de l'élasticité de l'informatique en nuage, étendue aux couches hautes du Cloud, en vue de pallier aux limitations du modèle actuel.

Contents

1.1 Contexte et problématique

Le Cloud computing permet aux utilisateurs d'accéder instantanément à des ressources informatiques au sens large, mises à disposition par des fournisseurs de Cloud sous forme de services à la demande. Les consommateurs de ces services bénéficient ainsi d'une flexibilité importante, avec un effort minimal de gestion et un investissement moindre, sans engagement sur le long terme. Du point de vue du fournisseur de service, la mutualisation induite par le modèle du Cloud lui permet d'optimiser ses coûts de déploiement et de maintenance et ainsi de bénéficier d'une économie d'échelle.

Le Cloud computing révolutionne complètement la façon de gérer les ressources informatiques. Grâce à l'élasticité, les ressources peuvent être provisionnées en quelques minutes pour satisfaire un niveau de qualité de service (*QoS - Quality of Service*) formalisé par un accord de niveau de service (*SLA - Service Level Agreement*) entre les différents acteurs (fournisseur et consommateurs). Le principal défi pour le fournisseur de services est de maintenir la satisfaction de ses consommateurs en assumant les SLAs signés (i.e. éviter le sous-dimensionnement de ressources), tout en minimisant le coût de ses services (i.e. éviter le sur-dimensionnement de ressources synonyme de gaspillage énergétique et financier).

L'élasticité de l'infrastructure (couche *IaaS - Infrastructure as a Service*) est une caractéristique cruciale pour les fournisseurs SaaS (*Software as a Service*) dont les applications évoluent dans des environnements hautement variables. Il s'agit de répondre aux demandes des utilisateurs en temps normal, mais aussi dans les périodes de forte activité ou encore dans le cas d'événements imprédictibles bien que récurrents : pannes logicielles, matérielles, coupures de courant, etc. Le système doit être capable de supporter une montée en charge puis un retour à la normale, sans interruption de service, et si possible, sans répercussions sur le niveau de service.

On distingue actuellement deux types d'élasticité portant sur la couche IaaS : l'élasticité verticale et l'élasticité horizontale. Ces deux types d'élasticité s'appuient sur la technique de virtualisation consistant à reproduire le comportement d'une machine physique (*PM - Physical Machine*) dans un environnement logiciel appelé machine virtuelle (*VM - Virtual Machine*). L'élasticité verticale consiste à augmenter/diminuer les ressources d'une VM telles que le CPU ou la mémoire. En pratique, cela revient souvent à changer la taille d'une instance en paramétrant son offre (*VM offering*) en passant par exemple d'une *Small instance* à une *Large instance* proposant davantage de ressources CPU et/ou RAM. Ce dimensionnement est limité par la quantité de ressource libre disponible sur le serveur physique hébergeant l'instance. L'élasticité horizontale, quant à elle, offre la possibilité d'ajuster le nombre de VMs (ajout/retrait) en fonction de la demande et permet théoriquement d'augmenter la capacité de l'application de manière infinie.

Malgré les avantages indéniables de l'élasticité, le modèle actuel du Cloud est sujet à certaines limitations (physiques, conceptuelles et techniques) empêchant de jouir pleinement des bienfaits de celle-ci. Premièrement, les ressources sont limitées, ce qui signifie que l'on ne peut pas passer à l'échelle de manière infinie. Deuxièmement, le temps d'initialisation des ressources IaaS n'est pas négligeable ce qui induit parfois un manque de réactivité face aux changements d'états du système (e.g. pic de charge brutal). Troisièmement, le modèle de facturation à l'heure utilisé aujourd'hui admet une granularité trop importante parfois synonyme de gaspillage financier pour le consommateur. Enfin, malgré les récents efforts pour améliorer l'efficacité énergétique du matériel et plus généralement des centres de données, l'évolution de l'impact énergétique des nouvelles technologies de l'information reste préoccupante.

Bien que de nombreux travaux scientifiques se soient intéressés à l'élasticité du Cloud en vue d'améliorer la capacité d'adaptation des systèmes, la pratique montre que les solutions industrielles souffrent aujourd'hui d'un manque d'outillage. En effet, les solutions reposent aujourd'hui sur des services de dimensionnement automatique basiques qui n'ont que très peu évolués depuis l'avènement du Cloud. On constate alors un manque d'adhérence entre les travaux scientifiques et les solutions industrielles du fait que les propositions scientifiques sont souvent trop complexes à mettre en œuvre dans un contexte industriel et généralement non outillées ce qui empêche leur industrialisation. En ce sens, la pauvreté des solutions d'élasticité industrielles actuelles laisse présager un besoin crucial d'équiper les administrateurs Cloud d'outils leur permettant de paramétrer le processus de gestion de l'élasticité de bout en bout et ainsi jouir pleinement des bienfaits de celle-ci.

1.2 Objectifs de la thèse

Afin de repousser les limites actuelles de l'élasticité, les travaux présentés dans cette thèse proposent d'étendre l'élasticité du Cloud à la couche SaaS. On parle alors d'*élasticité logicielle*. Nous définissons l'*élasticité logicielle* comme la *capacité d'un Logiciel à s'adapter, idéalement de manière autonome, pour répondre aux changements de la demande et/ou aux limitations de l'élasticité des ressources de l'infrastructure.*

Par analogie avec l'élasticité de l'infrastructure où les ressources IaaS (e.g. VMs) sont ajustées dynamiquement pour satisfaire les contrats de niveau de service (i.e. SLA), l'*élasticité logicielle* offre les moyens d'ajuster les ressources SaaS (e.g. composants logiciels) de manière transparente et rapide afin de respecter davantage les attentes et contraintes du point de vue de la QoS, des performances et des coûts en palliant les manques de l'élasticité de l'infrastructure en termes de précision et/ou de réactivité d'adaptation.

L'*élasticité logicielle* revient à fournir des applications offrant différents niveaux de service, plus ou moins consommateurs de ressources (i.e. CPU, RAM, énergie, etc.). Une application admet ainsi plusieurs *configurations* possibles qui résultent du choix des composants logiciels qui la constitue, de leur paramétrage ainsi que de leur nombre. Ces composants sont à granularité variable et peuvent concerner le logiciel lui-même ou le *middleware*. Au même titre que les ressources IaaS (i.e. VM) qui peuvent admettre différentes configurations (*VM offering*), nous considérons que les composants de l'application peuvent eux aussi être fournis selon différents modes (i.e. *component offering*). Ces différents modes, tout comme les différents types d'instances pour la couche *IaaS*, vont offrir différents niveaux de service, plus ou moins consommateurs de ressources (i.e. CPU, RAM, énergie, etc.). Par analogie avec l'élasticité des ressources IaaS, nous déclinons l'*élasticité logicielle* en deux types d'élasticité : verticale et horizontale. L'*élasticité logicielle verticale* consiste à changer le niveau de service de l'application en changeant l'offre de ses composants (i.e. paramétrage) alors que l'*élasticité logicielle horizontale* revient à connecter/déconnecter des composants (i.e. reconfiguration architecturale).

L'*élasticité logicielle* n'a pas pour vocation de remplacer l'élasticité de l'infrastructure, inhérente au Cloud, mais plutôt de pallier ses manques en termes de réactivité, de flexibilité et d'empreinte énergétique. La force de notre proposition réside dans la complémentarité entre l'élasticité de l'infrastructure et l'*élasticité du Logiciel*. De ce fait, il s'agit de considérer une *élasticité multi-couche*. Cette proposition d'étendre l'élasticité aux couches hautes du Cloud prend tout son sens dans le contexte de l'entreprise d'accueil *Sigma Informatique*. Le fait que *Sigma* soit à la fois gestionnaire des applications et de l'infrastructure offre un avantage certain pour nos travaux. En effet, cela rend possible le fait de considérer l'élasticité de manière transverse en considérant le contexte global du système (i.e. depuis les machines physiques jusqu'à l'utilisateur final des applications), et d'agir à tous les niveaux en vue d'optimiser l'utilisation des ressources Cloud au sens large.

Cette thèse vise à montrer que l'*élasticité multi-couche* permet d'adresser plus finement et efficacement les problématiques de reconfiguration liées au dimensionnement du système. Néanmoins, les nouvelles dimensions d'élasticité offertes par l'*élasticité logicielle* décuplent les possibilités d'adaptation. En effet, il devient possible de combiner différentes dimensions (IaaS et SaaS) dans un même plan de reconfiguration pour bénéficier de la synergie entre l'élasticité de l'infrastructure et de l'application. Cela rend la prise de décision de reconfiguration ardue. Quelle(s) dimension(s) choisir ? Dans quel contexte ? Selon quels critères ?

L'adaptation des différentes ressources du Cloud, qui évoluent dans un environnement dynamique, nécessite de prendre en considération de très nombreux paramètres. La complexité croissante des systèmes informatiques rend les tâches d'administration et de maintenance impraticable par l'homme. L'informatique autonome (*autonomic computing* [Hor01] [KC03]) vise à rendre les systèmes informatiques capables de s'auto-gérer. Cette notion d'autogestion du système comprend le fait de pouvoir détecter et diagnostiquer des symptômes signes d'instabilité, de s'adapter dynamiquement et automatiquement aux variations de l'environnement d'exécution, d'optimiser l'utilisation des ressources pour améliorer les performances et la QoS en minimisant les coûts, etc.

1.3 Contributions

L'objectif général de cette thèse tend à proposer un nouveau modèle d'*élasticité multi-couche* pour le Cloud. Afin d'atteindre cet objectif, cette thèse propose les contributions suivantes :

1. *un modèle conceptuel de l'élasticité multi-couche* : il s'agit dans un premier temps de définir et de modéliser l'*élasticité logicielle* ainsi que les différents concepts relatifs à cette nouvelle capacité d'adaptation. Nous exposons notre modèle de ressources Cloud à savoir une représentation du système sous forme de graphe de ressources de différents types. Enfin, nous proposons une modélisation de l'*élasticité multi-couche* qui revient à considérer les deux types d'élasticité (i.e. IaaS et SaaS) dans un même processus d'adaptation automatique.

2. *un modèle de surveillance outillé* : nous proposons un modèle conceptuel de surveillance reposant sur le traitement des événements complexes et permettant la définition d'une observation avancée des ressources Cloud au sens large. Une implémentation de ce modèle a été proposée au travers de l'outil *perCEPtion*. Ce *framework* permet à l'administrateur Cloud de manipuler le modèle de surveillance en définissant en amont les *symptômes*, signes d'incohérence, qui seront identifiés à l'exécution et pourront faire l'objet d'une adaptation du système.

3. *un modèle d'adaptation outillé* : notre modèle d'adaptation repose sur la définition et l'exécution de *tactiques d'élasticité*. Une *tactique d'élasticité* correspond à un plan de reconfiguration générique spécifiant l'adaptation à réaliser face à un type de *symptôme*. Un tel plan peut faire intervenir une ou plusieurs actions d'adaptation portant sur différents types de ressources. Afin de décrire efficacement de telles *tactiques d'élasticité*, potentiellement multi-couche, nous proposons un langage dédié nommé *ElaScript*. *ElaScript* est un langage dédié à l'*élasticité multi-couche* permettant à l'administrateur Cloud de définir simplement et de manière concise des plans de reconfiguration complexes sous forme de *scripts*. Ce langage offre la possibilité de composer et d'orchestrer les multiples actions d'élasticité via des opérateurs de séquentialité, parallélisme et synchronisation.

4. *un modèle de décision d'adaptation* : ce modèle est chargé de la prise de décision concernant les changements à apporter au système. Notre modèle de décision prend la forme d'un processus constitué de trois étapes, représentées sous forme de *filtres* successifs et dont le point d'entrée est un *symptôme* remonté à l'exécution par notre outil *perCEPtion*. Il s'agit, à partir d'un *symptôme* signe d'instabilité, de décider de l'adaptation à mettre en œuvre sur le graphe de ressources, en vue de retrouver un état stable. La décision va revenir à choisir la meilleure adaptation possible face à un *symptôme*, parmi les différentes *tactiques d'élasticité* définies au préalable par l'administrateur humain avec le langage *ElaScript*. Notre modèle de décision est paramétrable et offre la possibilité à l'administrateur d'émettre ses préférences d'adaptation sous forme de *stratégie d'élasticité*. Le processus de décision va ainsi prendre en considération le contexte d'exécution courant du système et les contraintes associées ainsi que les préférences spécifiées par l'administrateur.

5. *un framework autonomique de gestion de l'élasticité multi-couche* : nous proposons le *framework ElaStuff*, une solution complète pour la gestion de l'*élasticité multi-couche*. Il s'agit d'un gestionnaire autonomique de type *MAPE-K* permettant d'assurer le processus de gestion de l'*élasticité multi-couche* de bout en bout, depuis son paramétrage par l'administrateur Cloud jusqu'à son exécution. Ce *framework* autonomique s'appuie sur les différents modèles évoqués ci-dessus ainsi que sur notre outil de surveillance *perCEPtion* et notre langage dédié *ElaScript*.

1.4 Plan de thèse

Ce mémoire de thèse est découpé en trois grandes parties s'organisant comme suit :

1. la première partie est constituée de l'état de l'art de la thèse. Le chapitre 2 apporte le contexte dans lequel s'inscrit le sujet de la thèse au travers d'un certain nombre de définitions, de concepts de base et de technologies liés à nos travaux. Enfin, nous exposons dans le chapitre 3 notre état de l'art scientifique qui vise à recenser et analyser plusieurs travaux liés à l'élasticité dans le Cloud computing en vue d'identifier les manques et limitations de l'existant et ainsi dégager les enjeux de cette thèse.

2. la seconde partie présente les contributions scientifiques de cette thèse. Elle inclut le chapitre 4 qui dévoile les résultats préliminaires de notre travail autour de l'*élasticité logicielle* (i.e. SaaS) au travers d'un projet, nommé *TUBA*, amorcé dès la première année à l'entreprise et faisant office de *preuve de concept* concernant l'*élasticité logicielle* et plus généralement l'élasticité *multi-couche*. Le chapitre 5 présente la définition de notre modèle d'élasticité *multi-couche* qui étend le modèle d'élasticité des ressources IaaS actuel en considérant les ressources propres aux couches hautes du Cloud (i.e. SaaS/PaaS). De plus, ce chapitre expose notre modèle de gestion autonome de l'élasticité *multi-couche* ainsi que notre *framework* autonomique prenant la forme d'une boucle de contrôle de type *MAPE-K*. Puis, nous présentons dans le chapitre 6 deux outils développés dans cette thèse dans le but d'outiller et d'industrialiser le processus de gestion de l'élasticité *multi-couche*. Le premier outil nommé *perCEPtion* est un *framework* basé sur le traitement des événements complexes (*CEP - Complex Event Processing*), s'appuyant sur un modèle de surveillance, et permettant à l'administrateur Cloud de mettre en place une observation avancée du système en définissant en amont les *symptômes*, signes d'incohérence, pouvant nécessiter une adaptation à l'exécution. Le second outil est un langage dédié (*DSL - Domain Specific Language*) à la gestion de l'élasticité *multi-couche* permettant à l'administrateur Cloud de définir simplement et de manière concise des plans de reconfiguration complexes sous forme de *scripts*. Ce langage offre la possibilité de composer et d'orchestrer les multiples actions d'élasticité via des opérateurs de séquentialité, parallélisme et synchronisation. Enfin, le chapitre 7 expose le processus de décision de notre *framework* autonomique chargé de la prise de décision concernant les changements à apporter au système.

3. la troisième partie inclut le chapitre 8 qui contient les validations expérimentales réalisées dans cette thèse. Nous y présentons les résultats des expériences effectuées en vue de valider notre proposition d'étendre l'élasticité aux couches hautes du Cloud, et plus particulièrement à la couche SaaS, selon plusieurs points de vue (i.e. QoS, énergie, réactivité et précision du passage à l'échelle, etc.). Enfin, le chapitre 9 conclut ce mémoire de thèse en récapitulant les contributions apportées et en dressant certaines perspectives de travaux futurs.

1.5 Diffusion scientifique

Yousri Kouki, Frederico Alvares, Simon Dupont and Thomas Ledoux. *Language support for Cloud Elasticity Management.* In IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid [ccg14]), Chicago, Illinois, USA, May 26-29, 2014. [KdODL14]

Simon Dupont, Jonathan Lejeune, Frederico Alvares and Thomas Ledoux. *Experimental Analysis on Autonomic Strategies for Cloud Elasticity.* In IEEE International Conference on Cloud and Autonomic Computing (CAC [cac15]), Cambridge, Massachusetts, USA, September 21-25, 2015 [DLAL15].

I

État de l’art

2

Contexte et concepts

Nous présentons brièvement dans ce chapitre le contexte dans lequel s'inscrit le sujet de la thèse. Nous détaillons un certain nombre de définitions, de concepts de base et de technologies liés à nos travaux. Nous commençons par présenter le modèle du Cloud computing ainsi que les grandes définitions associées. Puis, nous développons la notion d'élasticité du Cloud computing au cœur de cette thèse. Ensuite, nous élargissons le concept d'élasticité à la problématique de gestion de capacité et de dimensionnement automatique. Enfin, dans le contexte hautement dynamique et complexe que représente l'informatique en nuage, nous nous arrêtons sur les intentions du domaine de l'informatique autonomique visant à rendre les systèmes capables de s'auto-administrer.

Contents

2.1 Cloud computing

2.1.1 Définitions

La notion de Cloud fait référence à l'allégorie classique d'Internet, souvent représenté sous forme de nuage.

La métaphore du nuage désigne en réalité un réseau de ressources informatiques (i.e. matérielles et logicielles) virtualisées et mutualisées, accessibles à distance, à la demande et en libre-service via le réseau par le biais des technologies Internet [AFG+10] [ZCB10] [BYV+09] [VRMCL08].

L'informatique en nuage, ou Cloud computing, constitue une nouvelle façon de délivrer les ressources informatiques. On parle aussi d'informatique virtuelle ou encore d'informatique dématérialisée. Il s'agit d'accéder à des ressources informatiques, reposant sur une architecture distante gérée par une tierce partie : le fournisseur de service (i.e. *service provider*). Celui-ci assure la continuité et la maintenance du service dont il a la charge. Les utilisateurs accèdent à la partie applicative via leur connexion Internet sans se soucier du reste. Les utilisateurs peuvent ainsi louer (i.e. paiement à l'utilisation, *pay-per-use*) les technologies de l'information et bénéficier d'une flexibilité importante avec un effort minimal de gestion.

Il existe de très nombreuses définitions du Cloud computing, insistant parfois sur les aspects conceptuels, technologiques ou encore économiques. Nous donnons ci-dessous une première définition scientifique ainsi qu'une seconde définition du *National Institute of Standards and Technology* (*NIST*), largement acceptée par la communauté scientifique et industrielle.

> « *A Cloud is a type of parallel and distributed system consisting of a collection of interconnected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers [BYV08].* » (Buyya et al. - 2008)

> « *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models [MG11].* » (*NIST* - 2011)

Selon le *NIST*, le modèle du Cloud computing [LTM+11] repose sur trois modèles de service, quatre modèles de déploiement et cinq grandes caractéristiques (cf. Figure 2.1).

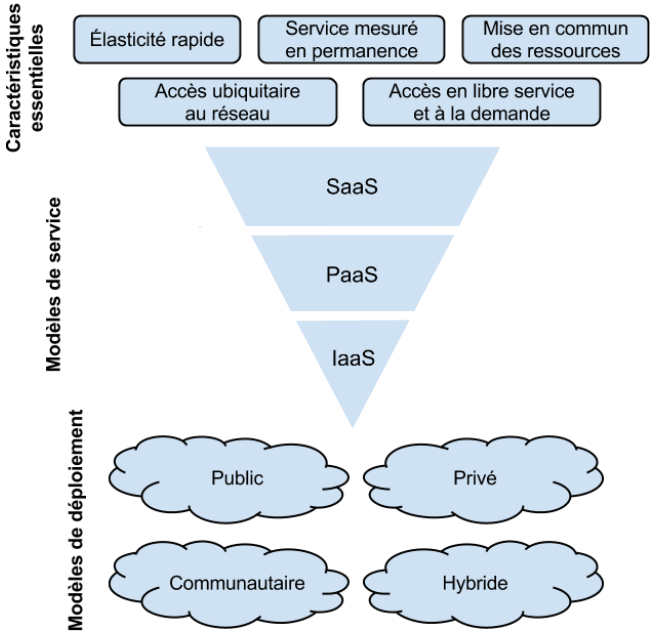


FIGURE 2.1 – Modèle du Cloud computing selon le *NIST*.

Modèles de service

On dénombre actuellement trois grands modèles de service Cloud souvent présentés comme le "modèle en couche" (cf. Figure 2.2) :

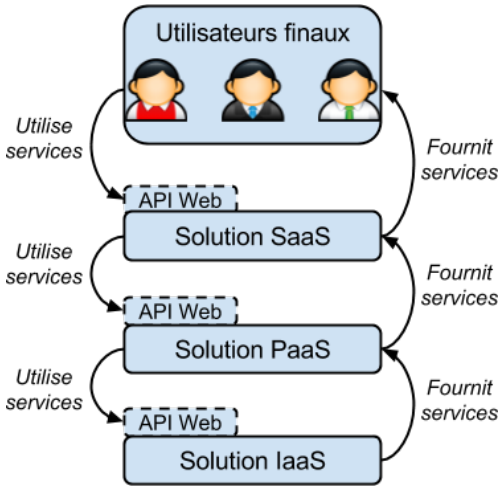


FIGURE 2.2 – Modèles de service : fournisseurs et utilisateurs.

— *SaaS (Software as a Service)* : Les "logiciels en tant que service" sont des applications accessibles via internet où le matériel, l'hébergement, le *framework* d'application et le logiciel sont dématérialisés. Les applications de type SaaS sont diverses et variées. On peut citer les applications grand public de type *B2C* (*Business to Consumer*) telles que la messagerie *Gmail*, le réseau social *Facebook* ou encore le service de cartographie en ligne *Google Maps*. Les applications peuvent aussi être à destination de professionnels. On parle alors d'applications *B2B* (*Business to Business*) parmi lesquelles on retrouve les *CRM* (*Customer Relationship Management*) comme *StayinFront B2B* ou encore les solutions logicielles proposées par *Salesforce.com*.

— *IaaS (Infrastructure as a Service)* : La ressource fournie est du matériel informatique virtualisé ou, en d'autres mots, une infrastructure informatique. Physiquement, les ressources matérielles (*hardware*) proviennent d'une multitude de serveurs et de réseaux généralement distribués à travers de nombreux centres de données (*datacenters*), que le fournisseur de services Cloud a la responsabilité d'entretenir. Une solution IaaS permet aux utilisateurs d'accéder de façon flexible à des systèmes virtuels complets en démarrant/arrêtant à la demande des ressources virtuelles dans des *datacenters*, sans se soucier de la gestion du matériel sous-jacent. Parmi les acteurs principaux de IaaS, on retrouve *Amazon EC2* et *Windows Azure*.

— *PaaS (Platform as a Service)* : Les "plateformes en tant que service" offrent l'accès à un environnement de développement administré, hébergé et maintenu par le fournisseurs PaaS, facilitant le déploiement et l'exécution des applications SaaS en ajoutant une couche de services à la couche IaaS. Situé entre la couche SaaS et la couche IaaS, le PaaS abstrait la couche IaaS à ses utilisateurs et encourage ainsi les équipes de développement à se concentrer sur l'architecture et la réalisation des applications. *Cloud Foundry*, *Stackato* ou encore *OpenShift* sont de bons exemples de solutions de type PaaS.

La Figure 2.3 (inspirée de *tecbizz.net*) met en lumière la distinction entre les préoccupations adressées par les fournisseurs des différents modèles de service en détaillant les tâches d'administration et de gestion qui les incombent. L'acronyme XaaS (*Anything as a Service* [Sch09]) a été créé en raison du nombre croissant d'applications basées sur le concept d'externalisation de fonctionnalités sous forme de services (e.g. DaaS : *Data as a Service*, NaaS : *Network as a Service*, etc.).

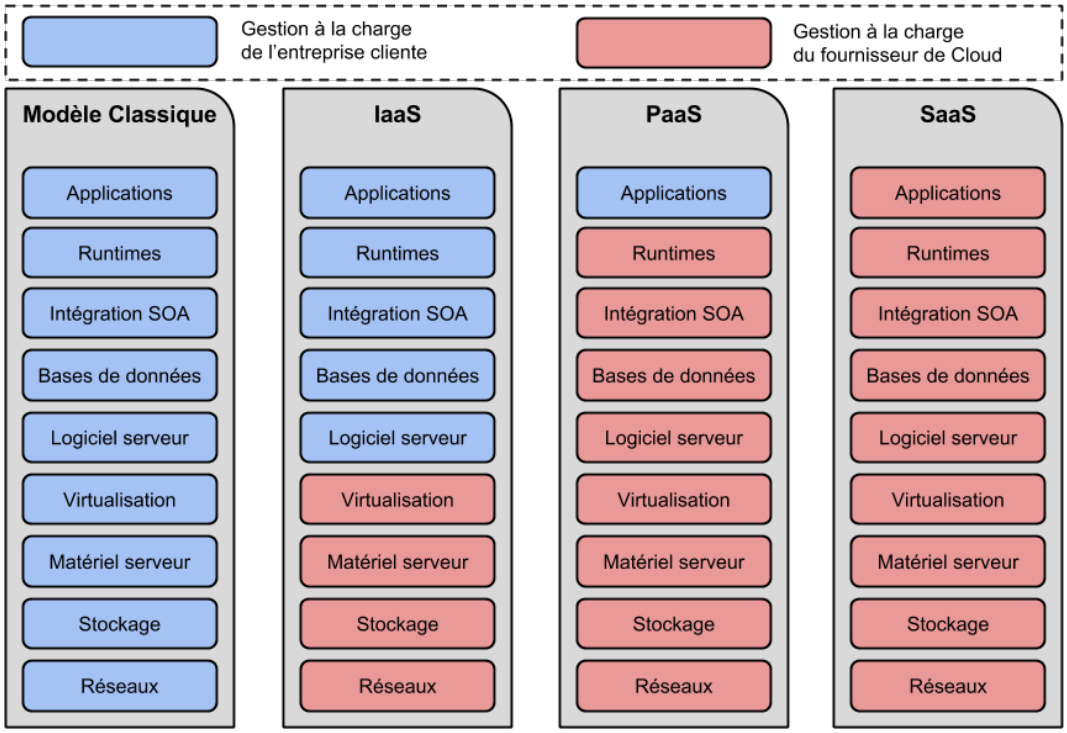


FIGURE 2.3 – Répartition des tâches d'administration des modèles de service.

Modèles de déploiement

Le *NIST* distingue quatre modèles de déploiement pour le Cloud correspondant à des usages différents des ressources du Nuage.

— *Cloud privé* : il s'agit d'un environnement dont les ressources servent exclusivement l'entreprise utilisatrice. L'infrastructure peut être localisée/gérée sur place ou par un tiers mais l'entreprise est la seule à l'utiliser.
— *Cloud communautaire* : l'infrastructure est partagée entre plusieurs organisations et peut être gérée par le groupe ou par un tiers.
— *Cloud public* : les ressources sont fournies par un prestataire, propriétaire de celles-ci, et mutualisées pour un usage partagé.
— *Cloud hybride* : il s'agit de la combinaison de plusieurs clouds indépendants publics ou privés. Ceux-ci doivent respecter des standards et des technologies communes pour assurer la portabilité des applications entre les clouds.

Caractéristiques essentielles

Le *NIST* attribue au concept de Cloud computing cinq caractéristiques indispensables et essentielles :

— *Accès en libre service à la demande (on-demand self-service)* : l'usager peut réserver/libérer les ressources en fonction de ses besoins sans interaction avec le fournisseur. Les ressources sont fournies d'une manière entièrement automatisée au client et celui-ci peut gérer à distance ses ressources au moyen d'une interface.

— *Accès ubiquitaire au réseau (broad network access)* : l'ensemble des ressources est accessible de partout via le réseau (Internet ou privé) en s'appuyant sur des mécanismes standards facilitant l'accès au service pour différents types de clients.
— *Mise en commun des ressources (resource pooling)* : le fournisseur mutualise les ressources (ou services) qu'il attribue dynamiquement aux différents clients en fonction de la demande. Ce partage des ressources est la caractéristique qui différencie le Cloud computing de l'infogérance.
— *Service mesuré en permanence (measured service)* : l'utilisateur est facturé en fonction de l'usage des ressources. L'utilisation des ressources/services est contrôlée/mesurée et communiquée au client/fournisseur de service de façon transparente.
— *Élasticité rapide (rapid elasticity)* : l'usager bénéficie d'un accès rapide et souple aux ressources. Celles-ci peuvent être réservées/libérées rapidement, et parfois de manière automatisée, pour répondre à des besoins qui évoluent vite.

2.1.2 Virtualisation

Définitions et Concepts

La virtualisation est une technique permettant de reproduire le comportement d'une machine physique (*Physical Machine* - PM) dans un environnement logiciel appelé machine virtuelle (*Virtual Machine* - VM). Ainsi, l'utilisateur a l'illusion de manipuler une PM alors qu'il s'agit en réalité d'un environnement logiciel. La virtualisation offre la possibilité de faire fonctionner de multiples systèmes d'exploitation et/ou applications sur un seul serveur physique. Chaque VM exécute son propre système d'exploitation (*Operating System* - OS) ce qui permet à l'utilisateur d'héberger des logiciels. On parle alors d'OS *invité* pour le distinguer de celui de la PM qualifié d'OS *hôte*.

Le concept de virtualisation, à la base du Cloud computing, apporte de très nombreux avantages tels que l'optimisation de l'utilisation des ressources existantes par la mutualisation des ressources physiques qui résulte en une économie sur le matériel. Une VM est donc un conteneur de logiciels isolé, capable d'exécuter ses propres systèmes d'exploitation et applications. Les VMs sont créées et gérées par des logiciels appelés hyperviseurs (cf. Figure 2.4). À l'instar d'une PM, une VM contient un processeur, une mémoire RAM ainsi qu'un disque dur et une carte d'interface réseau qui lui sont propres bien que virtuels [SN05].

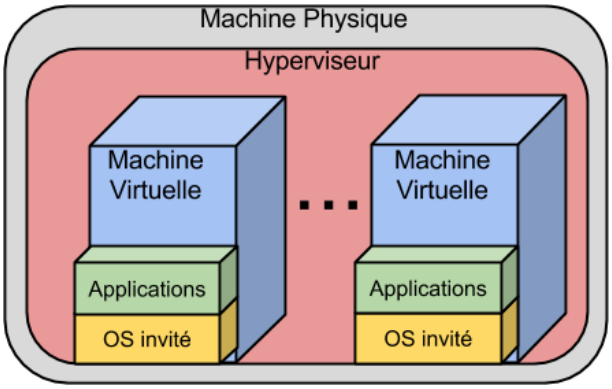


FIGURE 2.4 – Infrastructure virtualisée.

Conteneurs légers

Depuis une dizaine d'années, la virtualisation s'est imposée de manière naturelle dans le domaine de l'informatique. Bien que l'utilité de la virtualisation ne soit plus à prouver, une partie de la communauté a dernièrement remis en cause le niveau de granularité des VMs souvent considérées comme lourdes du fait qu'elles embarquent le système d'exploitation ainsi que l'ensemble des librairies en plus des applications.

Une autre approche a vu le jour dernièrement poussant un peu plus loin la notion de mutualisation : les conteneurs légers (par opposition à la VM, parfois qualifiée de conteneur lourd). Cette approche vise à mutualiser le système d'exploitation et certaines librairies allégeant ainsi considérablement la granularité des briques déployées ce qui permet de gagner en rapidité de déploiement (cf. Figure 2.5). Un conteneur léger peut être déployé aussi bien sur des PMs que sur des VMs. Bien que de plus en plus populaire, avec notamment l'avènement de *Docker* [DOC15] depuis 3 ans qui perturbe quelque peu le marché de la virtualisation des serveurs, cette approche n'est pas nouvelle et remonte à plus de 10 ans. *Sun* a introduit dans *Solaris* cette technologie de conteneurs : les Solaris Zones. Puis, cette technologie a été introduite dans le monde Linux avec l'outil *LXC* [LXC15] (*Linux Containers*), qui est d'ailleurs à l'origine de *Docker*.

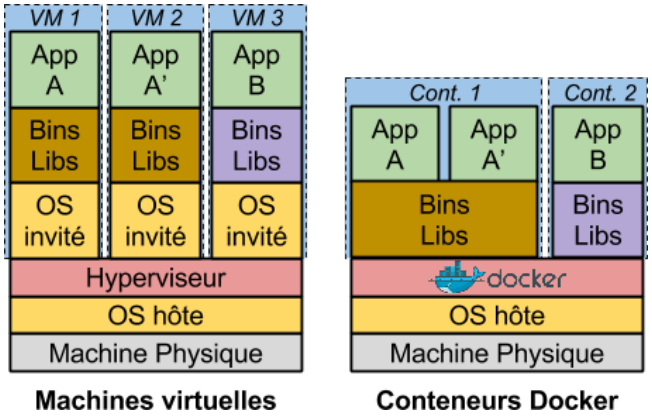


FIGURE 2.5 – Architectures : machines virtuelles et conteneurs *Docker*.

L'approche reste parfois critiquée en termes de sécurité et d'isolation par rapport aux VMs. En effet, il n'est pas possible à l'heure actuelle d'isoler les ressources (i.e. CPU, RAM, etc.) utilisées par des conteneurs basés sur la même machine, si bien qu'un conteneur peut très bien "vampiriser" les ressources de ses voisins et ainsi déstabiliser l'ensemble du système. La dernière version de *Docker* introduit la possibilité d'appliquer une limite maximum de consommation de ressource pour l'ensemble des conteneurs d'une même machine. Bien que cela évite les dérives, il n'est pas encore possible d'allouer de manière fine et dynamique des ressources aux conteneurs en fonction de leur besoin via l'API de haut niveau. Il reste néanmoins possible de s'appuyer sur la fonctionnalité *cgroups* (i.e. *control groups*) du noyau Linux dans le but de limiter, compter ou isoler l'utilisation des ressources (processeur, mémoire, utilisation disque, etc.) ou encore de prioriser celles-ci pour certains groupes.

2.1.3 Modèle économique

Économie d'échelle

Le Cloud computing, à travers la mutualisation des ressources informatiques, entraîne des économies d'échelle pour les fournisseurs de service. En effet, la mutualisation de la demande couplée aux avantages de la virtualisation en termes de flexibilité favorisent une exploitation maximale des PMs. D'autres avantages doivent être considérés comme la possibilité pour les gestionnaires de centres de données de bénéficier de remises quantitatives sur le matériel utilisé ou encore la possibilité d'automatiser un certain nombre de tâches d'administration, ce qui permet de réduire la masse salariale et les coûts associés.

Les utilisateurs des services du Cloud bénéficient eux aussi d'avantages financiers. En effet, la souplesse du modèle du Cloud permet un accès facile et quasi immédiat à des ressources informatiques sans investissement important (e.g. achat de matériel, de licences, etc.) et sans engagement à long terme. De plus, les coûts d'exploitation associés à l'utilisation de ces services sont moindres (e.g. frais de maintenance, de mise à jour, etc.). Ainsi, les utilisateurs estiment prendre moins de risques en confiant la réalisation de certaines fonctionnalités à un fournisseur plus expérimenté et trouvent cette option plus rentable du fait qu'ils bénéficient eux aussi des économies d'échelles réalisées par les fournisseurs.

Paiement à l'usage

Le modèle de facturation, selon la formule *"pay-as-you-go"*, est l'une des spécificité de l'informatique en nuage. On parle aussi de paiement ou facturation à l'usage (*pay-per-use*). La facturation à l'usage assure le client de ne payer que ce qu'il consomme réellement au moment où il consomme. Les offres Cloud tendent à se multiplier et se complexifier. Au niveau IaaS, la tarification est un sujet assez complexe à cause de la difficulté de comparaison des offres. En général, la tarification correspond à une heure d'instance consommée pour chaque instance. Dans le cas d'*Amazon* [EC215b], la tarification commence à partir du moment où une instance est lancée jusqu'à son arrêt. Néanmoins, comme la plupart des IaaS publics du marché, la granularité du modèle de facturation est à l'heure. Concrètement, cela induit que chaque heure d'instance partielle consommée sera facturée comme une heure pleine. Dans ce cas, l'utilisateur peut être amené à payer davantage que ce qu'il consomme réellement s'il ne libère pas les ressources à temps. Dans le passé, le modèle de facturation de *Windows Azure* était encore moins souple dans le sens où la facturation se faisait par heure naturelle fixe, c'est-à-dire qu'une instance démarrée à 11h55 et arrêtée à 12h05 était facturée comme 2 heures d'utilisation au client. Désormais, *Microsoft Azure* est passé à une facturation à la minute [AZR15].

Différentes offres

À l'instar de la téléphonie mobile proposant différents forfaits avec de multiples options ou extensions, les fournisseurs IaaS proposent toute une gamme d'instances en fonction des besoins des utilisateurs comme c'est le cas d'*Amazon EC2*. Il s'agit concrètement de multiples combinaisons en matière de capacités CPU, RAM, stockage et réseau. Chaque type d'instance inclut une ou plusieurs tailles possibles en fonction des exigences liées à la charge de travail ciblée. Actuellement, cela peut varier d'une *micro instance* avec 1 vCPU et 1 Gio de mémoire à une *10xlarge instance* avec 40 vCPU et 160 Gio de mémoire [EC215b]. On parle parfois d'offre de machine virtuelle (*VM offering*). La combinatoire des possibilités résulte en un nombre de configuration impressionnant. En novembre 2012, [GGRTRC13] dénombrait pas moins de 1758 configurations possibles pour une instance *Amazon* (sans considérer les options relatives à la couche *middleware*). La Figure 2.6 donne un aperçu de la tarification horaire d'*Amazon EC2* en fonction des offres de VM.

2.1.4 Accord de niveau de service - SLA

Qualité de service

La qualité de service (*Quality of Service* - QoS) est la capacité d'un service à répondre aux besoins des utilisateurs en respectant leurs exigences. Dans le contexte du Cloud, les critères considérés diffèrent selon les modèles de service [ADC10] et les services proposés. Néanmoins, on retrouve deux métriques récurrentes [PSG06] : la disponibilité (*availability*) et le temps de réponse (*response time* ou *latency*). Les principales composantes de la QoS sont fournies par des métriques caractérisées par un type, une unité et une fonction de calcul.

	vCPU	ECU	Mémoire (GiB)	Stockage des instances (Go)	Utilisation de Linux/UNIX
Usage général – Génération actuelle					
t2.micro	1	Variable	1	EBS uniquement	$0.014 par heure
t2.small	1	Variable	2	EBS uniquement	$0.028 par heure
t2.medium	2	Variable	4	EBS uniquement	$0.056 par heure
t2.large	2	Variable	8	EBS uniquement	$0.112 par heure
m4.large	2	6.5	8	EBS uniquement	$0.139 par heure
m4.xlarge	4	13	16	EBS uniquement	$0.278 par heure
m4.2xlarge	8	26	32	EBS uniquement	$0.556 par heure
m4.4xlarge	16	53.5	64	EBS uniquement	$1.112 par heure
m4.10xlarge	40	124.5	160	EBS uniquement	$2.78 par heure
m3.medium	1	3	3.75	1 x 4 SSD	$0.073 par heure
m3.large	2	6.5	7.5	1 x 32 SSD	$0.146 par heure
m3.xlarge	4	13	15	2 x 40 SSD	$0.293 par heure
m3.2xlarge	8	26	30	2 x 80 SSD	$0.585 par heure

FIGURE 2.6 – Illustration de la tarification horaire d'*Amazon* par type d'instance.

Service Level Agreement

Le concept de *Service Level Agreement* (SLA) a été introduit dans les années 80 par les opérateurs téléphoniques afin de gérer la QoS dans leurs contrats avec les entreprises [WB12]. Un SLA, que l'on peut traduire en français par "accord de niveau de service", est un contrat dans lequel la QoS est formalisée et contractualisée entre le prestataire de service et le(s) utilisateur(s). De nombreuses définitions du terme SLA ont été proposées par la communauté scientifique et industrielle. Celles-ci diffèrent généralement du fait qu'elles adressent des domaines distincts. Nous donnons ci-dessous deux définitions de SLA adressant respectivement le domaine du réseau et celui des services Web.

> « *An SLA is a contract between a network service provider and a customer that specifies, usually in measurable terms, what services the network service provider will supply and what penalties will assess if the service provider can not meet the established goals [WB12].* » (Réseau)

> « *SLA is an agreement used to guarantee web service delivery. It defines the understanding and expectations from service provider and service consumer [JMS02].* » (Service Web)

Un SLA contient la description et le rôle des parties impliquées, la ou les définition(s) de service(s), les clauses, une période de validité du contrat et les pénalités en cas violation [JMS02]. Les clauses correspondent à des SLOs (*Service Level Objective*) que l'on peut traduire en français par "objectifs de niveau de services". Un SLO exprime un engagement à maintenir un état particulier du service pendant une période donnée. Un exemple de SLO entre un fournisseur d'application SaaS et ses utilisateurs peut être que 90% des requêtes traitées par le service dans une journée doit l'être en moins de 2000ms. Au même titre que la QoS, les garanties dans un SLA diffèrent selon les modèles de service. Le modèle en couche du Cloud induit qu'un acteur peut avoir différents rôles. Par exemple, un SaaS est à la fois fournisseur de services auprès des utilisateurs finaux et consommateur de ressources virtuelles/matérielles auprès des PaaS/IaaS. On voit alors apparaître des dépendances entre les SLAs de différents niveaux [KL12].

2.1.5 Enjeux énergétiques

Un concept de base du Cloud computing est la concentration de la production d'énergie informatique dans de grands centres de données. La mutualisation des ressources induit une baisse de coût, et les technologies de virtualisation permettent de rationaliser l'usage des serveurs. Dans son argumentaire, le Cloud se veut donc plus éco-responsable que les architectures de type *Cluster* où chacun a son propre parc de ressources avec des serveurs plus ou moins utilisés. Néanmoins, l'évolution de la consommation énergétique des TICs (Technologies de l'Information et de la Communication) et plus précisément des centres de données reste préoccupante [Koo11].

Efficience énergétique

Les TICs se doivent aujourd'hui plus que jamais d'adresser les préoccupations énergétiques. On parle d'efficacité ou d'efficience énergétique (*energy efficiency*) pour définir le fait de réduire le bilan énergétique des systèmes. Dans le contexte du Cloud computing, il est difficile de définir plus précisément cette notion du fait que l'on adresse un large panel de ressources allant du matériel et systèmes de refroidissement associés, aux composants logiciels en passant par le réseau, etc. [MOC+14]. Nous donnons ci-dessous une définition générale du terme *efficiency* ainsi qu'une définition s'intéressant à l'efficience énergétique.

> « *Efficiency : achieving maximum productivity with minimum wasted effort or expense.* » (Oxford Dictionary)

> « *Energy efficiency refers to a reduction of energy used for a given service or level of activity [GdCdM+10].* » (World Energy Council - 2010)

En informatique, le concept d'efficience couvre plusieurs sous-objectifs. On distingue généralement trois objectifs principaux couverts par les termes *resource efficiency*, *energy efficiency* et *cost efficiency* qui visent à maximiser la productivité tout en minimisant respectivement la quantité de ressource utilisée, la consommation énergétique et les coûts. Les trois termes ci-dessus sont corrélés du fait qu'une optimisation de l'utilisation des ressources va entrainer une réduction de l'empreinte énergétique elle-même synonyme de diminution des coûts.

Différents travaux

Des efforts importants ont été faits ces dernières années pour améliorer l'efficience énergétique du Cloud computing en rendant notamment les salles serveurs moins consommatrices de ressources : machines moins gourmandes, systèmes de refroidissements améliorés, optimisation des installations électriques, etc. De plus, de nombreux travaux se sont intéressés à optimiser l'utilisation des ressources informatiques dans les centres de données physiques (i.e. consolidation [VAN08a] [VAN08b]).

Le *PUE* (*Power Usage Effectiveness*) s'est aujourd'hui imposé comme un indicateur technique universellement reconnu pour traduire l'efficacité énergétique des centres de données. Le PUE consiste à mesurer l'efficacité d'utilisation de l'énergie qui alimente un centre de données. Il s'agit d'évaluer la quantité d'énergie totale consommée par le centre de données par rapport à la quantité d'énergie nécessaire au fonctionnement des équipements informatiques (i.e. sans considérer les lumières, les systèmes de refroidissement, etc.). Plus le résultat est proche du chiffre 1, plus le centre de données est considéré comme *éco-responsable* (i.e. efficient énergétiquement).

Dans le contexte du Cloud computing, la plupart des efforts visant à améliorer l'efficience énergétique se sont focalisés sur le matériel ou encore l'infrastructure c'est-à-dire les couches basses du Cloud. D'autres efforts, plus récents, visent à rendre les logiciels plus à l'écoute des contraintes environnementales et ainsi accroître leur efficience énergétique [SAM13] [Sax10]. Ces travaux partent du constat que des gains énergétiques sont aussi possibles sur les couches hautes du Cloud.

Paradoxe de Jevons

Jusqu'à l'avènement du Cloud computing, l'habitude consistait à installer chez le client une architecture surdimensionnée pour limiter l'impact d'un événement (e.g. panne d'un serveur). Cela coûtait cher à l'achat et en fonctionnement (i.e. consommation énergétique) mais était rarement capable de supporter les cas exceptionnels. Avec le Cloud, où la ressource est facturée à l'usage, la solution est d'activer de nouvelles machines pour absorber l'activité. Cela a parfois des effets indésirables comme l'augmentation de l'empreinte énergétique et les coûts associés. Ainsi, sans adaptation réfléchie, le passage au Cloud n'est pas plus économique, ni moins gourmand en énergie. Le *Paradoxe de Jevons*, pourtant vieux de 150 ans, dit qu' *"à mesure que les améliorations technologiques augmentent l'efficacité avec laquelle une ressource est employée, la consommation totale de cette ressource tend à augmenter au lieu de diminuer"*. Bien que provenant d'un ouvrage traitant de l'utilisation du charbon, ce paradoxe reste tout à fait d'actualité.

> « *As technological improvements increase the efficiency with which a resource is used, total consumption of that resource may increase, rather than decrease.* » (W.S. Jevons - The Coal Question - 1865)

2.2 Élasticité du Cloud

2.2.1 Définitions

Environnement dynamique

L'élasticité est une caractéristique cruciale pour les fournisseurs SaaS dont les applications évoluent dans des environnements hautement variables. Il s'agit de répondre aux demandes des utilisateurs en temps normal mais aussi dans les périodes de forte activité avec une réactivité maximale. Les systèmes doivent être capables de supporter une montée en charge puis un retour à la normale, sans interruption de service, et si possible, sans répercussions sur le niveau de service, c'est-à-dire dans le respect des SLAs signés. Il existe d'autres événements imprédictibles bien que récurrents : pannes logicielles, matérielles, coupures de courant, etc. L'élasticité doit là aussi permettre d'amortir ces événements sans impacter le service rendu à l'utilisateur. Du point de vue du fournisseur, l'élasticité est un des rouages nécessaires à l'optimisation des ressources, tout en délivrant aux utilisateurs le niveau de service souhaité, dans un contexte d'exécution de plus en plus dynamique.

Passage à l'échelle

L'élasticité, présentée comme l'une des caractéristiques essentielles du Cloud computing [MG11], peut être corrélée à la notion de "passage à l'échelle" ou "scalabilité" (*scalability*). On parle aussi de "capacité à monter en charge" ou d'"extensibilité" pour définir la capacité d'un système à s'adapter aux dimensions du problème qu'il a à traiter. Dans le contexte du Cloud computing, la scalabilité correspond à l'aptitude d'une application à répondre à la montée en charge en ajoutant/retirant des ressources afin de maintenir un certain niveau de performance et de QoS. Concrètement, cela signifie que la puissance de calcul, la mémoire ou le stockage utilisés par le système peuvent facilement être augmentés ou diminués en fonction des besoins.

Dans leur article, les auteurs de [WHGK14] insistent sur la différence entre le concept de scalabilité et d'élasticité. Comme nous allons le voir par la suite, l'extensibilité est une condition nécessaire à l'informatique élastique mais le terme ne renvoie pas au caractère temporaire de cette augmentation de capacité. Ainsi, l'élasticité inclut la scalabilité, mais une application dite "scalable" n'est pas nécessairement une application élastique.

> « *Scalability just describes how much additional resources a system needs when the load increases to be able to offer a constant service level. Thus, scalability does not provide any information about the system's ability to scale resources on demand in a fast and accurate manner, it even does not make any assumptions about the existence of an automated scaling mechanism [WHGK14].* » (Andreas Weber et al. - 2014)

Élasticité

> « *Aptitude d'un corps à reprendre, après sollicitations, la forme et les dimensions qu'il avait avant d'être soumis à ces sollicitations.* » (Larousse)

Le concept d'élasticité étant au cœur du modèle du Cloud computing, de nombreuses définitions ont été proposées aussi bien par la communauté scientifique que industrielle. Parmi ces multiples définitions, nous avons retenu trois définitions récentes. Les deux premières définitions proposées par le *NIST* [MG11] et Nikolas Roman Herbst et al. [HKR13] insistent sur l'automatisation du processus avec les mots *automatically* et *autonomic*. De plus, la dimension temporelle est adressée avec les termes "at any time" ou "at each point in time" ainsi que notion de réactivité avec "rapidly". [HKR13] évoque la notion de précision du dimensionnement avec les termes "as closely as possible" indiquant que la quantité de ressources attribuée doit être au proche de la demande. Comme nous allons le voir par la suite, cette notion de précision du dimensionnement fait référence aux phénomènes néfastes de sur-dimensionnement et de sous-dimensionnement.

> « *Rapid elasticity. Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time [MG11].* » (NIST - 2011)

> « *Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible [HKR13].* » (Nikolas Roman Herbst et al. - 2013)

Les définitions d'élasticité proposées par [HKR13] et [WHGK14] mettent en lumière la distinction entre le concept de scalabilité et d'élasticité. En effet, bien que ces termes soient corrélés, Andreas Weber et al. indiquent que la notion d'élasticité va plus loin que la scalabilité dans le sens où celle-ci intègre la dimension temporelle de manière inhérente ainsi que la notion de qualité du processus d'adaptation. Ainsi, il s'agit d'ajuster la quantité de ressources en fonction de la demande (scalabilité), mais de manière optimale ce qui induit des propriétés comme la réactivité, la précision et plus généralement l'efficacité du processus d'adaptation. La préoccupation d'automatisation est indirectement évoquée avec la notion de mécanisme pour contrôler le processus d'adaptation.

> « *Scalability is a prerequisite for elasticity, but it does not consider temporal aspects of how fast, how often, and at what granularity scaling actions can be performed. Scalability is the ability of the system to sustain increasing workloads by making use of additional resources, and therefore, in contrast to elasticity, it is not directly related to how well the actual resource demands are matched by the provisioned resources at any point in time. [HKR13].* » (Nikolas Roman Herbst et al. - 2013)

> « *While scalability - in the cloud context - describes the degree to which a system is able to adapt to a varying load intensity by using a scaled resource amount, elasticity reflects the quality of the adaptation process in relation to load intensity variations over time. Thus, elasticity adds a temporal component to scalability. As elasticity describes properties of an adaptation process, elasticity requires the existence of a mechanism that controls the adaptation [WHGK14].* » (Andreas Weber et al. - 2014)

2.2.2 Motivations de l'élasticité

L'élasticité vise à gérer finement l'ajout et le retrait de ressources afin d'être toujours au plus proche de la demande et éviter selon les cas le *sur-dimensionnement* ou le *sous-dimensionnement*, deux aspects néfastes qui comme nous allons le voir touchent aussi bien le fournisseur de service que les utilisateurs de celui-ci (cf. Figure 2.7).

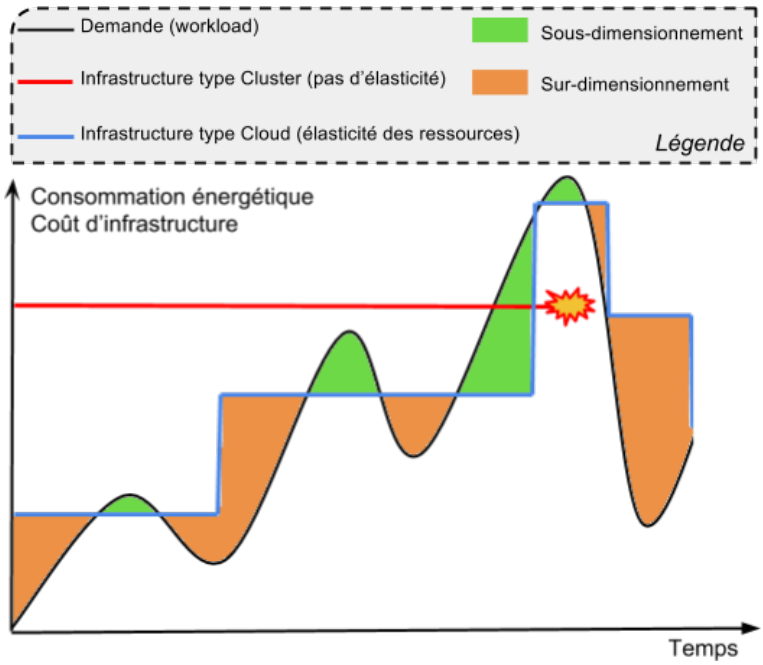


FIGURE 2.7 – Élasticité, sur-dimensionnement et sous-dimensionnement.

Sur-dimensionnement

On parle de sur-dimensionnement ou sur-approvisionnement (*over-provisioning*) lorsque la quantité de ressource (e.g. nombre de VMs) attribuée au système est supérieure à ses besoins. On peut voir sur la Figure 2.7 l'évolution de la demande (cf. courbe noire) dans le temps ainsi que la quantité de ressource allouée au système et ses répercussions d'un point de vue énergétique et financier. La courbe rouge indique la capacité en ressources d'une infrastructure de type *Cluster*, c'est-à-dire non élastique. On peut constater que ce type d'infrastructure est sur-dimensionnée la plupart du temps mais ne permet pas de gérer les événements exceptionnels (e.g. pic de charge important) ce qui peut dans certains cas aboutir à une interruption complète du service.

La courbe bleue, quant à elle, représente la quantité de ressource d'une infrastructure Cloud élastique. On peut voir que l'élasticité permet de suivre grossièrement la demande en ajoutant/retirant des ressources à l'exécution. Néanmoins, on peut constater que dans certains cas, la quantité de ressource reste supérieure aux besoins réels (cf. zone orange). Cela se traduit par un gaspillage énergétique et donc des coûts inutiles pour le fournisseur de service auxquels peuvent s'ajouter des frais de licence ou de maintenance superflus. L'objectif pour le fournisseur de service est donc d'éviter le sur-dimensionnement afin de limiter ses coûts et ainsi augmenter son profit.

Sous-dimensionnement

Le sous-dimensionnement ou sous-approvisionnement (*under-provisioning*) est aussi un problème, qui touche cette fois aussi bien le fournisseur de service que les utilisateurs de celui-ci. On parle de sous-dimensionnement lorsque la quantité de ressource attribuée au système est trop faible par rapport à la demande (cf. zone verte). Le manque de ressource va avoir un impact direct sur les performances et la QoS perçues par les utilisateurs du système. Par exemple, dans le cas d'une application SaaS, les utilisateurs vont constater des temps de réponse de plus en plus importants pour leurs requêtes ou pire encore, une indisponibilité du service.

Contrairement au sur-dimensionnement qui a un impact direct sur les coûts pour le fournisseur de service, le sous-dimensionnement va impacter les performances, la QoS et donc plus généralement le niveau de service. Cela va entrainer une insatisfaction des utilisateurs pouvant aboutir à une perte potentielle de clients pour le fournisseur et donc une baisse de profit. De plus, le fournisseur s'engage auprès de ses utilisateurs à fournir un certain niveau de service contractualisé dans un SLA signé par les deux parties. L'impact du sous-dimensionnement sur les performances et la QoS du système peut potentiellement entrainer la violation de certains SLOs et donc la nécessité pour le fournisseur de s'acquitter des pénalités associées.

Bilan

Le principal défi pour le fournisseur de services est de maintenir la satisfaction de ses consommateurs en assumant les SLAs signés (i.e. éviter le sous-dimensionnement), tout en minimisant le coût de ses services (i.e. éviter le sur-dimensionnement synonyme de gaspillage énergétique et financier). Dans l'idéal, il s'agirait pour le fournisseur de service d'attribuer exactement la bonne quantité de ressources au système à chaque instant. De manière schématique sur la Figure 2.7, cela reviendrait à obtenir une courbe bleue venant épouser la courbe noire de la demande en permanence et ce quels que soient les événements pouvant survenir à l'exécution du système. On parle parfois de "dimensionnement juste à temps" (i.e. *Just-in-Time Provisioning* [YQL09]).

2.2.3 Élasticité, la pratique

Nous avons pu voir précédemment que l'élasticité est la capacité d'un système à s'auto-redimensionner en fonction de la demande, c'est-à-dire ajouter/retirer des ressources à l'exécution en fonction du besoin courant. On distingue actuellement deux types de dimensionnement : le dimensionnement horizontal (*horizontal scaling*) et le dimensionnement vertical (*vertical scaling*). On parle parfois d'élasticité horizontale et verticale (cf. Figure 2.8).

Dimensionnement horizontal

Dans le contexte d'une infrastructure virtualisée, le dimensionnement horizontal, ou élasticité horizontale, consiste à ajuster le nombre de VMs en fonction de la demande. Ce type d'élasticité permet théoriquement d'augmenter la capacité de l'application de manière infinie [ABLS13]. Concrètement, on va ajouter ou retirer des VMs au groupement de ressource existant (*resource pool*). Dans le cas de l'élasticité horizontale, les termes *Scale Out* et *Scale In* sont utilisés dans la littérature pour définir respectivement l'ajout et le retrait de VM(s) (cf. Figure 2.8).

Le dimensionnement horizontal, que l'on retrouve parfois selon le terme "réplication", est proposé par tous les IaaS du marché. Ce type de dimensionnement est étroitement lié au mécanisme de répartition de charge (*load balancing*). Dans notre contexte virtualisé, il s'agit de répartir la charge de travail entres les différentes VMs du groupement de VMs. Cela se fait par le biais d'un répartiteur de charge (*load balancer*). Dans la Figure 2.9, on peut voir plusieurs VMs reliées au même répartiteur de charge qui a la responsabilité de distribuer équitablement la demande entre les différentes VMs.

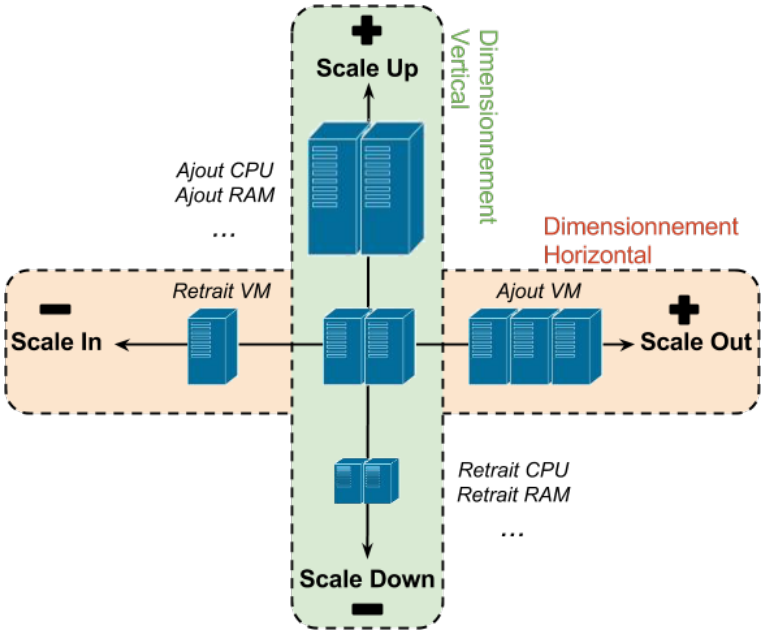


FIGURE 2.8 – Dimensionnement horizontal et vertical.

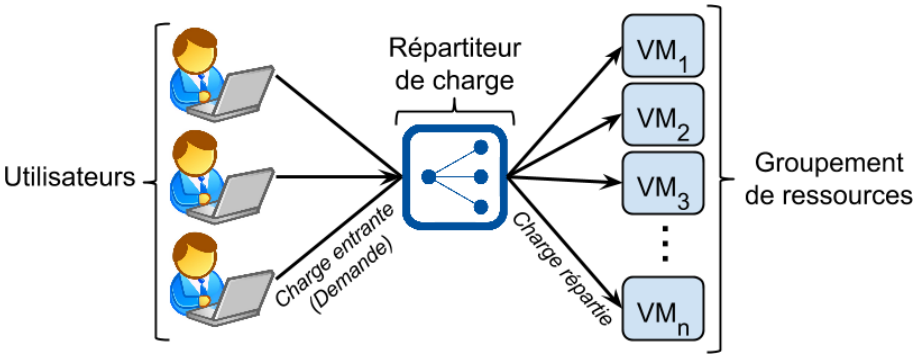


FIGURE 2.9 – Répartition de charge.

Dimensionnement vertical

La virtualisation permet une autre façon de dimensionner l'infrastructure : le dimensionnement vertical, aussi appelé élasticité verticale. On trouve parfois les termes anglais *resizing* ou *redimensioning* pour définir ce type de dimensionnement consistant à augmenter/diminuer les ressources d'une VM existante telles que le CPU, la mémoire ou encore la capacité de stockage. Ce type de dimensionnement, contrairement au dimensionnement horizontal, est limité par la quantité de ressource libre disponible sur le serveur physique hébergeant l'instance (i.e. PM). Concrètement, on ne peut pas attribuer davantage de ressources virtuelles aux VMs que de ressources physiques disponibles sur la PM hôte.

Dans le cas de l'élasticité verticale, les termes *Scale Up* et *Scale Down* sont utilisés dans la littérature pour définir respectivement l'ajout et le retrait de ressource (i.e. CPU, RAM, etc.) sur une VM existante (cf. Figure 2.8). En pratique, cela revient souvent à changer la taille d'une instance en paramétrant son offre (*VM offering*) en passant par exemple d'une *Small instance* à une *Large instance* proposant davantage de ressources CPU et/ou RAM. Ce changement d'offre nécessite généralement de redémarrer l'instance, on parle alors d'élasticité verticale à froid, et réciproquement d'élasticité verticale à chaud si l'on peut par exemple ajouter 2 Go de mémoire à une VM sans l'arrêter. L'hyperviseur (cf. section 2.1.2) prend en charge le redimensionnement des VMs en gérant l'allocation et la désallocation des ressources.

Migration

La migration élargit le champ d'application du dimensionnement [KF11] [SSSS11]. Elle consiste à déplacer une VM d'un hyperviseur à un autre afin de réorganiser la consommation de ressources. Ce mécanisme est largement utilisé pour la consolidation des centres de données [VAN08a] [VAN08b] qui vise à réduire les coûts d'exploitation du fournisseur IaaS. On distingue deux types de migration. La migration à froid consiste à éteindre la VM pour transférer son image disque sur un autre hyperviseur, une solution qui ne convient pas aux applications de haute disponibilité. La migration à chaud [CFH+05] pallie à ce problème en permettant de déplacer une VM sans l'arrêter. Par exemple, *vMotion* [vMo15] est un module de la suite *VMware vSphere* permettant de migrer des VMs à chaud sans interruption de service.

La migration est un mécanisme efficace permettant d'optimiser le placement des ressources virtuelles sur les ressources physiques. L'objectif final étant de minimiser la quantité de ressources physiques. Il s'agit d'éteindre les PMs superflues synonymes de gaspillage énergétique et financier. En ce sens, la migration peut être corrélée au concept d'élasticité visant à optimiser l'utilisation des ressources.

Bilan

L'élasticité horizontale, plus populaire que l'élasticité verticale, a fait l'objet de nombreux travaux scientifiques [BHD13] [DKM+11] [IKLL12] et existe dans toutes les solutions commerciales depuis plusieurs années [EC215b] [AZR15]. L'élasticité verticale, quant à elle, a vu le jour beaucoup plus récemment dans les solutions commerciales bien que certains travaux de recherche s'y intéressent depuis plusieurs années [MBS11] [GGW10] [PHS+09] [RBX+09]. De plus, l'élasticité verticale à chaud reste très rarement proposée par les IaaS et seulement quelques solutions offrent ce moyen de dimensionnement (e.g. *ProfitBricks* [Pro15]) en imposant souvent toute une pile technologique comme le choix de l'OS ou de l'hyperviseur [TL14].

2.3 Gestion de capacité et dimensionnement des ressources

Le dimensionnement désigne le fait de déterminer la quantité de ressources nécessaire pour un système. L'objectif est d'agir sur les ressources du système pour contrôler sa capacité de traitement et indirectement ses performances et sa QoS. Dans l'idéal, le processus de dimensionnement doit être automatisé, on parle alors de dimensionnement automatique (*autoscaling*). Le dimensionnement automatique peut être qualifié d'implémentation du concept d'élasticité du Cloud computing.

2.3.1 Différentes approches

On distingue trois manières de mettre en œuvre le dimensionnement automatique :

— *Dimensionnement Réactif* : dimensionnement automatique basé sur la demande et plus généralement sur l'état courant du système. Un dimensionnement dit réactif réagit aux changements mais ne les prévoit pas. En s'appuyant sur un service de surveillance (*monitoring*), chargé de mesurer l'utilisation des ressources ou encore la demande courante, le système peut indiquer des situations nécessitant d'augmenter ou de réduire la quantité de ressources.

— *Dimensionnement Pro-actif (Prédictif)* : le dimensionnement pro-actif vise à prévoir les besoins futurs en terme de ressources. Contrairement à l'approche réactive s'intéressant à l'état courant du système, on s'intéresse ici à son état futur. On parle aussi de dimensionnement prédictif pour qualifier le fait d'anticiper la demande en vue de fournir les ressources suffisantes à l'avance.

— *Dimensionnement Hybride* : combinaison d'un dimensionnement réactif et pro-actif. Ce type d'approche considère l'état courant et futur du système. Il s'agit d'être à la fois réactif aux changements mais aussi de prévoir à l'avance les besoins du système en termes de ressources ou encore d'anticiper les performances futures.

2.3.2 Théories et techniques employées : monde scientifique

Parmi les travaux existants, on distingue cinq théories principales utilisées dans le cadre de la gestion de capacité : la théorie des files d'attente (*queuing theory*), la théorie du contrôle (*control theory*), l'apprentissage par renforcement (*reinforcement learning*), l'analyse des séries chronologiques (*time series analysis*) et enfin les règles à base de seuil (*threshold-based rules*).

— *Théorie des files d'attente* : il s'agit d'une théorie mathématique provenant du domaine des probabilités [MADD04]. Cette théorie étudie les solutions optimales de gestion des files d'attente, aussi appelées queues, en s'appuyant sur des modèles analytiques. Il est possible de représenter un système par un ensemble de files d'attente, chaque file modélisant une ressource par exemple. Dans un système, une file d'attente se crée lorsque l'offre (e.g. ressources) est inférieure à la demande (e.g. clients). Une file d'attente est généralement définie par sa capacité, les instants d'arrivées des clients, les temps de service des clients, l'ordre dans lequel seront servis les clients et parfois le nombre maximum de client.
Le but de l'analyse des files d'attentes est de caractériser le degré de performance du système en répondant aux questions telles que combien de temps attend un client avant d'être servi en moyenne ? Quel est le nombre moyen de clients dans le système ? Quel est le taux d'utilisation moyen des serveurs ? Un modèle analytique basé sur la théorie des files d'attente peut prédire le comportement du système. La méthode d'analyse par valeur moyenne (*Mean Value Analysis* [RL80]) est la plus utilisée. Cette méthode permet de répondre aux questions évoquées ci-dessus en estimant par exemple les temps de réponse, la disponibilité ou le débit du système.

— *Théorie du contrôle* : théorie utilisée dans le domaine des mathématiques et de l'ingénierie (i.e. automatique) en vue d'étudier le comportement des systèmes qualifiés de dynamiques. L'objectif initial de cette théorie est de comprendre la façon dont une commande permet à un humain d'agir sur un système qu'il souhaite maîtriser. La théorie du contrôle analyse les propriétés d'un tel système dans le but de l'amener d'un état initial donné à un état final souhaité par le biais de la commande (contrôle) et en respectant éventuellement certains critères. Ainsi, un système de contrôle peut être vu comme un processus itératif visant à mesurer, comparer, appliquer et corriger. L'objectif peut être de stabiliser le système dynamique en le rendant insensible à certaines perturbations ou encore de l'optimiser selon certains critères ou contraintes [AETE12].

— *Apprentissage par renforcement* : l'apprentissage par renforcement [SB98] fait référence à une classe de problèmes d'apprentissage automatique. Le but est d'apprendre, à partir d'expériences passées, ce qu'il convient de faire en différentes situations. Il s'agit pour le système de raffiner sa manière de réagir en vue d'optimiser une récompense numérique au cours du temps. Le système va ainsi acquérir de manière automatisée des compétences dans sa prise de décision en fonction des échecs (récompense négative) ou des succès (récompense positive) constatés. Le système va ainsi suivre le processus suivant de manière itérative : observer les effets de ses actions, déduire de ses observations la qualité de ses actions et améliorer ses actions futures.

— *Analyse des séries chronologiques* : une série chronologique, aussi appelée série temporelle (*time series*), est une suite de valeurs numériques représentant l'évolution d'une quantité spécifique au cours du temps. Ces suites de valeurs numériques peuvent être formalisées mathématiquement en vue de les analyser d'un point de vue probabiliste ou statistique. Le but de l'analyse des séries chronologiques est d'étudier le comportement passé de ces suites en déterminant éventuellement des tendances (parfois appelé *pattern* [GGW10]) pour en prévoir le comportement futur. L'objectif premier des statistiques en séries temporelles est d'identifier la nature des dépendances temporelles dans les séries, c'est-à-dire la manière dont une valeur est liée aux valeurs précédemment observées.

Les différentes théories évoquées ci-dessus sont parfois combinées pour obtenir diverses variantes. D'autres approches s'appuient sur la théorie des jeux [NRTV07], la programmation par contraintes [RVBW06] ou encore la programmation linéaire [SH10] [ZPL+12]. Il faut préciser que les différentes théories évoquées ci-dessus sont majoritairement réservées au monde scientifique.

2.3.3 Règles à base de seuil : monde industriel

Les grands fournisseurs IaaS et PaaS du marché proposent des services de dimensionnement automatique (*auto-scaling*). On peut notamment citer *Amazon Auto-Scaling* [EC215a], *Microsoft Enterprise Library Autoscaling Application Block (WASABi)* [Mic15] ou encore *Google Compute Engine Autoscaler* [Goo15b] et *Rackspace Auto Scale* [Rac15]. Le point commun de tous ces services de dimensionnement automatique est qu'ils sont basés sur une approche réactive et plus précisément sur la méthode des "règles à base de seuils" (*threshold-based rules*).

Le but du dimensionnement automatique reposant sur des règles à base de seuils est simple, du moins en apparence. Il s'agit de définir des règles qui seront évaluées lors de l'exécution du système pour réguler l'ajout ou le retrait de ressources selon certaines conditions. Les règles sont parfois appelées politiques de dimensionnement automatique (*autoscaling policies*). Une règle est composée de deux parties : une *condition* et une *action*. Si la condition est vérifiée, l'action est exécutée. En ce sens, une règle de dimensionnement est proche de la structure de contrôle algorithmique *Si-Alors* (*If-Then*).

Les conditions portent généralement sur une agrégation de la valeur de certaines métriques collectées par un système de surveillance (*monitoring*) que l'on va confronter à des valeurs seuils. Le choix des métriques de décision, de la valeur des seuils ainsi que de la quantité de ressources à ajouter/retirer est de la responsabilité de l'opérateur humain en charge de la gestion de capacité du système. Comme nous le verrons plus tard, cette tâche s'avère délicate et nécessite une vraie expertise.

On distingue deux types de règles. Les règles visant à augmenter la capacité du système c'est-à-dire ajouter des ressources, et les règles visant à réduire la capacité du système en diminuant la quantité de ressources. Les règles de dimensionnement automatique suivent en général le *template* décrit dans le Tableau 2.1 tel que définit par [LBMAL12].

Tableau 2.1 – Template des règles à base de seuils.

Structure d'une règle	**si** *condition* **alors** *action*
Règle ajout de ressources	**si** $m > seuil_{haut,m}$ **pendant** $temps_{haut}$ **alors** ajouter k_{ajout}
Règle retrait de ressources	**si** $m < seuil_{bas,m}$ **pendant** $temps_{bas}$ **alors** retirer $k_{retrait}$

Dans la partie *condition* d'une règle de dimensionnement automatique (cf. Tableau 2.1), *m* correspond à la métrique (e.g. taux de consommation CPU) que l'on va confronter à $seuil_{haut,m}$ et $seuil_{bas,m}$ qui sont respectivement le seuil supérieur et le seuil inférieur définis pour la métrique *m* (e.g. 70% et 30%). Les deux périodes de temps $temps_{haut}$ et $temps_{bas}$ définissent la durée pendant laquelle la condition doit être remplie pour déclencher l'exécution de la partie *action* de la règle (e.g. 1 minute). Enfin, k_{ajout} et $k_{retrait}$ correspondent respectivement aux actions de reconfiguration d'ajout et de retrait de ressources à mettre en place si la condition est remplie sur la durée spécifiée. Dans le langage naturel, un exemple de règle peut être : "Si le taux de consommation CPU excède 75% pendant 1 minute, alors démarrer 2 nouvelles VMs".

Bien que n'apparaissant pas dans le *template* des règles à base de seuils (cf. Tableau 2.1), on trouve parfois la notion de "période de calme" ou de "refroidissement" (*cooldown period*). Cette période pouvant être associée à une règle désigne une durée d'attente après le déclenchement de celle-ci, c'est-à-dire suite à l'exécution de sa partie *action*. Il s'agit concrètement de désactiver la règle pour un certain temps afin que le système retrouve une certaine stabilité. Cela a souvent pour objectif d'attendre de constater le résultat de la reconfiguration précédente qui peut être sujette à une certaine latence (e.g. temps d'initialisation d'une VM). Il faut noter que les solutions industrielles ne sont pas les seules à utiliser les règles à base de seuils. En effet, certains travaux scientifiques comme *Cloudscale* [SSGW11] ou *Smartscale* [DGVV12] s'appuient aussi sur cette méthode pour la gestion du dimensionnement automatique. Nous reviendrons plus tard dans ce manuscrit sur les avantages et inconvénients des différentes approches (cf. chapitre 3).

2.4 Informatique autonomique

Afin de faire face à l'environnement hautement dynamique induit par le Cloud (i.e. charge de travail variable, pannes, etc.), il devient indispensable de s'appuyer sur des modèles qui permettent de réagir au contexte d'exécution afin de garantir la performance et la fiabilité du Cloud. De plus, la complexité croissante des systèmes informatiques rend la tâche d'administration impossible pour un être humain. Il devient donc nécessaire de rendre les systèmes informatiques capables de se gérer de façon autonome.

2.4.1 Définitions, motivations et propriétés

L'informatique autonomique a été introduite en 2001 par *IBM* [Hor01] comme réponse à la difficulté d'administration des systèmes informatiques. En effet, la complexité croissante des systèmes informatiques (i.e. en termes d'envergure, d'architecture, de technologies, etc.) ainsi que le contexte hautement dynamique dans lequel ils évoluent (i.e. sollicitations variables, pannes logicielles/matérielles, etc.) a rendu la tâche d'administration ardue voir même impossible pour l'Homme. L'objectif principal de l'informatique autonome est de soulager l'opérateur humain en charge de la gestion du système, en l'assistant dans les tâches d'administration. Dans [Hor01], Paul Horn définit les 7 commandements indispensables pour un système informatique autonomique :

1. Apprendre à se connaître lui-même ;
2. Détecter les pannes et les éviter ;
3. Réparer les pannes et les erreurs commises ;
4. Garder la maîtrise des données ;
5. S'optimiser en s'auto-éduquant ;
6. Comprendre les intentions des humains ;
7. Exister et évoluer dans un environnement ouvert et hétérogène.

L'Informatique autonomique a ensuite été définie en 2003 par *Jeffrey O. Kephart* dans un article fondateur du domaine : *A vision for autonomic computing*. Dans cet ouvrage, [KC03] dresse un parallèle entre le fonctionnement du système nerveux humain et les systèmes du domaine informatique. Ainsi, un système informatique autonome pourrait se comporter comme le corps humain qui intègre des propriétés autonomiques lui permettant par exemple de réguler le rythme cardiaque en fonction de l'effort qu'il fournit.

> « *Autonomic computing is the ability of an IT infrastructure to adapt to change in accordance with business policies and objectives [KC03].* » (Kephart et al. - 2003)

Auto-gestion (*self-management*)

Un système peut être qualifié d'auto-administré (*self-managed*) s'il met en œuvre les 4 paradigmes définis par [KC03] ([BBC+03] [HM08]) :

— *Auto-configuration* (self-configuration) : capacité du système à s'adapter dynamiquement et automatiquement aux variations de l'environnement d'exécution, en suivant des objectifs de haut niveau, indiquant les états désirables du système. Ainsi, le système est en mesure de s'installer, de se paramétrer ou de se mettre à jour tout seul, en fonction de son état courant ou de ses besoins, en suivant éventuellement certaines recommandations pré-définies en amont par l'être humain.

— *Auto-optimisation* (self-optimization) : capacité du système à optimiser en permanence l'utilisation de ses ressources. Cette optimisation peut être guidée par des politiques de gestion définies par l'être humain et dont le but est de spécifier les critères concernés par l'optimisation (e.g. consommation énergétique, nombre de requêtes traitées par seconde, etc.). L'objectif est de s'assurer que le système reste dans un état optimal ce qui induit par exemple le fait d'éviter le gaspillage de ressources.

— *Auto-guérison* (self-healing) : capacité à détecter, diagnostiquer puis compenser ou réparer des anomalies survenant dans le système au cours de son fonctionnement. Le système est en mesure de détecter une défaillance ou une panne (e.g. matérielle, logicielle) et d'initier une réparation par lui-même.

— *Auto-protection* (self-protection) : capacité défensive du système qui vise à anticiper les problèmes et ajuster son comportement pour se protéger des événements qui peuvent altérer son état et ainsi le déstabiliser. Cette propriété doit lui permettre de réagir à tout types de menaces et d'anticiper les attaques dans le but de prendre les précautions adéquates. Cette capacité est étroitement liée à la notion de sécurité.

2.4.2 Boucle de contrôle autonomique

Dans cette section, nous présentons un modèle de référence [KC03] introduit par *IBM* au début des années 2000 visant à atteindre les objectifs d'une informatique autonomique présentés dans [Hor01]. Il s'agit de la boucle de contrôle autonomique *MAPE-K* (*Monitor, Analyze, Plan, Execute - Knowledge*) qui s'inspire des boucles de contrôles utilisées dans le domaine de l'automatique (cf. Figure 2.10).

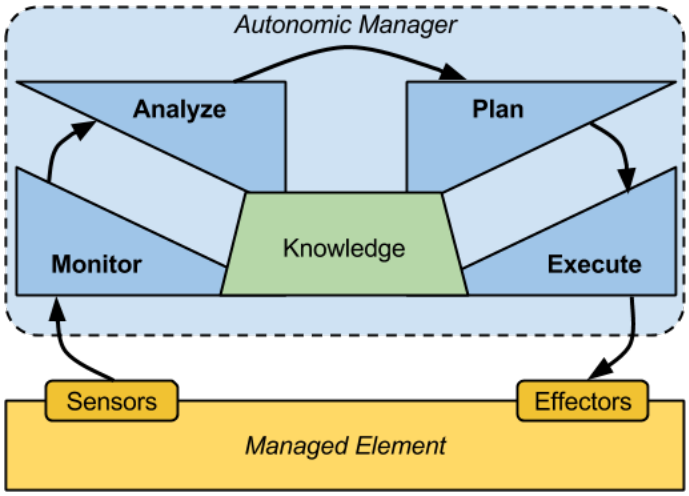


FIGURE 2.10 – Boucle de contrôle autonomique *MAPE-K*.

Éléments gérés (*managed element*)

Les éléments gérés représentent le système dont on attend un comportement autonomique en le couplant avec un gestionnaire autonomique. Il s'agit concrètement des ressources logicielles ou matérielles constitutives du système que l'on souhaite administrer de manière autonome par le biais de la boucle *MAPE-K*.

La communication entre les éléments gérés et le gestionnaire autonomique se fait au travers de deux interfaces. La première interface va adresser la surveillance et la collecte d'informations (introspection), tandis que la seconde interface va concerner la reconfiguration et l'application des changements sur le système géré (intercession).

La première interface correspond aux capteurs, ou sondes, (*sensors* ou *probes* dans la littérature) qui collectent des informations, ou métriques, sur les éléments gérés. Ainsi, les métriques représentatives de l'état de santé du système sont exposées au gestionnaire autonomique. Ces informations sont cruciales pour assumer un comportement autonomique. Idéalement, il est nécessaire d'avoir une parfaite connaissance du système et de son environnement ce qui implique que chaque élément géré doit être en mesure de fournir des métriques représentatives de son état.

La seconde interface correspond aux actionneurs (*effectors* ou *actuators* dans la littérature). Cette interface adresse la prise en charge des changements à effectuer sur les éléments gérés. Ces changements peuvent admettre différents niveaux de granularité. Concrètement, il s'agit des leviers possibles pour reconfigurer le système. L'ensemble des actionneurs constituent l'*API* de reconfiguration du système.

Gestionnaire autonomique (*autonomic manager*)

Le gestionnaire autonomique ne déroge pas au 6^{eme} commandement de *Paul Horn* concernant la compréhension des intentions humaines. En effet, un gestionnaire autonomique est un composant logiciel qui doit être configuré manuellement par des administrateurs humains capables de spécifier leurs intentions, aussi appelées stratégies, sous forme de politiques de haut niveau.

Le gestionnaire autonomique est alimenté par les métriques collectées par les multiples capteurs des éléments gérés et lui offrant les informations nécessaires sur l'état du système dans sa globalité. Il peut ensuite analyser toutes ces informations et décider de (re)configurer, d'optimiser, de guérir ou encore de protéger (4 objectifs de l'auto-gestion) le système sous gestion. Enfin le gestionnaire autonomique va appliquer le résultat de son diagnostique par le biais des actionneurs. Nous détaillons ci-dessous les différentes composantes d'une boucle *MAPE-K* :

— *Phase d'observation* : La première phase de la boucle *MAPE-K*, appelée phase d'observation (*Monitoring - M*), est en charge de la capture des informations exposées par les éléments gérés au travers des capteurs. Cette phase *M* s'interface donc avec le système dont on souhaite un comportement autonomique en collectant les métriques des ressources. Cette phase de collecte d'informations est essentielle car c'est par ce biais que le gestionnaire autonomique va avoir conscience de l'état de santé courant du système (i.e. des éléments gérés).

— *Phase d'analyse* : La seconde phase de *MAPE-K*, qui succède la phase d'observation, est la phase d'analyse (*Analyze - A*). Elle en charge de prendre en compte les données d'observation issues de la phase *M* pour les confronter aux politiques et stratégies globales de gestion définies au préalable. Il s'agit de comparer l'état courant du système à l'état optimal attendu. En d'autres termes, le *A* est chargé d'identifier quelles actions sont à entreprendre en vue d'"améliorer" l'état du système. L'objectif final de cette phase est de prendre une décision sur le(s) changement(s) à apporter au système.

— *Phase de planification* : La phase de planification (*Plan - P*) qui succède à la phase d'analyse, est en charge de définir une série de changements à effectuer sur les éléments gérés et ce à partir du résultat de l'analyse. Il s'agit concrètement de traduire le diagnostic de l'analyse et de préparer un plan complet des actions à mettre en œuvre sur le système. Le plan de reconfiguration doit être complet et précis. Il doit aussi adresser les aspects temporels (e.g. ordonnancement des actions) ou les actions prioritaires en essayant par exemple de minimiser le temps de reconfiguration. Les phases d'analyse et de planification sont parfois regroupées et considérées comme une unique *phase de décision*.

— *Phase d'exécution* : La phase d'exécution (*Execute - E*), dernière étape de la boucle *MAPE - K*, est chargée d'appliquer le plan de reconfiguration (issu de *P*) sur le(s) élément(s) gérés. Cela se fait par le biais d'appel(s) aux actionneurs qui opèrent les modifications nécessaires sur le système. La phase d'exécution est donc le résultat du traitement effectué par la boucle autonomique *MAPE-K*.

— *Base de connaissances* : Les différentes phases de la boucle autonomique *MAPE-K* sont reliées à la base de connaissances (*Knowledge - K*). Cette base de connaissances peut contenir un large panel de données relatives au système ou à l'environnement dans lequel il évolue. Elle peut être peuplée automatiquement ou manuellement par différentes sources. Le *K* peut par exemple contenir des historiques de la phase d'observation, les reconfigurations passées ou encore des informations rajoutées par des experts humains pouvant aider le gestionnaire autonomique dans sa prise de décision. Les quatre phases *MAPE* ainsi que l'administrateur humain peuvent interagir avec cette base de connaissances aussi bien en lecture qu'en écriture.

2.5 Conclusion

Dans ce premier chapitre de l'état de l'art, nous avons présenté le contexte dans lequel s'inscrit le sujet de thèse en introduisant de nombreuses définitions, concepts de base et technologies. Nous avons commencé par présenter le modèle ainsi que les grandes définitions du Cloud computing. Ensuite, nous avons développé et illustré la notion d'élasticité du Cloud computing au cœur de cette thèse. Enfin, dans le contexte hautement dynamique et complexe que représente l'informatique en nuage, nous nous sommes arrêté sur les intentions du domaine de l'informatique autonomique visant à rendre les systèmes capables de s'auto-administrer.

3

État de l'art scientifique

L'objectif de ce chapitre est de recenser et d'analyser plusieurs travaux liés au problème d'élasticité pour l'informatique en nuage. Dans la section 3.1, nous rappelons brièvement la problématique de cette thèse et donnons nos critères d'évaluation des différents travaux scientifiques. Nous recensons ces travaux que nous classons dans trois catégories. Les deux premières catégories s'intéressent aux travaux centrés sur la décision d'adaptation et la qualité du dimensionnement en termes de précision et de réactivité. La première (cf. section 3.2) concerne la couche IaaS seulement tandis que la seconde (cf. section 3.3) s'intéresse aux autres couches du Cloud (PaaS/SaaS). La troisième catégorie (cf. section 3.4) regroupe quant à elle les travaux portant sur la gestion de l'élasticité, c'est-à-dire les solutions visant à simplifier, outiller ou plus généralement gérer le processus de dimensionnement automatique. Enfin, nous terminons ce chapitre par une conclusion sur l'évaluation des travaux ainsi qu'une ouverture sur les challenges adressés par cette thèse (cf. section 3.5).

Contents

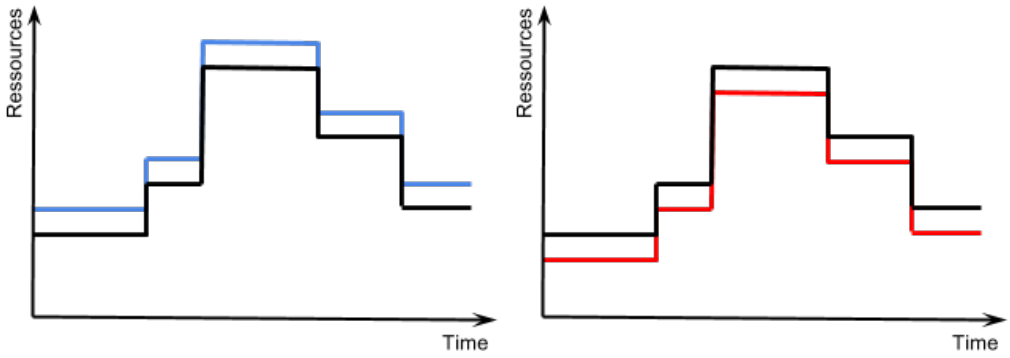
3.1 Positionnement des travaux

3.1.1 Problématique

L'élasticité a été définie par Nikolas Roman Herbst et al. [HKR13] comme le degré selon lequel un système est capable de s'adapter à une demande variable en ajoutant ou retirant des ressources de manière automatique afin que la quantité de ressource disponible soit en permanence au plus proche de la demande. Cette définition induit deux caractéristiques essentielles attendues d'un processus de gestion de l'élasticité. Il s'agit de la *précision* du dimensionnement et de la *réactivité* du dimensionnement [WHGK14]. Il est nécessaire d'adresser ces deux problématiques afin d'éviter les aspects négatifs du sur-dimensionnement et du sous-dimensionnement évoqués dans la sous-section 2.2.2.

Précision de l'adaptation

La précision de l'adaptation revient à fournir la bonne quantité de ressource. Il s'agit d'éviter de fournir trop de ressource synonyme de gaspillage énergétique et financier (i.e. sur-dimensionnement) mais aussi d'éviter d'en fournir trop peu ce qui a des répercussions néfastes sur les performances et la QoS (i.e. sous-dimensionnement). La Figure 3.1 met en avant deux systèmes élastiques capables de réagir instantanément à une demande variable (cf. courbe noire). Bien que la réactivité d'adaptation des deux systèmes soit idéale, la quantité de ressource attribuée, à savoir la précision d'adaptation, est en revanche critiquable. Dans le cas de la Figure 3.1a, le système attribue une quantité de ressource toujours supérieure à la demande ce qui va permettre d'assumer les engagements en termes de performances et de qualité de service (SLA) aux dépens des coûts induits par ce sur-dimensionnement. En revanche, la Figure 3.1b représente un système essentiellement sous-dimensionné permettant des économies sur les coûts d'infrastructure au détriment des performances et de la satisfaction des utilisateurs. Attention néanmoins à considérer les pénalités financières importantes induites par les manquements aux SLAs pouvant rapidement recouvrir les économies d'infrastructure effectuées.



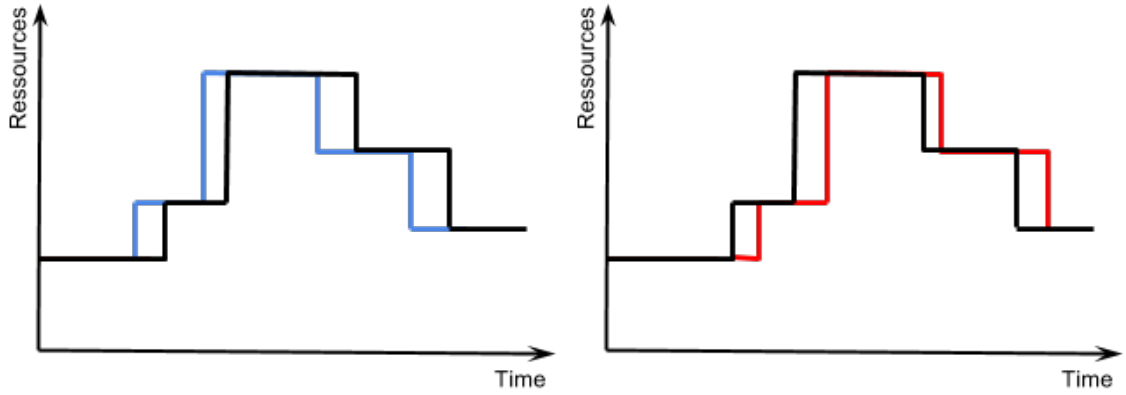
(a) (b)

FIGURE 3.1 – Précision du passage à l'échelle.

Réactivité de l'adaptation

Le second point essentiel de l'élasticité concerne la dimension temporelle et plus précisément la réactivité de l'adaptation face au contexte dynamique. Il s'agit de la capacité du système à réagir rapidement, dans l'idéal instantanément (i.e. *just-in-time provisioning* [YQL09]), au changement. On retrouve dans la littérature les termes *scaling speed* [HKR13] [SSJL12], *timing* [WHGK14] ou encore *elasticity time* pour définir cette dimension temporelle du passage à l'échelle.

La Figure 3.2 illustre cet aspect à travers deux systèmes dont la précision d'adaptation en termes de quantité de ressource est idéale mais dont le *timing* de l'adaptation admet des failles. En effet, le système de la Figure 3.2a est toujours en avance dans le passage à l'échelle ce qui peut résulter d'une politique d'adaptation trop agressive. À l'inverse, le système de la Figure 3.2b admet un retard perpétuel dans son adaptation ce qui peut s'expliquer par une politique d'adaptation trop lâche ou encore le temps d'initialisation des ressources non négligeable dans le cas d'un dimensionnement horizontal vers le haut (i.e. *Scale Out*). La réactivité du passage à l'échelle est une caractéristique essentielle dans le contexte hautement dynamique dans lequel évolue les systèmes.



(a) (b)

FIGURE 3.2 – Réactivité du passage à l'échelle.

Précision et Réactivité

Nous avons évoqué de manière indépendante l'importance de la précision du dimensionnement et de la réactivité du passage à l'échelle pour une élasticité efficace. Néanmoins, ces deux dimensions ne sont pas orthogonales et les systèmes sont généralement confrontées à plusieurs des défauts évoquées ci-dessus. En effet, un mauvais *timing* dans l'adaptation du système va par exemple contribuer à un sur-approvisionnement ou un sous-approvisionnement. Comme nous allons le voir par la suite, bien que les deux dimensions soient corrélées, la plupart des travaux scientifiques visent à améliorer soit la précision d'adaptation, soit la réactivité de l'adaptation. Cependant, certaines solutions tentent d'adresser ces deux problématiques essentielles de l'élasticité.

Perfectionner l'élasticité du Cloud

L'objectif général poursuivi par cette thèse est d'améliorer la capacité d'élasticité du Cloud dans sa globalité. Malgré les avantages indéniables de l'élasticité, le modèle actuel du Cloud est sujet à certaines limitations (physiques, conceptuelles et techniques) empêchant de jouir pleinement des bienfaits de celle-ci :

1. *Les ressources sont limitées*, ce qui signifie que l'on ne peut pas passer à l'échelle de manière infinie comme le promet le Cloud dans son argumentaire [ABLS13]. En effet, l'approvisionnement en ressource apparait souvent illimité pour l'utilisateur lui donnant l'impression de pouvoir demander n'importe quelle quantité de ressource à n'importe quel moment. Néanmoins, dans la pratique, il existe des cas où la quantité de ressource peut être limitée.

   Outre les limites imposées par les ressources physiques elles-mêmes (e.g. PM, énergie), les fournisseurs de IaaS bornent généralement le nombre maximum d'instances (i.e. VMs) pouvant être allouées pour un utilisateur dans une zone afin de faciliter la tâche d'administration des centres de données. C'est d'ailleurs le cas d'*Amazon EC2* [EC215b] qui limite à 20 le nombre d'instances par utilisateur par zone.

La quantité de ressource accessible peut aussi être restreinte en cas de panne (e.g. réseau). Enfin, le client peut lui-même contraindre l'approvisionnement pour causes de limite budgétaire ou de certaines contraintes architecturales. Ainsi, la quantité maximale de ressource accessible peut parfois être atteinte et/ou imposée par le contexte, le matériel, le fournisseur IaaS ou encore par des restrictions provenant des clients eux-mêmes.

2. *Le temps d'initialisation des ressources informatiques n'est pas négligeable*, ce qui induit parfois un manque de réactivité face aux changements d'états du système. En effet, le temps nécessaire au démarrage d'une instance et de tous les services associés (OS, *Tomcat*, *MySQL*, etc.) peut prendre plusieurs minutes [MH12] empêchant d'atteindre l'idéal du "dimensionnement juste-à-temps" (*just-in-time provisioning* [YQL09]). La réactivité du passage à l'échelle est une caractéristique essentielle dans le contexte hautement dynamique dans lequel évolue les systèmes. Malheureusement, comme l'indique la citation ci-dessous, le temps d'initialisation des ressources virtuelles entrave la réactivité du passage à l'échelle.

   > « *My biggest problem is elasticity. VM spin-up time... Ten to twenty minutes is just too long to handle a spike in Yahoo's traffic when big news breaks such as the Japan tsunami or the death of Osama bin Laden or Michael Jackson [Yah11].* » (Todd Papaioannou, Vice Président des architectures Cloud de Yahoo - 2011)

   Dans le cas d'applications ayant une charge de travail irrégulière variant rapidement (*fluctuating*, *oscillating workload*), le temps d'initialisation non-négligeable des ressources peut conduire à un phénomène connu sous le nom d'effet *ping-pong*. Il s'agit de démarrer continuellement des ressources puis de les arrêter sans même les avoir sollicitées, ce qui a un impact considérable sur les coûts d'infrastructure. Cela s'explique par le fait que les ressources sont "prêtes à l'emploi" trop tardivement par rapport à un besoin ponctuel ce qui se traduit par un manque de réactivité ne permettant pas de réagir immédiatement à un changement de demande.

   En théorie, le dimensionnement vertical à chaud, c'est-à-dire sans interruption de service, permet d'adresser cette limitation (cf. sous-section 2.2.3). Néanmoins, ce type de dimensionnement est très rarement proposé par les IaaS ou alors en imposant toute une pile technologique comme le choix de l'OS ou de l'hyperviseur [TL14]. De plus, malgré la flexibilité offerte par la virtualisation, le dimensionnement vertical est plus difficile à mettre en œuvre que le dimensionnement horizontal et reste limité par la quantité de ressource physique de la PM (cf. limitation précédente).

3. *La granularité du modèle de facturation est trop importante*, ce qui est parfois synonyme de gaspillage financier pour le consommateur. En effet, la majorité des fournisseurs IaaS du marché ont adopté une facturation des ressources à l'heure ce qui induit parfois un gaspillage financier pour les utilisateurs ayant des besoins ponctuels et courts (i.e. inférieurs à une heure). Cet aspect néfaste va à l'encontre de la notion de paiement à l'usage (*pay-per-use*, *pay-as-you-go*) que prône le modèle du Cloud (cf. sous-section 2.1.3). En effet, les utilisateurs peuvent être facturés plus que ce qu'ils consomment vraiment. On retrouve le terme *partial usage waste* dans la littérature pour définir cette sur-facturation imposée par la granularité du modèle économique adopté par les grands IaaS du marché [JWW+14].

4. Enfin, malgré les récents efforts pour *améliorer l'efficacité énergétique* du matériel et plus généralement des centres de données, l'évolution de l'impact énergétique des nouvelles technologies de l'information reste préoccupante [BAB12]. Bien qu'il ne s'agisse pas d'une limitation directe, il s'agit d'un paramètre à prendre en compte dans la gestion de l'élasticité. En effet, les ressources énergétiques ne sont pas infinies (cf. première limitation) et le prix croissant de l'énergie peut aussi indirectement limiter l'approvisionnement en ressources.

Les limites du modèle d'élasticité actuel évoquées ci-dessus ne permettent pas d'atteindre pleinement les objectifs théoriques initiaux de l'élasticité. En effet, les systèmes actuels peuvent être qualifiés de partiellement élastiques du fait du manque de flexibilité (i.e. granularité trop importante) et de réactivité du passage à l'échelle. Cela se traduit par des systèmes sur-dimensionnés et/ou sous-dimensionnés qui peinent à s'adapter au contexte hautement dynamique dans lequel ils évoluent. *Paul Brebner* a d'ailleurs évoqué ces deux dimensions dans [Bre12] pour définir une élasticité "parfaite" en ajoutant la notion de granularité de la facturation (i.e. paiement à l'usage).

> « *To be perfectly elastic, the resources exactly match the demand (no more or less), there is no time delay between detecting load changes and changing resourcing levels (resourcing is instantaneous), and you only pay for what you consume (charging is fine-grain consumption based) [Bre12].* » (P. Brebner)

3.1.2 Critères de sélection et de classification des travaux

Nous allons discuter des différents critères considérés dans la sélection et la classification des travaux scientifiques étudiés. Nous nous sommes appuyés sur des articles de type *survey* portant sur l'élasticité [GB12] ou le dimensionnement automatique [LBMAL14] [ALHL14], pour identifier certains critères "classiques" du domaine, que nous avons parfois redéfinis, et auxquels nous avons rajouter nos propres critères. Ainsi, dans l'analyse de l'état de l'art scientifique, nous rattachons les travaux aux différents critères présentés ci-dessous. Enfin, nous proposons un tableau récapitulatif des travaux étudiés en les positionnant en fonction de ces critères (cf. Tableau 3.1).

Approche du dimensionnement

Le critère d'approche du dimensionnement, parfois présenté sous le terme *policy* dans la littérature [GB12], est lié aux interactions nécessaires pour l'exécution des actions de l'élasticité. L'élasticité peut être mise en œuvre de manière manuelle ou automatique. Dans le cas d'une implémentation automatique de l'élasticité, on distingue les approches dites réactives des approches dites pro-actives (ou prédictives).

- *Manuelle (M)* : l'approche manuelle signifie que l'utilisateur (e.g. administrateur) est responsable de l'adaptation dans sa globalité. Il est en charge des phases d'observation, de décision et de réaction. Néanmoins, il peut s'appuyer sur des interfaces de surveillance ou d'exécution, fournies par le fournisseur IaaS (e.g. *API HTTP*) et grâce auxquelles l'utilisateur peut interagir avec le système. L'approche manuelle, bien que répandue dans le monde industriel, s'éloigne du concept d'élasticité qui induit en théorie une notion d'automatisation du processus de dimensionnement [HKR13].

- *Automatique (A)* : cette approche implique que le processus de gestion de l'élasticité est automatisé. Le système (ou un autre système dédié à cette tâche) prend en charge le contrôle de l'élasticité (i.e. observation, décision et réaction), en conformité avec les paramètres préalablement définis par l'utilisateur/administrateur (e.g. contrainte budgétaire, SLA, etc.). L'approche automatique peut-être mise en place de manière réactive, pro-active ou hybride (i.e. réactive et pro-active).

  - *Réactive (R)* : l'approche réactive réagit à un changement d'état du système généralement lié à une variation de la charge/demande. Les solutions réactives, populaires dans le monde industriel, sont essentiellement basées sur des règles de type *Événement-Condition-Action* (*Event-Condition-Action - ECA*) aussi appelées règles à base de seuils (cf. sous-section 2.3.3). L'événement désigne un changement d'état du système, identifié par la surveillance (e.g. taux de consommation CPU), la condition exprime généralement un seuil portant sur les indicateurs observés (e.g. CPU > 75%) et l'action correspond à une reconfiguration des ressources du système (e.g. ajouter 2 VMs).

— *Pro-active/Prédictive (P)* : les solutions dites pro-actives ou prédictives visent à anticiper les besoins en ressources. Il s'agit d'ajuster la quantité de ressources en amont en s'appuyant sur des modèles permettant d'estimer les états et besoins futurs du système (e.g. évolution de la charge de travail, performances, etc.). L'objectif est de prédire à l'avance les besoins en termes de ressource en vue de reconfigurer le système avant qu'il ne soit trop tard (i.e. éviter de dimensionner trop tardivement l'infrastructure). Ce type d'approche, principalement issue du monde scientifique, permet notamment de limiter l'impact du temps d'initialisation des ressources décrit précédemment.

Dans notre état de l'art, nous nous intéresserons uniquement aux solutions automatiques. Certaines solutions sont qualifiées d'hybrides, c'est-à-dire qu'elles mettent en œuvre les deux approches décrites ci-dessus [AETE12] en s'appuyant par exemple sur du réactif pour l'approvisionnement de ressources et du pro-actif pour la libération de ressources [IDCJ11].

Portée de la surveillance (Observation)

Ce critère, que nous avons introduit, s'intéresse à l'observation du système dans les différents travaux. Il s'agit dans un premier d'identifier quels types de sondes ont été mises en place dans les différentes solutions mais surtout quelles sont les métriques considérées dans la décision d'adaptation et à quel niveau se situent-elles. Comme nous le verrons, certains travaux considèrent plusieurs métriques de différents niveaux pour la prise de décision de reconfiguration.

— *IaaS* : les métriques considérées dans la prise de décision d'adaptation sont des métriques de bas niveau associées à la couche IaaS ;
— *PaaS/SaaS* : les métriques considérées dans la prise de décision d'adaptation sont des métriques de haut niveau associées aux modèles de service PaaS et/ou SaaS.

Portée de l'adaptation et Méthode de dimensionnement (Adaptation)

L'élasticité permet d'ajouter ou de retirer des ressources informatiques en fonction des besoins du système. Dans le contexte virtualisé du Cloud computing, ces ressources informatiques sont généralement des ressources virtuelles (e.g. VM, RAM, CPU, etc.). L'élasticité du Cloud est essentiellement implémentée et mise en place sur la couche IaaS bien que l'on déporte parfois sa gestion sur la couche PaaS. Concrètement, les actions de dimensionnement portent sur des ressources IaaS et sont exécutées sur l'infrastructure.

Les différentes définitions d'élasticité (cf. sous-section 2.2.1) ne limitent pas la portée des actions de dimensionnement à la couche IaaS. En effet, le terme "ressource informatique" peut couvrir un large panel de ressources allant du *hardware* au code lui-même en passant par les ressources virtuelles ou les composants logiciels. Le critère de portée de l'adaptation revient à considérer le modèle de service sur lequel les actions d'adaptation sont exécutées. Bien que la très grande majorité des travaux revient à reconfigurer les ressources de la couche IaaS, certaines initiatives visent à intervenir sur les couches hautes du Cloud (i.e. PaaS/SaaS) et les ressources associées (e.g. répartiteur de charge, composants applicatifs, serveur *HTTP*, etc.) en reconfigurant par exemple les applications en fonction du contexte d'exécution. Il faut noter que certains travaux exécutent des actions de reconfiguration sur plusieurs couches.

— *IaaS* : les actions d'adaptation sont exécutées sur la couche IaaS. Nous avons précédemment évoqué les différentes méthodes de dimensionnement dans la sous-section 2.2.3. Il s'agit là-aussi d'un critère habituel pour classifier les travaux traitant de l'élasticité dans le Cloud [Let14]. Pour rappel, on distingue deux méthodes de dimensionnement au niveau IaaS, communément appelées élasticité horizontale et élasticité verticale, auxquelles vient s'ajouter le mécanisme de migration qui peut s'apparenter à un autre type de dimensionnement. Parmi les travaux existants, on retrouve parfois des solutions hybrides mêlant plusieurs méthodes de dimensionnement [HGGG12] [SHRE13] [DGVV12].

  - *Dimensionnement Horizontal (H)* : ajuster le nombre de VMs en fonction de la demande. On retrouve les termes *duplication* ou *replication* dans la littérature [GB12] pour faire référence à cette méthode de dimensionnement.
  - *Dimensionnement Vertical (V)* : augmenter/diminuer les ressources d'une VM existante telles que le CPU, la mémoire ou encore la capacité de stockage. On trouve aussi les termes *resizing* ou *redimensioning* dans la littérature [GB12].
  - *Migration (M)* : déplacer une VM depuis sa PM source vers une PM cible permettant ainsi de réorganiser la consommation de ressources.

- *PaaS/SaaS* : les actions d'adaptation sont exécutées sur les couches PaaS/SaaS.
  - *Reconfiguration Architecturale (A)* : adapter dynamiquement l'architecture de l'application. Les architectures logicielles sont généralement modélisées sous forme de graphe [CGB+02] où les nœuds représentent des composants logiciels (i.e. ressources) plus ou moins complexes et les arêtes, appelées connecteurs, représentent les interactions entre ces composants. La reconfiguration architecturale revient concrètement à modifier la structure du graphe représentant l'architecture de l'application en ajoutant/retirant des composants logiciels. Dans notre cas, nous nous intéresserons à des reconfigurations architecturales dynamiques [GCH+04] qui adressent les préoccupations de passage à l'échelle et d'élasticité.

  - *Paramétrisation du middleware et du Logiciel (P)* : modifier les paramètres de configuration des composants logiciels. Contrairement à la reconfiguration architecturale qui vise à modifier la structure du graphe de composants en ajoutant/retirant des nœuds à celui-ci, il s'agit ici de modifier les paramètres/attributs exposés par les composants existants. Un exemple de paramétrisation *middleware* peut être le changement de la valeur du paramètre *MaxClients* [DTM11] d'un serveur *Apache* [apa15] permettant de fixer la limite maximale de requêtes simultanées que le serveur peut prendre en charge ou encore la modification de la valeur du paramètre *session-timeout* de *Tomcat* [tom15] qui permet de définir la durée pendant laquelle la session d'un utilisateur reste active. Concernant le logiciel lui-même (i.e. application SaaS), la paramétrisation peut revenir à activer certaines fonctionnalités comme la lettre d'information (*newsletter*) ou encore les *flux RSS* (*Really Simple Syndication*).

Techniques et Théories employées (Décision)

Les techniques et théories utilisées pour la planification de la capacité constituent aussi un critère habituel pour classifier les travaux [LBMAL14]. Ces différentes techniques sont utilisées dans la partie décision du processus d'élasticité (que l'on peut rattacher à la phase d'*Analyse* de la boucle *MAPE-K*). L'objectif étant de répondre à la question "combien de ressource faut-il ajouter/retirer au système ?". On distingue cinq théories principales parmi les travaux existants (cf. sous-section 2.3.1) :

- *Théorie des files d'attente (Queuing Theory - QT)* ;
- *Théorie du contrôle (Control Theory - CT)* ;
- *Apprentissage par renforcement (Reinforcement Learning - RL)* ;
- *Analyse des séries chronologiques (Time Series analysis - TSA)* ;
- *Règles à base de seuils (Threshold-Based Rules - TBR).*

La liste des techniques évoquée ci-dessus n'est pas exhaustive. En effet, d'autres théories sont parfois préférées ou combinées à celles-ci en vue de prédire, contrôler ou modéliser le dimensionnement automatique. On peut par exemple citer la théorie des jeux, la programmation par contrainte ou encore la programmation linéaire.

Intention

Du point de vue du fournisseur, l'élasticité assure une meilleure utilisation des ressources informatiques permettant des économies d'échelle. Bien que tous les travaux étudiés tendent à améliorer l'efficience générale du système, ils admettent parfois des intentions distinctes (i.e. *purpose* [GB12]). Nous distinguons trois grandes intentions récurrentes pour lesquelles l'élasticité est étudiée :

— *Performances/QoS* : du point de vue de l'utilisateur, l'élasticité permet d'éviter l'insuffisance de ressource et donc la dégradation des performances du système ;
— *Coût* : réduire les coûts d'infrastructure du point de vue du fournisseur de service en optimisant l'utilisation des ressources ;
— *Énergie* : réduire la consommation énergétique globale du système ou accroître l'efficience énergétique du point de vue du fournisseur. Cette intention est étroitement liée à la réduction des coûts.

Accord de niveau de service - SLA

Ce critère de classification vise à indiquer les travaux qui intègrent explicitement (ou implicitement) la notion de SLA en précisant la (ou les) métrique(s) concernée(s). Généralement, les travaux dont l'intention porte sur les performances (cf. critère *intention* ci-dessus) s'intéressent à la métrique de temps de réponse (*response time* ou *latency*). Néanmoins, d'autres métriques de QoS sont parfois considérées comme la disponibilité (*availability*) qui correspond au taux de requêtes traitées "normalement" (lié au taux de requêtes tombées en erreur), ou encore le débit/rendement (*throughput*) généralement représenté par la métrique indiquant le nombre de requêtes traitées par seconde. D'autres travaux, s'intéressant aux intentions de coût ou d'énergie, vont considérer des SLOs portant sur des métriques associées à ces préoccupations comme le coût d'infrastructure, le budget, la consommation énergétique ou encore le coût de la reconfiguration.

L'état de l'art du domaine adressé dans cette thèse est assez large. De plus, la gestion de l'élasticité dans le Cloud est un sujet d'actualité pouvant être qualifié de *hot topic* ce qui se traduit par de nombreuses publications récentes sur cette thématique rendant parfois les anciennes publications "obsolètes". Ainsi, comme tout état de l'art, il a été nécessaire de recouper les différents critères évoqués ci-dessus avec certaines "méta-informations" essentielles parmi lesquelles on note l'année de parution du papier, le type de publication (journal, conférence, workshop, etc.) ainsi que la liste des auteurs et leurs affiliations respectives ou encore le nombre de citations.

3.2 Élasticité de l'infrastructure

Dans cette section, nous nous penchons sur les travaux s'intéressant aux problématiques de dimensionnement automatique sur la couche IaaS. Cela comprend un grand nombre de travaux que nous avons regroupé en trois catégories dépendamment des approches choisies à savoir une approche pro-active (cf. sous-section 3.2.1), réactive (cf. sous-section 3.2.2) ou hybride (cf. sous-section 3.2.3). Cette classification résulte du fait que le choix de l'approche de dimensionnement constitue selon nous un véritable parti pris scientifique (contrairement, par exemple, au choix de la méthode de dimensionnement : vertical ou horizontal). Enfin, pour chacune de ces catégories, nous avons regroupé les travaux en fonction des techniques et théories employées.

3.2.1 Solutions pro-actives

Nous nous intéressons ici aux solutions pro-actives, aussi appelées prédictives, qui tendent à prédire la charge de travail future en vue d'anticiper le dimensionnement.

Apprentissage par renforcement

Les travaux ci-dessous reposent sur la théorie d'apprentissage par renforcement (*Reinforcement Learning* [SB98] - *RL*) pour la prise de décision de dimensionnement (cf. sous-section 2.3.2).

Vasić et al ont proposé le *framework DejaVu* [VNM+12] pour la gestion des ressources virtuelles qui adresse essentiellement la préoccupation de réactivité du passage à l'échelle. Il s'agit globalement d'anticiper la demande (i.e. charge de travail) en vue de prendre automatiquement et rapidement les décisions d'adaptation (i.e. dimensionnement horizontal et vertical). Le but du *framework* est d'apprendre des expériences passées, les configurations de ressource ayant données les meilleures résultats (e.g. en termes de performance avec le respect d'un SLO sur les temps de réponse) en fonction des charges de travail constatées. Cela passe bien entendu par une phase d'apprentissage. *DejaVu* identifie et stocke différents profils de charge à partir des expériences passées (i.e. période d'entrainement/apprentissage). Chaque profil se voit attribuer une signature, générée par la relation entre la charge et les ressources qui ont été utilisées pour cette charge. Ainsi, lorsque le *framework* détecte que la demande varie à l'exécution, il est en mesure de rechercher le profil de charge courant parmi les différentes signatures stockées, et obtenir ainsi la configuration de ressource correspondante ayant donnée les meilleurs résultats.

Barrett et al [BHD13] proposent une approche basée sur le *Q-learning* (i.e. algorithme d'apprentissage par renforcement [WD92]) utilisant des agents d'apprentissage parallèles pour déterminer des politiques optimales d'allocation de ressource (i.e. dimensionnement horizontal). Le principe est le suivant (cf. Figure 3.3) : chaque VM se voit attribuer un agent qui a la charge de sa gestion (*learning agent*). Les différents agents constituent ainsi une architecture parallèle d'apprentissage par renforcement. Chaque agent prend ses propres décisions de manière indépendante en considérant sa demande entrante (suite à une répartition sur les différents agents) et son expérience individuelle passée (pénalités et récompenses des adaptations précédentes) en essayant d'apprendre une politique qui est individuellement optimale. Les agents partagent ensuite leurs observations en communiquant directement les informations aux autres agents. Le gestionnaire d'instance (cf. *instance manager*) est alors chargé d'exécuter les actions de dimensionnement en se basant sur les instructions de l'agent d'apprentissage. Le partage de l'information entre agents permet de réduire le temps nécessaire pour converger vers une politique globalement stable.

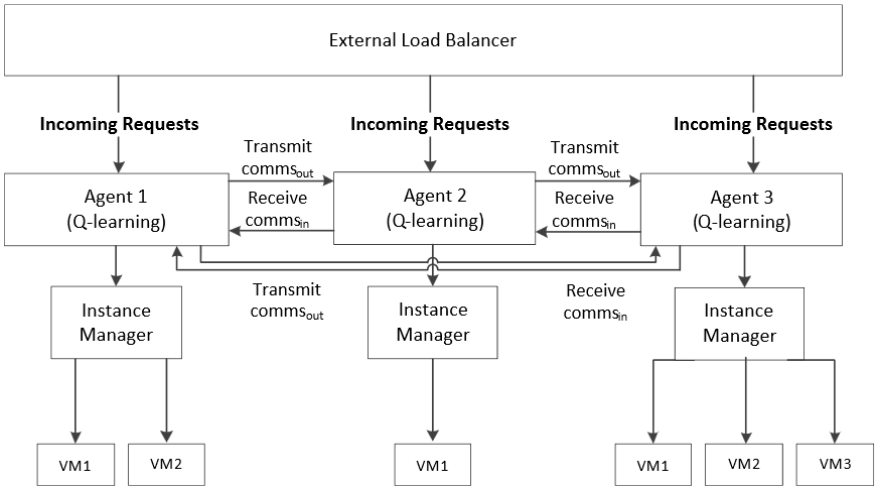


FIGURE 3.3 – Architecture parallèle de *Q-learning* proposée par [BHD13].

Rao et al ont proposé *VCONF* [RBX+09], une approche basée sur l'apprentissage par renforcement pour automatiser le processus de (re)configuration des VMs (i.e. dimensionnement vertical) en vue d'optimiser les performances globales de celles-ci. *VCONF* est un agent (cf. Figure 3.4) qui utilise des algorithmes basés sur l'apprentissage par renforcement permettant de diriger automatiquement la configuration de chaque VM vers une solution optimale (ou proche de l'optimal). Il s'agit concrètement d'ajuster indépendamment les différentes ressources des VMs (e.g. CPU et RAM). Pour chaque ressource considérée, il est possible d'augmenter la quantité, de la diminuer (selon des paliers prédéfinis) ou de ne rien faire. L'action prise en considération par le mécanisme d'apprentissage résulte dans la combinaison des différents choix pour chaque ressource (e.g. augmenter CPU et diminuer RAM). Le résultat (i.e. *score*) d'une action entreprise sur une VM est calculé en fonction des performances de celle-ci. Le score correspond au rapport du débit de sortie actuel (*throughput*) sur un débit de référence ("idéal") auquel on soustrait les éventuelles pénalités en cas de manquement au SLA (i.e. SLO portant sur le temps de réponse).

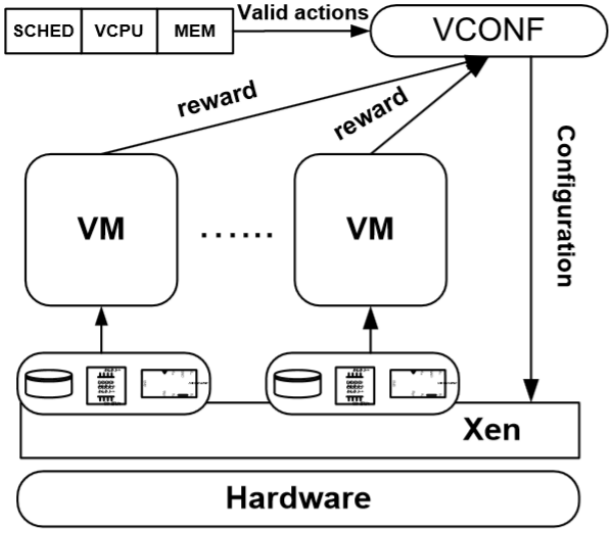


FIGURE 3.4 – Architecture de *VCONF* proposée par [RBX+09].

Théorie du contrôle

D'autres travaux font appel à la théorie du contrôle pour le dimensionnement automatique de ressources. La théorie du contrôle analyse les propriétés d'un système dynamique dans le but de l'amener d'un état initial donné à un état final souhaité par le biais de la commande (contrôle) tout en respectant certains critères ou contraintes (cf. sous-section 2.3.2).

Xu et al [XZF+07] proposent un système de gestion de ressources à deux niveaux permettant d'ajuster automatiquement les ressources virtuelles (i.e. dimensionnement vertical) en fonction de la demande. Il s'agit concrètement d'estimer le besoin CPU en fonction de la charge en vue de respecter les SLAs signés. L'approche est dite à deux niveaux dans le sens où le contrôle des ressources est réalisé sur deux niveaux d'abstraction : au niveau des VMs elles-mêmes et au niveau du groupement de VMs (*resource pool*). Le système intègre ainsi deux types de contrôleur : local et global. L'allocation (et désallocation) autonomique de ressource est réalisée à travers l'interaction entre ces deux contrôleurs. Un contrôleur local est créé pour chaque VM et a la responsabilité de déterminer la quantité de ressource nécessaire au bon fonctionnement de celle-ci en fonction de la charge de travail entrante puis de demander l'ajout ou le retrait de ressource (i.e. CPU). Le contrôleur local repose sur la logique floue (i.e. *fuzzy logic* [KY95]).

La logique floue permet de résoudre des problèmes de prise de décision dans lesquels on dispose de connaissances (i.e. données) imprécises, soumises à des incertitudes (e.g. charge de travail). Dans le cas du contrôleur local, il s'agit de modéliser le comportement de la VM sans connaissance particulière du système (i.e. approche boîte noire) en considérant uniquement ses données en entrée (i.e. demande) et en sortie (i.e. ressources). Ainsi, la VM apprend elle-même la relation entre la charge de travail qu'elle reçoit et la quantité de ressource nécessaire correspondante pour respecter la QoS spécifiée dans les SLAs. Un contrôleur global est créé pour l'ensemble des VMs (e.g. sur la PM hôte). Il est chargé de recevoir l'ensemble des requêtes des contrôleurs locaux en termes de ressource et d'attribuer les ressources en essayant de maximiser le profit total (en considérant les pénalités liées aux violations de SLA).

Padala et al ont proposé *AutoControl* [PHS+09], un système de rétroaction d'allocation de ressources (i.e. dimensionnement vertical) qui s'adapte automatiquement aux changements de charge de travail en vue d'atteindre certains SLOs portant sur les performances (e.g. temps de réponse, débit de sortie, etc.).

AutoControl est une combinaison d'un modèle de prédiction et d'un contrôleur adaptatif (cf. Figure 3.5). Le modèle de prédiction auto-régressif et moyenne mobile *ARMA* (*Auto Regressive Moving Average*) est utilisé. Il est capable de capturer les caractéristiques d'une série de données (i.e. identifier des *patterns*) et d'en prédire les valeurs futures. Le contrôleur adaptatif, quant à lui, est de type *MIMO* (*Multiple-Input, Multiple-Output*) c'est-à-dire multi-entrées (plusieurs sondes) multi-sorties (plusieurs commandes).

AutoControl attribue la capacité de ressource nécessaire pour garantir le SLA en détectant les goulots d'étranglement (e.g. CPU ou réseau). De plus, il fournit un niveau de différenciation des services en fonction de la priorité des demandes. Cette approche modélise un SLO par le triplet *priorité-métrique-cible*, où *priorité* indique la priorité de la demande, *métrique* correspond à la métrique de performance considérée et *cible* indique la valeur (i.e. seuil) pour cette métrique. Les auteurs s'intéressent à des métriques de plusieurs niveaux en considérant par exemple le CPU et les entrées/sorties disque pour les métriques de bas niveau (i.e. IaaS) et le temps de réponse moyen comme mesure de performance haut niveau (i.e. PaaS/SaaS).

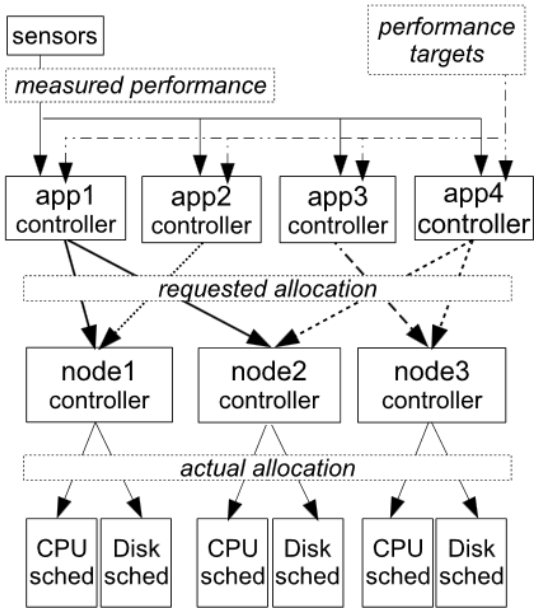


FIGURE 3.5 – Architecture logique de *AutoControl* proposée par [PHS+09].

Dans [GSL+12], Ghanbari et al s'intéressent à l'utilisation de politiques d'élasticité horizontale comme dans leur précédent article [GSLI11]. La solution respecte une approche de dimensionnement pro-active et se base sur la théorie du contrôle et des heuristiques en utilisant une prédiction par modélisation stochastique. Ghanbari et al proposent ainsi des politiques d'élasticité dérivées des objectifs de haut niveau spécifiés par le fournisseur d'application SaaS. Les objectifs de haut niveau peuvent prendre différentes formes selon les auteurs mais reviennent généralement à un souhait de minimiser, maximiser ou encore réguler certaines valeurs. Le but de ce travail est de satisfaire les objectifs du fournisseur tout en minimisant les coûts liés à l'allocation de ressources.

Les politiques d'élasticité définies dans [GSLI11] peuvent être qualifiées de règles à base de seuils étendues (cf. Figure 3.6) où vont être spécifiés, en plus des informations classiques (i.e. valeur du seuil, action d'adaptation, etc.), différents paramètres comme la fréquence d'évaluation des règles ou encore les périodes de calme à respecter suite à une adaptation. Ces différents paramètres permettent de définir des politiques d'élasticité réactives adaptées au contexte d'exécution de l'application (e.g. dépendamment du type de charge de travail).

Parameter	EP_1	EP_2	EP_3
resize numbers	1	1	2
decision threshold	51%	51%	51%
operating interval	[50,45]	[55,40]	[55,50]
decision duration	7 min	7 min	8 min
refractory_period	8 min	8 min	6 min
instance bounds	[2,20]	[2,20]	[2,20]

FIGURE 3.6 – Exemples de politiques d'élasticité définies par [GSLI11].

Théorie des files d'attente

Jiang et al [JLZL13] ont proposé une solution pro-active de dimensionnement automatique pour adresser au mieux le compromis coût-performance du point de vue du fournisseur d'application SaaS. Il s'agit d'adapter au plus juste le nombre de VM (i.e. dimensionnement horizontal) en vue de satisfaire les SLAs signés (i.e. temps de réponse). La décision de passage à l'échelle est prise à intervalle régulier et le résultat de cette décision peut être l'ajout de VM, le retrait de VM ou encore de ne rien faire (i.e. *no operation*).

En vue d'estimer le nombre optimal d'instances à allouer pour l'application, Jiang et al proposent une approche prédictive qui s'appuie sur l'analyse des séries chronologiques et les files d'attente. Il s'agit d'analyser l'historique des charges de travail passées pour identifier des *patterns* dans l'évolution de la demande et ainsi prédire celle-ci à l'exécution. Une fois la prédiction de la demande effectuée, les auteurs reposent sur la théorie des files d'attente (i.e. *M/M/m queue*) et l'optimisation multi-objectif pour identifier l'allocation de ressource optimale dans le but d'obtenir le meilleur compromis coût-performance.

Analyse des séries chronologiques

Nous nous intéressons ici aux solutions de dimensionnement automatique reposant sur l'analyse des séries chronologiques qui vise à prédire l'évolution de certaines métriques dans le temps (cf. sous-section 2.3.2). De ce fait, toutes les solutions analysées ci-dessous peuvent être qualifiée de pro-actives. Certains des travaux étudiés se concentrent uniquement sur le dimensionnement horizontal [CHL+08] [IKLL12] ou le dimensionnement vertical [GGW10] [SSGW11]. D'autres travaux s'intéressent à l'utilisation conjointe de ces deux méthodes de dimensionnement [FLWC12] [DGVV12] ou encore au mécanisme de migration [FLWC12] pour la gestion de capacité.

Les travaux de Chen et al [CHL+08] se concentrent sur la préoccupation de consommation énergétique dans le cadre du dimensionnement horizontal de ressources IaaS. Ils admettent une approche de dimensionnement pro-active en s'appuyant sur une méthode de prévision de charge à court terme reposant sur l'analyse des séries chronologiques (i.e. *auto-regression method*). Ils s'intéressent aux applications web avec de très nombreuses connections et s'appuient sur le cas d'utilisation de *Windows Live Messenger* (i.e. *traces* et *logs*).

Dans un premier temps, les auteurs fournissent un modèle de consommation énergétique, de performance et d'expérience utilisateur (QoS). Ils répertorient ensuite différents algorithmes de répartition de charge dont ils évaluent et analysent l'impact des différentes algorithmes en termes de consommation énergétique et de QoS fournie aux utilisateurs (i.e. disponibilité). Ils donnent les briques conceptuelles d'un *framework* qui permet de jouer sur le compromis énergie-QoS en changeant d'algorithme de répartition de charge.

Ils proposent un algorithme de répartition de charge "contre-intuitif" appelé *Load Skewing* (i.e. "charge biaisée"). Par opposition aux algorithmes de répartition de charge classiques qui visent à répartir uniformément la charge sur les multiples serveurs, cette algorithme tend à faire contraire. Concrètement, il s'agit d'aiguiller les nouvelles requêtes (i.e. nouvelles connexions) vers les serveurs les plus chargés tant que ceux-ci sont en mesure de les gérer convenablement (i.e. performances). Le but est de maintenir quelques serveurs sous-utilisés pour être en mesure d'absorber une montée en charge soudaine en sollicitant ces serveurs le temps d'initialiser éventuellement de nouvelles ressources (i.e. *Scale Out*). Si la charge est amenée à baisser, ces serveurs sous-utilisés ne vont plus recevoir de nouvelles connexions et vont être progressivement éteints.

Les résultats des expériences montrent qu'il peut être intéressant de changer d'algorithme de répartition de charge dynamiquement pour gérer le compromis énergie-QoS. Les auteurs montrent notamment que l'utilisation conjointe d'un algorithme de répartition de charge classique et de *Load Skewing*, respectivement pour les montées et les baisses de charge, offre le meilleur compromis énergie-disponibilité.

Islam et al [IKLL12] ont présenté un modèle prédictif pour améliorer le dimensionnement automatique des applications e-commerce qui évoluent dans un contexte hautement dynamique (i.e. charge de travail variable). Les auteurs s'appuient sur deux méthodes statistiques d'analyse des séries chronologiques à savoir le réseau neuronal (*neural network* [HN04]) et la régression linéaire (*linear regression* [Wei05]) qu'ils confrontent dans leurs expérimentations. Ces deux algorithmes de prédiction sont utilisés dans les système d'apprentissage automatique et leur précision dépend de la fenêtre (de données passées) considérée. Ainsi, les auteurs utilisent aussi la technique des fenêtres glissantes (*sliding window* [Die02]).

L'objectif de ce travail est de fournir une meilleure prévision de la capacité du système (i.e. en termes de CPU) en faisant varier la fenêtre de prédiction. Les auteurs s'intéressent au dimensionnement horizontal qui implique un temps d'initialisation des ressources non négligeable. Il s'agit alors de fixer une fenêtre de prédiction en adéquation avec le temps nécessaire pour passer à l'échelle (i.e. démarrage une VM) en vue d'absorber la durée de l'adaptation.

Le résultat des expérimentations montre que la méthode basée sur le réseau neuronal est plus efficace pour la prévision de l'utilisation des ressources que celle s'appuyant sur la régression linéaire. Les expérimentations amènent les auteurs à proposer un intervalle de 12 minutes comme fenêtre de prédiction afin d'absorber le temps nécessaire au démarrage d'une VM qui est autour de 5-15 minutes dans leurs évaluations. On peut noter que le modèle proposé par [IKLL12] n'intègre pas les préoccupations liées à la QoS, aux performances ou encore aux coûts d'infrastructure associés au dimensionnement.

Dans leur travail, Gong et al [GGW10] ont proposé le système *PRESS* (*PRedictive Elastic reSource Scaling*), qui vise à dimensionner de manière pro-active les ressources. *PRESS* s'intéresse au dimensionnement vertical et s'appuie sur l'analyse des séries chronologiques et plus précisément sur la méthode de transformation de Fourier rapide (i.e. *Fast Fourier Transform*). Cette méthode est utilisée pour identifier, à partir de la charge de travail entrante, des motifs dans l'évolution des ressources (i.e. CPU, RAM et réseau) au cours du temps. Ces motifs (ou *patterns*) font office de "signatures" représentatives d'une certaine demande.

Les auteurs vont ensuite confronter les observations effectuées à l'exécution du système (e.g. en termes de CPU et/ou de RAM) aux différents signatures pour prédire les besoins en termes de ressource. Le modèle de prédiction de ressource est mis à jour à l'exécution si les motifs de consommation de ressource sont amenés à changer. Dans le cas où aucune signature n'est détectée à l'exécution, *PRESS* emploie une approche statistique basée sur les états (i.e. *statistical state-driven approach*) pour identifier des motifs à plus court terme concernant la demande. Pour ce faire, Gong et al utilisent la méthode des *chaînes de Markov discrètes* qui vise à prédire un état futur en s'appuyant sur l'état présent et non sur l'état passé. Il s'agit d'ailleurs de la propriété caractéristique d'une chaîne de Markov qui indique que "la prédiction du futur à partir du présent n'est pas rendue plus précise par des éléments d'information supplémentaires concernant le passé, car toute l'information utile pour la prédiction du futur est contenue dans l'état présent du processus" (Andreï Markov).

PRESS (cf. Figure 3.7) met ainsi en œuvre deux types de prédiction, l'une à long terme et basée sur les données passées (i.e. signatures issues de la méthode de transformation de Fourier) et l'autre à court terme, qui s'intéresse à l'état courant des ressources pour en prédire l'état futur (i.e. chaînes de Markov). *PRESS* ne gère ni la surveillance, ni l'exécution des actions et s'intéresse uniquement à la décision de dimensionnement. Les observations et les adaptations sont respectivement fournies et exécutées par le système (i.e. IaaS). Bien que la solution tend à estimer le juste niveau de ressource en fonction de la demande, les auteurs indiquent que *PRESS* préfère le sur-dimensionnement au sous-dimensionnement en vue d'éviter les violations de SLA. En ce sens, les auteurs privilégient les performances au détriment des coûts.

Dans leurs expérimentations réalisées sur la plateforme Xen [xen15b] et s'appuyant sur des traces réelles d'un *cluster Google*, les auteurs comparent *PRESS* à d'autres approches à savoir l'auto-régression, la prédiction basée sur les histogrammes ou encore l'auto-corrélation, une technique utilisée en mathématique et en traitement du signal qui permet de détecter des régularités ou des profils répétés dans un signal. Les expérimentations montrent que *PRESS* offre de meilleurs résultats en termes de prédiction et d'adaptation (i.e. consommation de CPU, violation de SLO sur les temps de réponse).

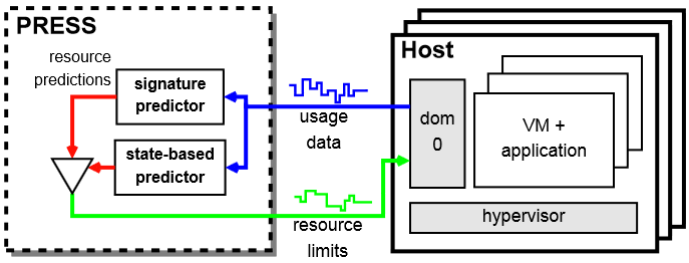


FIGURE 3.7 – Architecture générale de *PRESS* [GGW10].

Fang et al [FLWC12] ont proposé le *framework RPPS* (*cloud Resource Prediction and Provisioning Scheme*) qui permet de provisionner des ressources IaaS de manière automatique en vue de satisfaire des SLAs. La solution suit une approche pro-active pour exécuter des actions de passage à l'échelle en amont en s'appuyant sur une prédiction de la demande future. Le *framework* prend en considération les différentes méthodes de passage à l'échelle à savoir le dimensionnement horizontal et vertical (i.e. CPU et RAM) auxquelles vient s'ajouter la migration pour la consolidation.

Dans leur travail, Fang et al [FLWC12] disent adresser trois problèmes qui sont la prédiction précise de la charge de travail, l'efficience de l'adaptation pour éviter le sur-dimensionnement et le sous-dimensionnement et enfin la migration de VM qui en fonction de la stratégie choisie va avoir un impact différent sur l'énergie, les coûts et les performances. La prédiction de la charge de travail repose sur l'analyse des séries chronologiques et comme [RDG11], les auteurs s'appuient sur le modèle *ARMA* (*AutoRegressive Moving Average*).

Les expérimentations réalisées sur *Xen* [xen15b], *KVM* [kvm15] et s'appuyant sur des traces réelles de centres de données indiquent que la prédiction de la charge de travail admet globalement une marge d'erreur de l'ordre de 10% (i.e. sur-estimation et sous-estimation) et montrent des décisions d'adaptation pertinentes prises par *RPPS*.

Dutta et al [DGVV12] proposent *SmartScale*, un *framework* pour le dimensionnement automatique suivant une approche pro-active et visant à combiner les méthodes de dimensionnement horizontale et verticale. Les auteurs partent du constat que le dimensionnement vertical a une portée limitée, notamment due aux ressources physiques de la machine physique hôte, mais que cette méthode admet des coûts moindre en termes de ressource (i.e. granularité plus fine) et d'adaptation (e.g. temps de reconfiguration). Le dimensionnement horizontal, quant à lui, a une portée plus importante (potentiellement infinie), mais généralement au détriment des coûts.

SmartScale vise à combiner les deux méthodes de dimensionnement pour bénéficier de leurs avantages respectifs. Ainsi, la solution tend à privilégier le passage à l'échelle vertical (i.e. *Scale Up* et *Scale Down*) dans le cas de fluctuations légères de la charge et réserver le passage à l'échelle horizontal (i.e. *Scale Out* et *Scale In*) pour les variations importantes de la demande. Le but de la solution est de bénéficier de cette synergie en vue de réduire le coût total induit par la quantité de ressources nécessaire au fonctionnement de l'application.

L'objectif revient à trouver une configuration qui va minimiser le coût total du système. Celui-ci est calculé comme étant la somme des coûts des ressources (*hardware cost*), de la gestion de celles-ci (*labour cost*), des licences induites (*license cost*) mais aussi du coût de la reconfiguration (*reconfiguration cost*) et enfin des pénalités éventuelles en cas de manquement aux SLAs signés (i.e. temps de réponse).

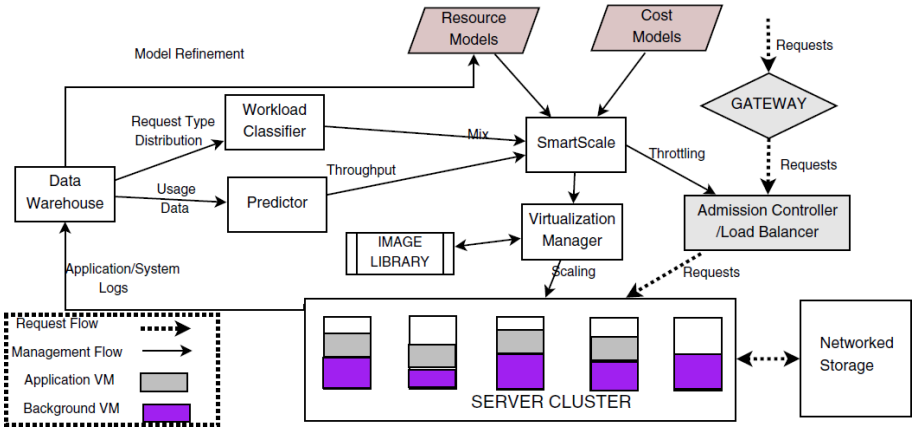


FIGURE 3.8 – Architecture de *SmartScale* proposée par [DGVV12].

SmartScale (cf. Figure 3.8) propose un composant, appelé *Predictor*, qui prend en entrée l'historique des différentes requêtes passées et qui repose sur l'analyse des séries chronologiques, et plus précisément la méthode appelée approximation polynomiale (*polynomial approximation*), pour prédire le nombre de requêtes (de chaque type) sur le prochain intervalle. De plus, la solution intègre un composant, nommé *Workload Classifier*, visant à classifier la charge de travail selon certains profils établis au préalable.

Les expérimentations confrontent *SmartScale* à deux autres stratégies de dimensionnement automatique. La première stratégie, appelée *Vertical Scaling*, revient, comme son nom l'indique, à passer à l'échelle de manière verticale les VMs existantes jusqu'à ce que la PM hôte soit saturée en ressources (i.e. CPU ou RAM). Une fois que le maximum de ressources est atteint sur toutes les PMs, de nouvelles instances sont créées, puis on refait appel au dimensionnement vertical, etc. La seconde stratégie, nommée *Fixed-Size*, va quant à elle ajouter des instances de taille fixe (i.e. toujours la même offre) dès que les VMs existantes sont saturées en termes de RAM ou de CPU. Les résultats des différentes expérimentations montrent que l'approche hybride (i.e. horizontal et vertical) adoptée par *SmartScale* associée à l'approche pro-active mise en place (*Predictor*) permet de réduire considérablement les coûts induits par le dimensionnement automatique.

Autres travaux respectant une approche pro-active

Outre les théories classiques utilisées pour le dimensionnement automatique (cf. sous-section 2.3.2), certains travaux s'appuient sur d'autres techniques. Sharma et al [SSSS11] proposent notamment *Kingfisher*, une solution pro-active de dimensionnement horizontal couplée aux mécanismes de migration dans le but de réduire les coûts d'infrastructure ainsi que le temps de reconfiguration (i.e. *transition cost*). La prédiction de *Kingficher* repose sur le profilage empirique (*empirical profiling*) et vise à passer au banc d'essai différentes configurations en faisant augmenter graduellement la demande pour constater à partir de quel moment la configuration sature. Il s'agit ensuite d'associer à différents profils de demande des configurations "viables" mais surtout d'éliminer les configurations dont on sait à l'avance qu'elles ne permettront pas de satisfaire la charge de travail entrante. La solution de prise de décision de Sharma et al repose sur l'optimisation linéaire en nombres entiers qui adresse des problèmes d'optimisation d'une forme particulière. Ces problèmes sont décrits par une fonction de coût ainsi que des contraintes linéaires et des variables entières. Il s'agit de sélectionner la configuration la moins couteuse permettant de satisfaire la demande et les contraintes (i.e. SLA). Les expérimentations menées montrent des résultats intéressants en termes de réduction des coûts d'infrastructure et de reconfiguration. De plus, les résultats mettent en lumière l'intérêt d'utiliser conjointement le dimensionnement horizontal et les mécanismes de migration.

3.2.2 Solutions réactives

Nous étudions ici des solutions de la littérature admettant une approche de dimensionnement automatique purement réactive, c'est-à-dire réagissant aux changements d'états du système.

Règles à base de seuils

Les publications étudiées ci-dessous mettent en place un dimensionnement automatique reposant sur les règles à base de seuils (cf. sous-section 2.3.3). Pour rappel, il s'agit de la technique majoritairement adoptée par les solutions reposant sur une approche réactive.

Han et al [HGGG12] ont proposé une approche réactive combinant l'élasticité horizontale et verticale en vue d'optimiser l'utilisation des ressources dans le contexte d'applications respectant une architecture multi-tier. La solution repose sur des algorithmes de prise de décision qui mixent à la fois le dimensionnement horizontal, qualifié de gros grain, et le dimensionnement vertical (i.e. CPU, RAM et I/O) décrit comme un passage à l'échelle à granularité fine. Les auteurs s'intéressent à l'impact des actions de dimensionnement en termes de QoS en intégrant un SLO sur les temps de réponse. L'algorithme de dimensionnement général proposé par Han et al, appelé algorithme de dimensionnement léger (*Lightweight Scaling algorithm - LS*), est composé d'une action d'observation (*monitoring*), effectuée à intervalles réguliers, puis de la transcription algorithmique de deux règles à base de seuils définissant respectivement si l'application nécessite un passage à l'échelle vers le haut ou vers le bas.

Il s'agit ici de confronter les valeurs de l'observation en termes de temps de réponse à un intervalle de temps de réponse acceptables (basé sur le SLO et utilisé dans les différents algorithmes pour définir si oui ou non l'adaptation effectuée est suffisante). En cas de condition évaluée à vrai, les règles vont respectivement appelées les algorithmes *LSU* (*Lightweight Scaling Up*) et *LSD* (*Lightweight Scaling Down*).

L'algorithme *LSU* fait dans un premier temps appel à l'algorithme appelé *Self-Healing* (auto-guérison) avant d'effectuer des actions de redimensionnement. Cet algorithme s'intéresse aux ressources physiques sous-jacentes (i.e. PMs). Le fonctionnement est simple, si deux VMs ayant la même PM hôte nécessitent une adaptation verticale contraire pour la même ressource (e.g. RAM), on va pouvoir retirer les ressources de la VM surdimensionnée (e.g. 2048 MB de mémoire) pour les attribuer à la VM sous-dimensionnée en faisant appel aux algorithmes de dimensionnement vertical associés (*Resource-level scaling down* et *Resource-level scaling up*). Si un tel couple de VMs n'existe pas ou si l'adaptation effectuée s'avère insuffisante, *LSU* fait appel de manière itérative aux algorithmes de dimensionnement horizontal (*VM-level scaling up*) et/ou vertical (*Resource-level scaling up*).

L'algorithme *LSD* est plus simple et fait intervenir en premier lieu le dimensionnement horizontal (*VM-level scaling down*) avant le dimensionnement vertical (*Resource-level scaling down*). Les deux algorithmes *LSU* et *LSD* induisent une certaine préférence d'élasticité de la part des auteurs. Dans le cadre d'un passage à l'échelle vers le haut (*LSU*), [HGGG12] privilégie le dimensionnement vertical (i.e. à granularité fine) au passage à l'échelle horizontal (i.e. gros grain) tandis qu'ils vont privilégier le retrait de VM dans le cas d'un passage à l'échelle vers le bas (*LSD*) du fait des gains supérieurs en termes de coût. Les résultats des expérimentations mettent en lumière l'intérêt d'utiliser conjointement l'élasticité verticale et horizontale. Le fait d'avoir recours à des actions de dimensionnement à granularité variable dans un contexte dynamique permet notamment de minimiser les coûts tout en respectant les engagements de niveau de service.

Maurer et al [MBS11] ont présenté une solution réactive de dimensionnement automatique basée sur les règles à base de seuils. Les auteurs s'intéressent essentiellement à l'élasticité verticale à laquelle peut venir s'ajouter l'élasticité horizontale, la migration ou encore le fait d'externaliser les ressources (i.e. *outsourcing*). Il s'agit concrètement de surveiller l'évolution des ressources des VMs et d'en ajuster la quantité. Les ressources considérées dans ce travail sont le CPU, la RAM, le stockage et la bande passante.

La proposition des auteurs repose sur une approche définissant différents seuils de menace (i.e. du vert au rouge) dont le but est de segmenter les problèmes de ressources des VMs (i.e. stockage, RAM, CPU et réseau) ainsi que cinq niveaux possibles d'adaptation constituant une échelle (i.e. *escalation levels*). Chaque niveau définit un type d'action de reconfiguration : *1)* changer la configuration de la VM (i.e. élasticité verticale), *2)* migrer l'application d'une VM à une autre (i.e. déploiement), *3)* migrer une VM depuis sa PM hôte à une PM cible (i.e. migration) ou créer une nouvelle VM sur la PM appropriée (i.e. élasticité horizontale), *4)* allumer/éteindre une PM et enfin *5)* externaliser ses ressources chez un autre fournisseur IaaS (i.e. *outsourcing*, *cloud bursting*).

Les différents niveaux admettent des actions à granularité variable nécessitant des efforts de reconfiguration plus ou moins importants. Le but est de résoudre les problèmes de ressource en faisant en priorité appel aux niveaux les plus hauts de la liste ci-dessus. Si le problème ne peut être résolu à un certain niveau de l'échelle, on va tenter de le résoudre en faisant appel au niveau suivant. Il s'agit de préférer les actions nécessitant un effort de reconfiguration moindre en considérant graduellement les choix d'adaptation possibles dépendamment des seuils de menace.

Théorie des files d'attente

Han et al [HGG+14] ont modélisé une application multi-tier comme un réseau de files d'attente où chaque tier est représenté par une queue. La solution met en place une méthode de dimensionnement horizontale pour le passage à l'échelle. Le modèle de file d'attente proposé est utilisé pour estimer le nombre de VM devant être ajouté ou retiré aux tiers qui en ont besoin (i.e. goulots d'étranglement) tout en considérant les coûts associés (i.e. budget maximum pour le déploiement de l'application). Il faut noter que les auteurs s'appuient sur un modèle de facturation des instances à la minute contrairement au modèle classique de facturation à l'heure. Ils proposent un algorithme de dimensionnement automatique permettant de réagir à un changement de la demande en ajustant la quantité de ressource des tiers et en intégrant de manière inhérente le critère de coût (*Cost-Aware Scaling - CAS algorithm*). Han et al donnent aussi les spécifications d'un modèle (XML) extensible permettant de décrire les propriétés des PMs, la configuration des VMs et des tiers ou encore les paramètres ou contraintes propres aux utilisateurs. En s'appuyant sur cette spécification et l'algorithme *CAS*, ils proposent un *middleware* (*iSSe* - cf. Figure 3.9) faisant la glu entre les fournisseurs d'infrastructure et les gestionnaires d'applications (i.e. fournisseur SaaS). Bien que les auteurs n'utilisent pas le terme *PaaS*, il s'agit bien d'un service de dimensionnement automatique de type PaaS.

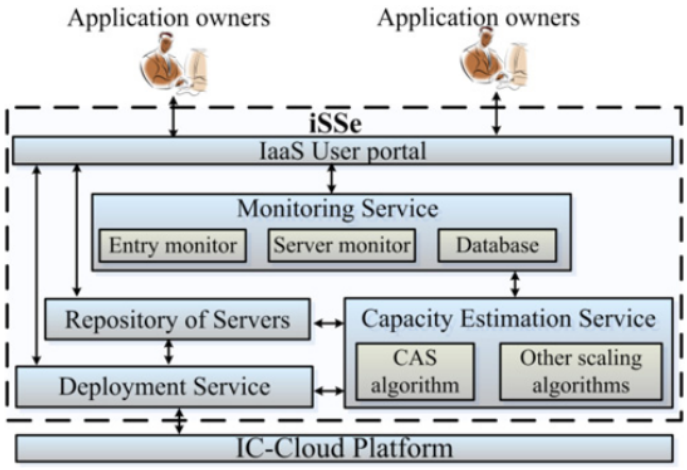


FIGURE 3.9 – Architecture de la plateforme *iSSe* proposée par [HGG+14].

Théorie du contrôle

Lim et al [LBCP09] ont proposé une solution de dimensionnement automatique réactive pour l'élasticité horizontale. Contrairement aux solutions réactives "classiques" qui reposent sur des règles simples, portant généralement sur une métrique que l'on va confronter à une unique valeur seuil statique, leur proposition se distingue du fait que la métrique est confrontée à des intervalles cibles dynamiques. Le contrôle est basé sur un régulateur proportionnel intégral (*proportional integral controller*) et plus précisément sur la technique de seuillage proportionnel (*proportional thresholding*) qui est adaptée aux systèmes avec des actions de dimensionnement (i.e. *actuators*) gros grains comme c'est le cas du passage à l'échelle horizontal. Le but est de réagir uniquement quand la valeur observée d'une métrique (i.e. consommation moyenne de CPU) est en dehors de l'intervalle cible en vue de réduire les adaptations "inutiles" pouvant être appliquées par une politique d'adaptation trop agressive (i.e. seuils statiques). L'objectif de Lim et al [LBCP09] est d'éviter les oscillations observées par le système en termes de dimensionnement c'est-à-dire l'ajout puis le retrait rapide et perpétuel de ressources (i.e. *Scale Out - Scale In*) qui s'avèrent néfastes du fait que la reconfiguration elle-même se révèle couteuse. Les expérimentations réalisées sur *Amazon EC2* [EC215b] montrent que l'approche utilisant la technique de seuillage proportionnel admet de meilleurs résultats que l'approche usuelle mettant en œuvre des seuils statiques.

Autres travaux respectant une approche réactive

Fitó et al ont présenté *Cloud Hosting Provider* (*CHP* [FGG10]), un nouveau type de fournisseur Cloud, proposant des services d'hébergement s'adressant aux fournisseurs d'application web (SaaS). *CHP* fait appel aux techniques d'externalisation des ressources IaaS (*outsourcing*) pour adresser les préoccupations de passage à l'échelle et de haute disponibilité (*availability*) des applications.

[FGG10] tend à combiner l'utilisation d'un Cloud privé (i.e. celui du *CHP*) avec un ou plusieurs Cloud public. Lorsque la capacité de ressources du Cloud privé n'est pas suffisante pour satisfaire la demande des applications hébergées, le *CHP* va faire appel à des ressources de Cloud public permettant ainsi d'étendre la capacité de ressources en vue de satisfaire les SLAs signés avec ses clients. On retrouve le terme *Cloud Bursting* (i.e. débordement) dans la littérature pour définir le fait d'avoir recours à des ressources provenant de Cloud public en cas de ressources limitées d'un Cloud privé.

Outre le fait de répondre aux besoins des applications en fonction de la demande, un des objectifs de ce travail est de maximiser le profit de ce nouveau type de fournisseur (*CHP*). La solution proposée par Fitó et al repose sur l'analyse des SLAs signés avec les fournisseurs d'application (i.e. temps de réponse) ainsi qu'un modèle économique.

Mao et al [MH13] ont présenté une solution de dimensionnement automatique suivant une approche réactive visant à maximiser les performances des applications tout en respectant des contraintes budgétaires en termes de dollars par heure. Ils s'intéressent à la fois à l'élasticité horizontale et verticale et prennent en considération le temps de reconfiguration avec notamment le temps d'initialisation des ressources induit par l'élasticité horizontale.

L'approche de dimensionnement automatique repose sur deux algorithmes distincts. Le premier algorithme, nommé *scheduling-first*, va prioriser l'ordonnancement à l'allocation. Cet algorithme distribue le budget global aux différentes tâches (*jobs*) en fonction de leur priorité puis propose un plan d'exécution (i.e. ordonnancement) le plus rapide possible avant d'acquérir les ressources (i.e. allocation). Le second algorithme, appelé *scaling-first*, va quant à lui déterminer le nombre et le type d'instance (i.e. *small, medium et large*) nécessaire dans la limite du budget fixé (i.e. chaque type d'instance a un coût par heure). Une fois les VMs acquises, l'algorithme ordonnance l'exécution des tâches en fonction de leur priorité en tentant de réduire la durée d'exécution de celles-ci.

Les résultats des expériences indiquent que l'algorithme *scaling-first* montre de meilleurs performances dans le cas d'un budget modeste (i.e. 5-15$/heure) tandis que l'algorithme *scheduling-first* est plus intéressant dans le cas de budget plus important (i.e. 15-30$/heure).

3.2.3 Solutions hybrides

Nous étudions ici différentes solutions hybrides de la littérature. Il s'agit de travaux intégrants à la fois une approche réactive et une approche pro-active.

Règles à base de seuils

Moore et al ont proposé un *framework* pour la gestion de l'élasticité dans le cadre d'application multi-tier : *Platform Insights* [LRME13]. Ce *framework*, pouvant être apparenté à un service PaaS pour la gestion de l'élasticité, considère uniquement le dimensionnement horizontal. La solution admet une approche hybride reposant sur deux contrôleurs. Le premier contrôleur est réactif et repose sur les règles à base de seuils évaluées à intervalles réguliers. Le second contrôleur, quant à lui, suit une approche prédictive. Il s'appuie sur l'analyse des séries chronologiques et sur le modèle probabiliste de classification naïve bayésienne (*Naive Bayes model*) pour prédire la charge de travail (i.e. quantité et tendance) à court/moyen terme et la QoS associée. Il ne nécessite pas d'apprentissage préalable (i.e. apprentissage incrémental à l'exécution).

Les deux contrôleurs sont exécutés de manière indépendante. Néanmoins, les décisions d'adaptation (*scaling requests*) des contrôleurs sont envoyées à un composant (*Decision Manager*) responsable de la synchronisation et de l'exécution des plans de reconfiguration. Contrairement au contrôleur réactif dont les règles sont évaluées à intervalles réguliers, le contrôleur prédictif peut transmettre une demande d'ajout/retrait de ressource (i.e. VM) dès qu'il évalue cela nécessaire. La priorité est donnée au contrôleur prédictif, néanmoins, si celui-ci ne parvient pas à identifier une décision d'adaptation, le *Decision Manager* donne la main au contrôleur réactif tandis que le contrôleur prédictif continue à apprendre en mettant à jour ses modèles prédictifs. De même, la priorité est donnée au contrôleur prédictif en cas de décisions d'adaptation conflictuelles.

Les simulations effectuées, s'appuyant sur des traces de charge de travail réelles [AJ98], indiquent que le dimensionnement hybride de *Platform Insights* montre de meilleurs résultats qu'une approche purement réactive (cf. Figure 3.10).

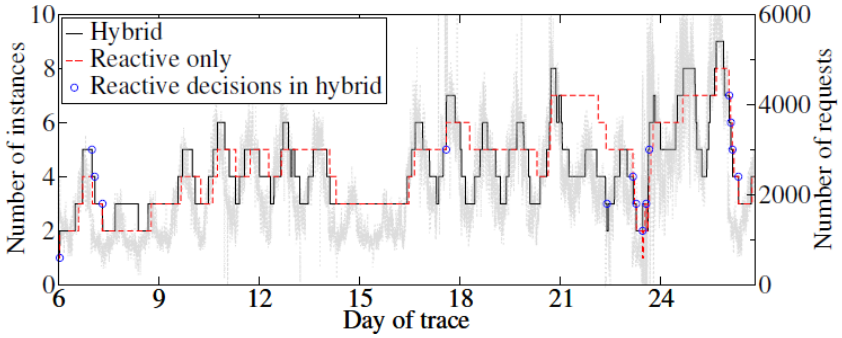


FIGURE 3.10 – Approche hybride *Platform Insights* vs approche purement réactive [LRME13].

Théorie des files d'attente

Urgaonkar et al [USC+08] s'intéressent au dimensionnement automatique de ressource pour les applications Web respectant une architecture multi-tier. Ils partent du constat que les applications multi-tier admettent des problématiques de dimensionnement qui diffèrent des applications mono-tier. En effet, le fait d'identifier et de reconfigurer le tier faisant office de goulot d'étranglement en termes de ressource peut simplement revenir à déplacer ce goulot à un autre tier et ainsi ne pas résoudre le problème de dimensionnement qui doit être pensé de manière transverse.

Les auteurs modélisent une application multi-tier comme un réseau de files d'attente (i.e. *G/G/1 queue*) où ils font correspondre à chaque serveur de l'application une file d'attente. Ainsi, un tier est constitué de plusieurs files d'attente dont la sortie est reliée au tier suivant.

Leur proposition permet de provisionner dynamiquement des ressources de manière horizontale en cas de pics de charge. Ce travail prend en considération les performances par le biais d'un SLA portant sur le temps de réponse moyen des requêtes de l'application. La solution repose sur une approche hybride combinant deux méthodes (i.e. prédictive et réactive) qui fonctionnent sur deux échelles de temps différents. La méthode prédictive attribue la capacité à une échelle de quelques heures ou jours, alors que la méthode réactive réagit de l'ordre de la minute.

La prédiction est basée sur les histogrammes et permet de prévoir la future demande (i.e. quelques heures) en se basant sur l'observation des charges de travail passées (cf. Figure 3.11). La méthode réactive est utilisée pour corriger les éventuelles erreurs de prédiction commises par la méthode prédictive (i.e. long terme) ou pour réagir à des événements imprédictibles ou des pics de charge inattendus.

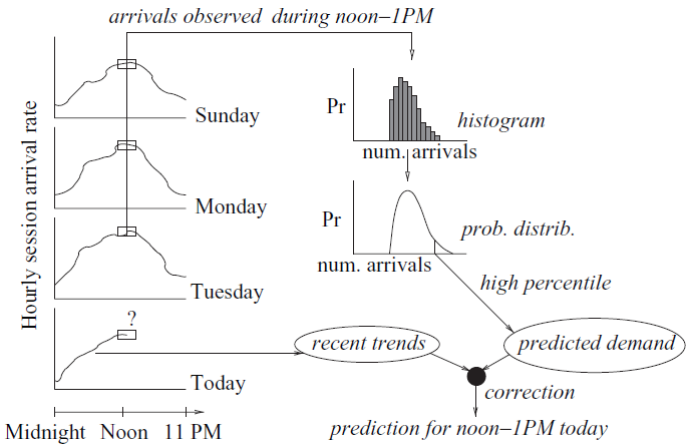


FIGURE 3.11 – Algorithme de prédiction de charge proposé par [USC+08].

Ali-Eldin et al [AETE12] adressent le problème du dimensionnement dynamique et autonomique pour une infrastructure Cloud en s'appuyant sur la théorie des files d'attente et la théorie du contrôle. Ils proposent une approche hybride pour adapter la quantité de ressource en vue de respecter des contrats de niveau de service en termes de disponibilité (*availability*). Ce travail s'intéresse à la méthode de dimensionnement horizontale et prend en considération la préoccupation de temps d'initialisation des ressources.

Les auteurs dressent dans un premier temps les différentes manières de construire un système de gestion de l'élasticité (i.e. horizontale) qualifié de hybride et constitué d'un contrôleur pour le passage à l'échelle vers le haut et d'un contrôleur pour le passage à l'échelle vers le bas. Chacun de ces contrôleurs peut être réactif, pro-actif ou les deux (i.e. hybride). Ainsi, de manière exhaustive, ils distinguent neuf implémentations possibles qu'ils répertorient dans un tableau (cf. Figure 3.12). Il faut noter que les auteurs font référence aux termes *Scale Up* et *Scale Down* dans leur travail mais qu'il s'agit en réalité des actions *Scale Out* et *Scale In* (i.e. dimensionnement horizontal).

En s'appuyant sur la modélisation d'une infrastructure Cloud basée sur la théorie des files d'attente (queue stable de type *G/G/N*), les auteurs ont développé 2 contrôleurs pro-actifs qui estiment la charge future du service en s'appuyant sur l'historique de la charge passée. Ils passent ensuite au banc d'essai les deux contrôleurs pro-actifs développés pour le retrait de VM en les associant à un contrôleur suivant une approche réactive responsable de l'ajout de VM. Dans le cas de décision contradictoire entre deux contrôleurs, la priorité est donnée au contrôleur réactif. Les résultats (i.e. simulation s'appuyant sur les traces de la coupe du monde *FIFA 1998* [AJ98]) montrent que le fait d'utiliser une approche hybride (i.e. réactive pour le *Scale Out* et pro-active pour le *Scale In*) permet de diviser entre 2 et 10 le nombre de violation de SLA par rapport à une approche purement réactive.

Engine Name	Scale up mechanism	Scale down mechanism
UR-DR	Reactive	Reactive
UR-DP	Reactive	Proactive
UR-DRP	Reactive	Reactive and Proactive
URP-DRP	Reactive and Proactive	Reactive and Proactive
URP-DR	Reactive and Proactive	Reactive
URP-DP	Reactive and Proactive	Proactive
UP-DP	Proactive	Proactive
UP-DR	Proactive	Reactive
UP-DRP	Proactive	Reactive and Proactive

FIGURE 3.12 – Implémentations possibles d'un contrôleur hybride d'élasticité selon [AETE12].

Analyse des séries chronologiques

Iqbal et al [IDCJ11] ont proposé un prototype de dimensionnement horizontal suivant une approche hybride. Comme [AETE12], ce travail met en œuvre une approche réactive pour le dimensionnement vers le haut (i.e. ajout de VM) et une approche pro-active pour le dimensionnement vers le bas (i.e. retrait de VM). L'approche réactive est mise en place par des règles portant sur l'utilisation de CPU tandis que l'approche pro-active utilise une méthode de séries chronologiques et plus précisément la régression polynomiale. L'objectif est de satisfaire des temps de réponse corrects (i.e. respect d'un SLO sur les temps de réponse) tout en minimisant l'utilisation des ressources. Le prototype, reposant sur une architecture multi-tier, vise à ajuster le nombre de VM du tier métier et du tier données. Après un nombre fixe d'intervalles pour lesquels le temps de réponse est satisfait, [IDCJ11] calcule le nombre d'instances nécessaire pour les tiers métier et base de données en utilisant la régression polynomiale de degré deux. Les auteurs considèrent une base de données en lecture seule (i.e. cas simple), qui contrairement aux bases de données en lecture-écriture, permet d'éviter le besoin de synchronisation des données en cas de duplication du tier correspondant (i.e. tâche longue et critique).

Shen et al ont proposé *CloudScale* [SSGW11], un système dérivé de *PRESS* [GGW10] qui s'attaque davantage aux préoccupations énergétique en offrant des mécanismes permettant de réduire la consommation énergétique. La solution s'appuient notamment sur la migration d'instance ainsi que la capacité des processeurs modernes à opérer selon plusieurs fréquences CPU ou encore différents niveaux de voltage (i.e. *Dynamic Voltage and Frequency Scaling - DVFS* [VLWYH09]). *CloudScale* (cf. Figure 3.13) va dans un premier temps prédire la charge de travail et les besoins futurs du système en termes de ressource en s'appuyant sur le modèle de prédiction de *PRESS* [GGW10] décrit précédemment. Ensuite, la solution va être en mesure de changer dynamiquement la capacité des VMs (i.e. dimensionnement vertical) mais aussi la fréquence CPU des processeurs ou encore d'augmenter (*overvolting*) ou de diminuer (*undervolting*) leur niveau de voltage en vue de rendre le système plus ou moins consommateur d'énergie tout en respectant les SLOs signés. Par exemple, si la prédiction indique que seulement 50% du CPU d'une machine hôte (i.e. PM) est nécessaire, il est possible de diviser de moitié la fréquence CPU de celle-ci tout en doublant la capacité des VMs associées ce qui va permettre de réduire la consommation énergétique (i.e. CPU fonctionne plus lentement) sans impacter les SLOs. La solution repose ainsi sur l'utilisation conjointe de *DVFS* et du dimensionnement vertical (avec migration). Il est possible de considérer *DVFS* comme un dimensionnement vertical au niveau des PMs (i.e. changer l'"offre" du processeur). Les auteurs, dans leurs expérimentations, comparent l'approche avec et sans *DVFS* ainsi que l'impact de la migration en termes de consommation de ressource (i.e. CPU alloué), d'énergie ou encore de performance (i.e. SLO sur les temps de réponse). Les résultats montrent que *CloudScale* permet de réaliser des économies énergétiques (et donc financières) tout en limitant les violations de SLO sans sur-coût important induit par la solution.

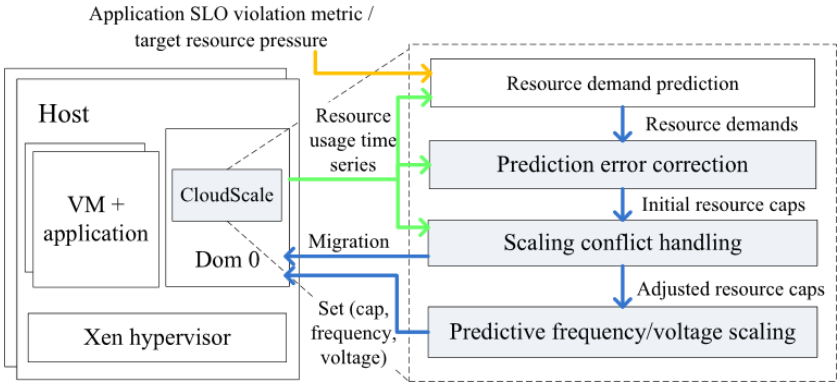


FIGURE 3.13 – Architecture générale de *CloudScale* [SSGW11].

3.3 Élasticité étendue aux autres couches

Nous discutons ici des travaux s'intéressant à l'élasticité et plus généralement à l'adaptation des ressources Cloud. Néanmoins, contrairement à la section précédente, les travaux de cette section ne se focalisent pas uniquement sur la couche IaaS. Ils considèrent différents types de ressource que l'on peut associer aux couches PaaS ou SaaS. Ainsi, nous avons regroupé ces travaux en deux catégories. La première catégorie comprend les travaux portant sur l'adaptation architecturale du logiciel et la paramétrisation des ressources logicielles (i.e. SaaS). La seconde catégorie, quant à elle, regroupe les travaux traitant l'adaptation du Cloud de manière transverse c'est-à-dire avec des actions portant sur plusieurs couches (e.g. IaaS-SaaS, PaaS-SaaS, etc.).

3.3.1 Adaptation du logiciel

Reconfiguration architecturale de l'application

Garlan et al se sont intéressés à l'adaptation autonomique des systèmes informatiques au début des années 2000 avant l'air du Cloud et la popularisation de la virtualisation. Ils ont proposé le *framework Rainbow* [GCH+04] qui s'appuie à la fois sur l'architecture des applications et une infrastructure qualifiée de réutilisable (*reusable*) par les auteurs pour offrir la capacité d'adaptation autonomique aux systèmes informatiques (i.e. logiciels au sens large). *Rainbow* permet de surveiller l'état et l'environnement d'exécution du système dont on attend un comportement autonomique. En confrontant ses observations aux modèles architecturaux du logiciel dont il a connaissance, le *framework* est en mesure de détecter les opportunités d'amélioration mais aussi de prévoir et d'exécuter un plan de reconfiguration permettant de mener le système dans un état préférable. Pour cela, *Rainbow* repose sur une modèle conceptuel proche de la boucle autonomique *MAPE-K* [KC03] dont la base de connaissance (*Knowledge* - K) est constituée d'un modèle architectural explicite du logiciel ainsi qu'un répertoire de stratégies (i.e. actions à entreprendre pour sortir d'un état indésirable), de contraintes et de préférences d'adaptation (cf. Figure 3.14). Dans leurs travaux, Garlan et al adressent les mêmes préoccupations que celles de l'élasticité du Cloud c'est-à-dire l'adaptation et le dimensionnement automatique des ressources en fonction de la demande. Ils mettent en œuvre différents types d'adaptation portant sur l'architecture du logiciel et sur l'infrastructure (i.e. serveur physique, pas de ressources virtuelles au sens IaaS). Ils présentent notamment un cas d'utilisation d'une application web de type client-serveur gérée par *Rainbow* qui dépendamment de la QoS observée (i.e. temps de réponse) ou de la quantité de ressource disponible (i.e. bande passante), va ajuster la quantité de ressource en ajoutant par exemple un serveur (i.e. dimensionnement horizontal physique) ou encore rediriger certains clients sur un autre serveur (i.e. adaptation architecturale).

Ahmad et al [AB14] ont présentés un langage permettant de définir des patrons d'adaptation réutilisables pour les applications Cloud. Il s'agit de proposer des solutions clé en main de reconfiguration architecturale face à des problèmes récurrents. Les auteurs motivent leur travail autour des patrons pour l'auto-adaptation (*self-adaptation*) par le fait que les applications Cloud sont confrontées à un environnement dynamique et sont généralement exposées aux mêmes problèmes (i.e. besoin d'élasticité). Ahmad et al proposent un méta-modèle pour la définition et la composition de patron d'adaptation. Un patron est constitué d'un nom, d'un objectif, d'un plan d'adaptation ainsi que de contraintes de type invariant, pré-condition et post-condition. La Figure 3.15 illustre la définition d'un patron d'adaptation nommé *Service Mediation* permettant de répondre au problème récurrent d'ajout d'un service (i.e. service de chiffrage) entre deux services directement connectés. Leur approche est empirique dans le sens où les situations nécessitant une reconfiguration ainsi que les patrons d'adaptation sont respectivement découvertes et définis de manière incrémentale à l'exécution en analysant les précédentes reconfiguration (i.e. *logs* d'adaptation).

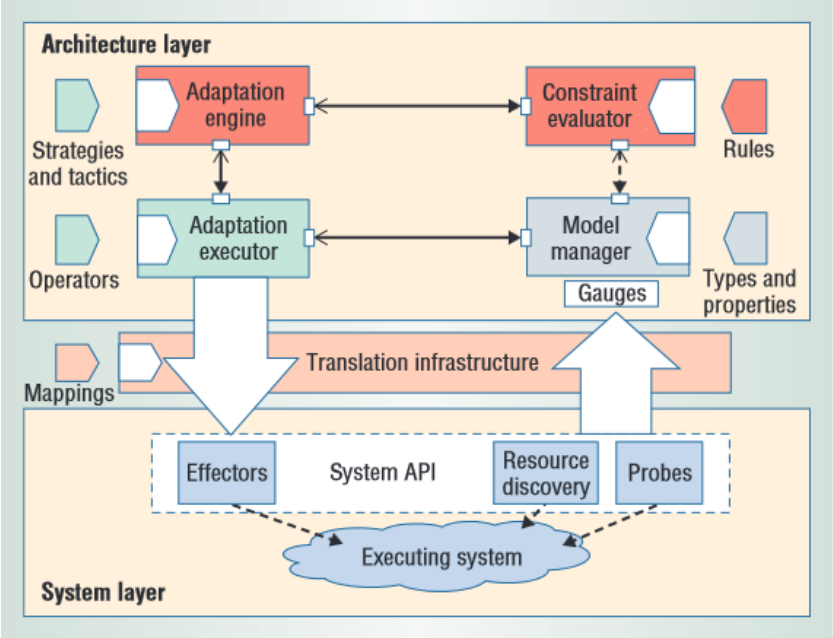


FIGURE 3.14 – Boucle de contrôle du *framework Rainbow* proposé par [GCH+04].

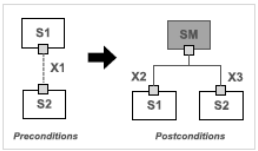
Pattern Name and Intent
Service Mediation ([S_M] <S_1, S_M, S_2>)
Intent – To *interpose* a mediator service component (S_M) among two or more directly connected components (S_1, S_2)
Pattern Constraints, Operators and Impact
Constraints – conditions before, during and after the change. – *Preconditions* S_1 and S_2 must be directly connected. – *Invariants* S_1 and S_2 must be disconnected. – *Post-conditions* S_1 and S_2 must connected with S_M.
Change Operators – to apply architecture restructuring. – Add a Component S_M – Remove a Connector X_1 (S_1, S_2) (more connectors may exist, see variant) – Add a Connector X_2 (S_1, S_M) – Add a Connector X_3 (S_M, S_2)
Impact on Architecture Models – the affected architecture model. 

FIGURE 3.15 – Spécification du patron *Service Mediation* proposée par [AB14].

Paramétrisation du Logiciel

Klein et al ont proposé *Brownout* [KMÅHR14], qu'ils définissent eux-mêmes comme un nouveau paradigme de programmation pour des applications auto-adaptables. Contrairement aux applications Cloud qui tendent à ajuster la quantité de ressource en vue de s'adapter au contexte dynamique (e.g. charge de travail variable) et aux événements imprédictibles bien que récurrents (e.g. pannes), il s'agit ici de reconfigurer l'application elle-même. Concrètement, la solution repose sur des morceaux de code ou des composants logiciels pouvant être (dés)activés dynamiquement en fonction du contexte d'exécution évitant ainsi le recours usuel au sur-dimensionnement de ressource.

Les auteurs estiment qu'il est préférable, dans certains cas, de dégrader l'expérience utilisateur plutôt que de voir la QoS se dégrader par manque de ressources (e.g. temps de réponse, disponibilité, etc.). L'approche consiste à désactiver certains composants (e.g. fonctionnalités optionnelles) en vue d'augmenter la capacité et ainsi accepter davantage de requêtes ce qui offre la possibilité d'essuyer des montées en charge importantes sans nécessairement avoir recours au dimensionnement des ressources IaaS pouvant s'avérer couteux voir impossible (e.g. contraintes budgétaires, maximum de ressource atteint, etc.). *Brownout* repose sur la théorie du contrôle et vise à contrôler automatiquement les besoins de capacité de l'application au détriment de l'expérience qualitative du service rendu à l'utilisateur final. Contrairement au dimensionnement automatique IaaS adressant les compromis incluant la quantité de ressources et les coûts, il s'agit ici d'adresser un nouveau genre de compromis incluant l'expérience utilisateur.

Les expérimentations de Klein et al s'appuient sur deux prototypes populaires d'applications web inspirés de *eBay.com* [eba15] : *RUBiS* [rub15b] et *RUBBoS* [rub15a]. L'application web considérée peut être qualifiée de site E-commerce standard à l'exception qu'elle offre une fonctionnalité de recommandation optionnelle. Cette fonctionnalité permet de suggérer à l'utilisateur certains produits en fonction de son panier actuel, de ses achats précédents ou encore des achats effectués par d'autres utilisateurs. Bien que cette fonctionnalité puisse s'avérer cruciale pour le fournisseur de service (e.g. *eBay*), ce n'est pas le cas pour les utilisateurs. De plus, les algorithmes nécessaires à sa mise en place sont souvent gourmands en termes de ressource et la désactivation d'une telle fonctionnalité peut permettre des gains considérables (i.e. soulager les ressources).

La solution proposée considère le fait que le périmètre fonctionnel des applications peut être étendu par un paramètre dynamique appelé *dimmer* qui va affecter l'expérience de navigation des utilisateurs ainsi que la capacité en ressource nécessaire pour rendre le service. Il s'agit d'offrir les fonctionnalités optionnelles quand la demande et les ressources le permettent. Dans le cas de leur prototype, la valeur du *dimmer* va régir le pourcentage de requête avec ou sans suggestion. Ainsi, plus la valeur du *dimmer* est importante, plus les recommandations seront fréquentes ce qui va améliorer l'expérience utilisateur tout en augmentant le besoin en ressource. Dans le cas de pic de charge important, le contrôleur va diminuer la valeur du *dimmer* pour soulager les ressources et ainsi continuer de fournir le service (i.e. dégradé) à un maximum d'utilisateur. Les résultats des expérimentations montrent que *Brownout*, en comparaison avec une approche standard (i.e. sans reconfiguration de l'application), permet de répondre à davantage d'utilisateurs ou de supporter la même demande avec moins de ressources tout en maximisant l'expérience utilisateur. Ce travail ne prend pas en compte l'élasticité de l'infrastructure mais les auteurs indiquent vouloir combiner leur approche avec le dimensionnement horizontal, vertical et la migration dans des travaux futurs.

Nikolov et al ont récemment présenté *CLOUDFARM* [NKHR14], une architecture PaaS pour la gestion des ressources SaaS, s'appuyant sur les SLOs définis au préalable. *CLOUDFARM* repose sur *OSGi* (*Open Services Gateway initiative* [osg15]) et combine certains éléments de *COSCA* [KDH11] et de *ARTOS* [NMR+]. Les auteurs s'intéressent à des applications offrants plusieurs modes (e.g. dégradé ou non dégradé) avec des complexités de calcul différentes et donc plus ou moins consommateurs de ressources (e.g. CPU, RAM, réseau, etc.). De même, chaque mode fournit un niveau de QoS différent.

Nikolov et al distinguent deux types de QoS et deux types de SLOs qui en résultent. La QoS dite fonctionnelle qui correspond à la qualité de rendu du service fourni à l'utilisateur (i.e. *quality level*). Un exemple de SLO portant sur ce type de QoS peut être "l'application doit être 90% du temps en mode non dégradé par jour". Le second type de QoS, défini comme non-fonctionnel, est quant à lui plus conventionnel. Il porte sur les propriétés usuelles de QoS permettant de définir des SLOs "classiques" (e.g. temps de réponse < 1500ms ou disponibilité > 99%). Les auteurs proposent un algorithme s'appuyant sur les SLOs définis permettant d'absorber les pics de charge éphémères.

L'objectif de *CLOUDFARM* est de reconfigurer dynamiquement l'application en ajustant le mode de celle-ci en fonction du contexte d'exécution en vue d'optimiser l'utilisation des ressources IaaS sous-jacentes (i.e. maximiser l'utilisation des ressources et donc le profit) ou encore d'éviter les situations de surcharge en respectant les SLOs portant sur la QoS fonctionnelle et non-fonctionnelle. Les auteurs s'intéressent à un cas d'utilisation d'application de vidéo *streaming* admettant différents modes correspondants à plusieurs niveaux de qualité vidéo en termes de résolution (i.e. plus ou moins consommateurs de bande passante). Ils indiquent que le fait de dégrader l'application est une adaptation à granularité fine qui permet d'absorber un court pic de charge plutôt que d'entreprendre une action de dimensionnement des ressources IaaS plus couteuse en termes de temps et de budget. Cela peut aussi être effectué en amont d'une telle décision de reconfiguration. Néanmoins, [NKHR14] n'inclut pas le dimensionnement de ressources IaaS dans leur solution.

3.3.2 Adaptation multi-couche du Nuage

Nous nous penchons ici sur les travaux considérant l'adaptation de manière multi-couche. Nous regroupons ainsi les travaux selon les couches du Cloud adressées par les actions d'adaptation (i.e. IaaS, PaaS, SaaS).

Couches IaaS et PaaS

Les mêmes auteurs que *VCONF* [RBX+09] ont aussi proposé *URL* [XRB12] et *CoTuner* [BRX13]. *URL* (*Unified Reinforcement Learning*) est une solution basée sur l'apprentissage par renforcement pour l'auto-configuration des VMs (i.e. dimensionnement vertical) ainsi que les différents paramètres des composants d'applications multi-tier (e.g. *Tomcat*, *MySQL*, etc.). La solution permet de gérer la configuration de la couche IaaS (i.e. ressources virtuelles) et des composants pouvant être associés à la couche PaaS (*middleware*). *URL* distingue ainsi deux types d'agent basés sur l'apprentissage par renforcement (cf. Figure 3.16).

Le *VM-Agent* (déployé sur une VM) est en charge de surveiller en permanence les performances des VMs et de les reconfigurer dynamiquement (en ajustant le CPU, la RAM, etc.) à intervalle régulier. Après chaque reconfiguration d'une VM, l'agent reçoit des informations (i.e. *feed-back*) sur les performances de celle-ci, traduisant le score de l'adaptation entreprise (i.e. gain ou pénalité), lui permettant ainsi de mettre à jour ses politiques d'adaptation. Le *App-Agent* (lui aussi déployé sur une VM) s'intéresse quant à lui à la configuration des différents paramètres des composants (i.e. *appliances*) déployés sur les VMs. Les auteurs indiquent que les composants offrent de très nombreux paramètres pouvant être configurés automatiquement en vue de s'adapter à l'environnement dynamique et ainsi satisfaire des objectifs qui leurs sont propres (e.g. en termes de performances). Ainsi, à l'instar du *VM-Agent*, l'*App-Agent* va surveiller les performances des composants de l'application (e.g. *Tomcat* [tom15]) et ajuster automatiquement leurs configurations pour répondre aux objectifs définis dans des SLOs. Les deux agents communiquent entre eux en vue de prendre des décisions de reconfiguration "cohérentes". En effet, ils adressent des objectifs différents bien que corrélés. Le *VM-Agent* vise à respecter les objectifs en termes d'utilisation de ressources tandis que le *App-Agent* tend à respecter les objectifs de performance de l'application (définis dans des SLOs). La communication entre les deux agents permet d'adresser le compromis entre la quantité de ressources attribuées au système et les performances de l'application.

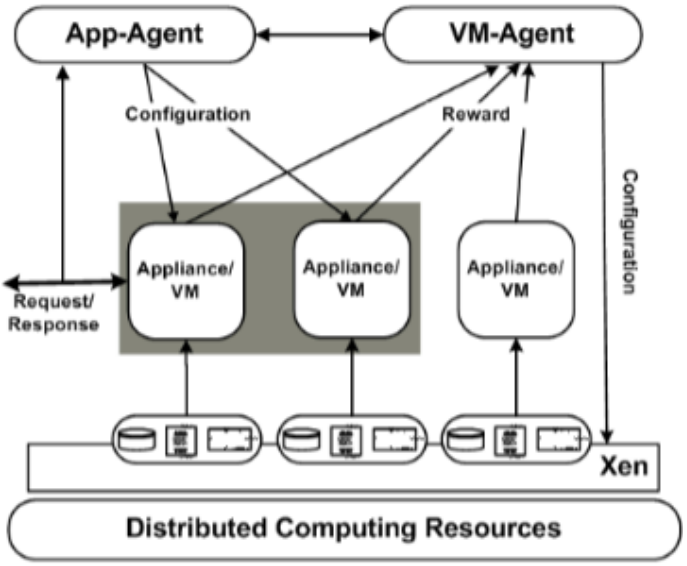


FIGURE 3.16 – Architecture de *URL* proposée par [XRB12].

Bu et al ont aussi proposé *CoTuner* [BRX13], un *framework* proche de l'architecture et des intentions de *URL* (cf. Figure 3.17) qui vise à configurer à la fois les composants *middleware* (e.g. paramétrage d'un serveur *Tomcat*) et les VMs sous-jacentes (i.e. dimensionnement vertical). Chaque *Cluster* se voit appliquer un *VM-Agent* en charge de la gestion des ressources des VMs correspondantes et dont le but est de conduire le système dans une configuration optimale (e.g. maximiser l'utilisation des ressources, le débit, etc.). De la même manière, chaque application se voit attribuer un *App-Agent* en charge de respecter les objectifs propres à l'application et ainsi assumer le SLA signé. En plus de l'apprentissage par renforcement, ce *framework* met en œuvre la méthode *Simplex* (i.e. algorithme de résolution des problèmes d'optimisation linéaire). La méthode *Simplex* est utilisée pour réduire considérablement l'espace des états possibles (i.e. configurations envisageables) en vue d'obtenir un ensemble "raisonnable" de configurations prometteuses et ainsi permettre aux agents d'apprentissage par renforcement de mener une recherche sur cet ensemble réduit, en évitant les configurations moins satisfaisantes.

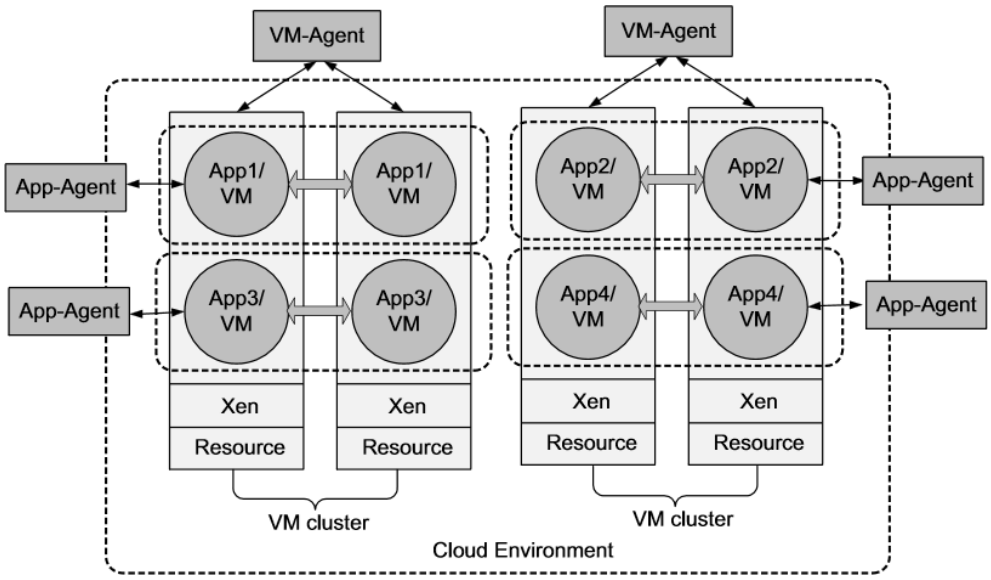


FIGURE 3.17 – Architecture de *CoTuner* proposée par [BRX13].

Dawoud et al [DTM11] [MDT13] ont proposé *Elastic VM*, une architecture visant à contrôler dynamiquement le dimensionnement vertical des ressources. Il s'agit d'ajuster au plus prêt la quantité de ressource des VMs (i.e. CPU, RAM) en fonction de la demande dans le but d'assumer des SLOs prédéfinis au niveau applicatif (i.e. temps de réponse). Dans leur solution, les auteurs s'intéressent aussi à la reconfiguration de la couche *middleware* à travers le paramétrage du serveur *HTTP*.

Les auteurs s'appuient sur une approche prédictive pour estimer les besoins futurs en termes de CPU. La solution est basée sur un contrôle adaptatif qui considère les dernières allocations ainsi que l'historique de l'utilisation de CPU. Le redimensionnement vertical est implémenté par le biais du gestionnaire d'allocation des ressources (i.e. ordonnanceur *Xen Credit scheduler* [xen15a]) qui permet de répartir la capacité de CPU de la machine hôte (i.e. PM) sur les différentes VMs en termes de pourcentage (e.g. 50% correspond à la moitié d'un CPU physique alors que 400% correspond à 4 CPU) tout en fixant éventuellement des seuils maximum.

La surveillance de *Elastic VM* (cf. Figure 3.18) est assurée par deux modules qui sont le *Resources monitor* et le *Performance monitor*. Le premier a la charge de la collecte des métriques liées à l'utilisation des ressources des VMs (par le biais de l'outil *xentop*). Le *Performance monitor* a quant à lui la responsabilité de la collecte des métriques liées aux performances (i.e. temps de réponse moyen, débit d'entrée et de sortie).

Le contrôle de l'élasticité de *Elastic VM* repose essentiellement sur trois contrôleurs fonctionnant en parallèle. Le premier contrôleur appelé *CPU controller* (i.e. *nested loop controller* [ZWS06]) vise à prédire la prochaine allocation CPU en s'appuyant sur les allocations précédentes et les consommations CPU correspondantes. Le second contrôleur, nommé *Memory controller* est une implémentation du contrôleur proposé par [HZPW09] et tend à ajuster la mémoire RAM. Les deux premiers contrôleurs précédemment cités admettent une portée d'action sur la couche IaaS.

Enfin, le dernier contrôleur appelé *Application controller* est en charge de contrôler le serveur *Apache* [apa15] et plus précisément la valeur du paramètre *MaxClients* qui permet de fixer la limite maximale de requêtes simultanées que le serveur peut prendre en charge. S'il y a plus d'utilisateurs simultanés que *MaxClients*, les requêtes seront mises en file d'attente. Les auteurs s'appuient sur certains travaux précédents [LSD+03] qui ont étudiés la valeur optimale pour *MaxClients* dépendamment des ressources virtuelles attribuées (i.e. CPU et RAM) pour obtenir les meilleurs résultats en termes de débit (i.e. *throughput*) et de temps de réponse. La portée de ce contrôleur peut être associée à du PaaS dans le sens où il s'agit d'ajuster la configuration du *middleware* (i.e. serveur *Apache*). Les résultats des expériences montrent que ce contrôleur permet de maintenir les performances tout en réduisant les violations de SLOs et le taux de requêtes admettant des temps de traitement trop important (i.e. *timeout*).

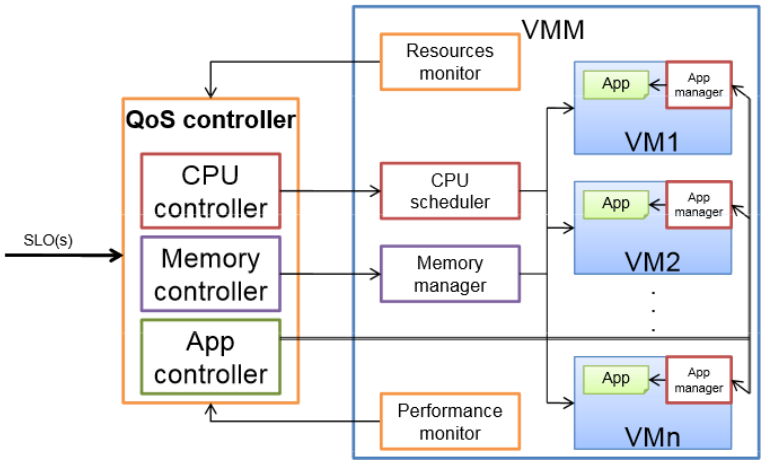


FIGURE 3.18 – Architecture de *Elastic VM* proposée par [DTM11].

Dans [ZHLM10], Zhang et al présentent *SmartRod*, un *framework* s'appuyant sur les boucles autonomiques *MAPE-K* [KC03] pour le dimensionnement automatique de ressource. La solution repose sur une approche réactive et s'intéresse au dimensionnement vertical des ressources (i.e. CPU et RAM). Zhang et al définissent leur solution comme un service PaaS pour la gestion automatique des ressources du Cloud. En tant que fournisseur PaaS, leur objectif est d'offrir une utilisation efficace des ressources IaaS tout en respectant les contrats de niveaux de service signés avec les fournisseurs SaaS. L'approche proposée repose sur deux boucles autonomiques *MAPE-K* (cf. Figure 3.19) dont la phase d'*analyse* est implémentée sous forme de règles à base de seuils. La première boucle *MAPE-K*, appelée *resource consumption optimization loop* est dédiée aux ressources PaaS. Son objectif est de surveiller les métriques de cette couche (e.g. *threadBusy* et *threadCount* d'un serveur *Tomcat* [tom15] ou *QueueSize* d'un serveur *JBoss* [jbo15]) et d'ajuster les paramètres de la couche *middleware* (e.g. *maxThreads* de *Tomcat*) dans le but d'améliorer l'utilisation des ressources et les performances associées. La seconde boucle, nommée *resource allocation loop* porte quant à elle sur les ressources IaaS et vise à ajuster la quantité de ressource (i.e. CPU et RAM) de la VM hébergeant l'application pour respecter un certain niveau de performance. La boucle d'optimisation (i.e. *resource consumption optimization loop*) est exécutée en continu pour améliorer l'utilisation des ressources et les performances. Lorsque les actions opérées par la première boucle ne permettent pas un état satisfaisant, une demande de ressource est faite à la seconde boucle (i.e. *resource allocation loop*) chargée de déterminer et d'allouer la quantité de ressource IaaS nécessaire. La coordination des deux boucles est basée sur les performances de l'application, les ressources disponibles ainsi que les SLAs signés entre le fournisseur SaaS et le fournisseur PaaS. Bien que les auteurs n'insistent pas sur ce point, leur solution privilégie les adaptations PaaS (i.e. boucle 1), dont l'effort et le coût de reconfiguration est moindre en comparaison aux adaptations IaaS (i.e. boucle 2). Cela s'explique sans doute du fait qu'un fournisseur PaaS, comme n'importe quel fournisseur, tend à augmenter son profit en diminuant ses coûts (i.e. coûts des ressources IaaS).

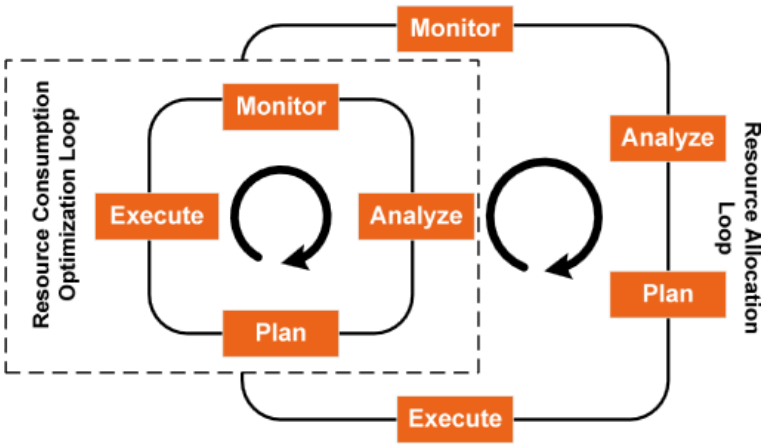


FIGURE 3.19 – Boucles de contrôle de la solution *SmartRod* proposée par [ZHLM10].

Couches IaaS et SaaS

Garlan et al ont poursuivi leur travail autour du *framework Rainbow* [GSC09] [CGS09] en s'appuyant sur le cas d'utilisation *Znn.com* [znn15], une application web multi-tier offrant un service de nouvelles sous forme de contenu multimédia (i.e. vidéos). L'objectif pour le gestionnaire de *Znn.com*, que l'on peut apparenter à un fournisseur SaaS, est de répondre aux requêtes de ses utilisateurs avec un niveau de QoS acceptable (i.e. temps de réponse raisonnable) tout en respectant une contrainte budgétaire portant sur les ressources sous-jacentes (i.e. coûts des serveurs). Il s'agit donc pour le fournisseur de service d'ajuster la quantité de ressources (i.e. nombre de serveurs) en fonction de la demande en vue de satisfaire les objectifs en termes de QoS.

Dans certains cas exceptionnels (e.g. pic de charge rapide et important, parfois référencé sous le nom *Slashdot effect* ou *slashdotting* [sla15]), il se peut que le service de nouvelles ne puisse pas respecter ses objectifs en termes de QoS malgré le fait que l'infrastructure sous-jacente soit dimensionnée à son maximum (i.e. plus de serveur disponible ou limite budgétaire atteinte). Dans ce cas de figure, *Znn.com* offre la possibilité d'ajuster le format des nouvelles fournies aux utilisateurs en passant d'un contenu multimédia (i.e. vidéo) à un contenu textuel en paramétrant dynamiquement l'application. Bien que la qualité de rendu du service (i.e. *content quality*) soit moindre du point de vue de l'utilisateur, cela permet de soulager les ressources (i.e. bande passante) ce qui va se ressentir en termes de performance (i.e. temps de réponse) et permettre d'absorber le pic de charge en augmentant la capacité de ressources en adaptant le contenu (i.e. logiciel) plutôt que le contenant (i.e. ressources physiques).

Le fait que le service de nouvelles proposé par *Znn.com* offre différents modes (i.e. texte et vidéo) accroit les possibilités d'adaptation du point de vue du fournisseur de service. Il a désormais la possibilité d'ajuster les ressources de l'infrastructure et les services de l'application elle-même en basculant le mode de ses services. Les auteurs de [GSC09] définissent ainsi 4 stratégies d'adaptation, gérée par le composant *Adaptation Manager* de *Rainbow* : *EnlargeServerPool* (i.e. *Scale Out*), *ShrinkServerPool* (i.e. *Scale In*), *SwitchToTextualMode* et *SwitchToMultimediaMode*. Bien que les auteurs de [GSC09] [CGS09] n'évoquent pas directement les modèles de services IaaS et SaaS du Cloud comme définis par le *NIST* [MG11], on peut parler d'adaptation multi-couche (i.e. ressources de l'infrastructure et services applicatifs).

Zhu et al proposent un *framework* [ZA10] qui utilise conjointement l'apprentissage par renforcement et la théorie du contrôle pour une gestion pro-active du dimensionnement ainsi que la reconfiguration des applications. En effet, en plus du dimensionnement vertical des ressources IaaS, les auteurs s'intéressent à des applications dites "adaptatives" (*adaptive applications*) proposant des services dont on peut configurer dynamiquement différents paramètres (*adaptive parameters*) dépendamment du contexte d'exécution.

Un des cas d'utilisation proposé dans ce travail concerne une application de rendu volumique direct (*volume rendering*) permettant d'afficher une projection 2D d'une série de données 3D. Cette application offre différents paramètres que l'on peut configurer comme la taille des images générées ou encore la résolution de celles-ci. La configuration de ces différents paramètres va avoir un impact éventuel sur les performances du service (e.g. temps de réponse) ou encore la consommation de ressource (e.g. CPU). Il s'agit ainsi de paramétrer dynamiquement l'application et d'ajuster la quantité de ressource en vue de maximiser le profit du fournisseur de service tout en respectant des contraintes en termes de performances (i.e. SLO sur les temps de réponse) et de coûts (i.e. contraintes budgétaires).

Dans un travail précédent [CGS06], Cheng et al s'étaient déjà appuyés sur un cas d'utilisation similaire appelé *Z.com*. Cette application est l'ancêtre de *Znn.com* et propose elle-aussi un service de nouvelles en ligne paramétrable offrant différents niveaux de rendu (i.e. format textuel ou graphique). Les auteurs s'intéressent à la reconfiguration du système en considérant l'adaptation comme un problème multi-objectifs. Ils considèrent trois critères pour la décision d'adaptation : les temps de réponse constatés par les utilisateurs du service, la qualité du rendu des nouvelles fournies et les coûts induits par les ressource allouées (i.e. serveurs physiques). Ces critères induisent différents compromis qui doivent être pris en considération dans la décision de reconfiguration en suivant les préférences d'adaptation prédéfinies par les différents acteurs (i.e. fournisseur ou utilisateur). [CGS06] distingue quatre actions d'adaptation, appelées *tactics*. Deux actions concernent l'application et permettent de paramétrer le service de nouvelles (i.e. basculer le service du mode texte au mode graphique et réciproquement) et deux actions concernent l'infrastructure et permettent d'ajuster la quantité de ressources (i.e. *Scale Out* et *Scale In*).

Le système est modélisé comme une structure de *Kripke*, un modèle de calcul proche d'un automate fini non-déterministe. Le système est ainsi représenté par un graphe dont les nœuds représentent les états possibles du système et dont les arêtes représentent les transitions entre états. Chaque tactique (i.e. action) se voit attribuer une note utilitaire. Cette note est calculée en faisant la somme des notes de quatre attributs dérivés des critères évoqués ci-dessus (i.e. parmi un ensemble de notes prédéfinies comprises entre 0 et 1). Il s'agit du temps de réponse, de la qualité de rendu et du budget auxquels vient s'ajouter l'attribut *disruption* permettant d'indiquer le coût de la reconfiguration (e.g. temps de mise en place de la tactique). Enfin, les préférences d'adaptation peuvent être précisées en associant un poids aux différents attributs en vue d'orienter la prise de décision.

Couches IaaS, PaaS et SaaS

Kranas et al ont proposé *ElaaS* (*Elasticity-as-a-Service* [KAMV12]), un *framework* pour une gestion autonomique de l'élasticité du Cloud adressant les différents modèles de service (i.e. couches IaaS, PaaS et SaaS). Le *framework* intègre cinq composants différents pour le traitement de l'élasticité (cf. Figure 3.20). Ces composants sont unitairement accessibles au moyen de web services.

— *ElaaS Core* : chargé de coordonner l'activité globale. Il gère l'initialisation des autres composants (i.e. paramétrage) et leurs canaux de communication ainsi que les éléments de stockage de données requis pour le fonctionnement des composants. Á l'exécution, il est chargé d'orchestrer les échanges entre les autres composants.

— *Application Manager* : récupère la description de l'application lors de la phase d'initialisation ainsi que la description des SLAs et les métriques essentielles à surveiller (*Key Performance Indicator - KPI*).

— *Monitoring Manager* : chargé de la communication avec les différentes sondes. Il est configuré pour ne consommer que les métriques connues de l'*Application Manager*. De plus, il gère le stockage des données observées de sorte à créer et maintenir un historique des relevés des métriques utilisé pour la prise de décision d'élasticité.

— *Business Logic Manager* : chargé d'orchestrer la prise de décision d'élasticité. Néanmoins, il n'inclut pas de logique de prise de décision qui reste à la charge du développeur ou d'un service tiers. En effet, *ElaaS* est un *framework* qui offre la possibilité d'intégrer de multiples solutions de prise de décision pouvant suivre différentes approches (e.g. réactive ou pro-active).

— *Action Manager* : joue un rôle de passe-plats entre les décisions émises par le *Business Logic Manager* et les composants applicatifs ou les éléments de la plateforme Cloud sous-jacente (i.e. *actuators*). Il peut aussi être amené à compléter les informations quant aux décisions émises afin d'affiner les plans de reconfiguration avant de les exécuter (e.g. ajouter un répartiteur de charge dans le cas d'une décision de *Scale Out*).

La modélisation de l'application repose sur deux modèles avec un niveau de granularité différent. Le graphe d'application, qualifié de haut-niveau, renseigne l'ensemble des dépendances d'une application et repose sur une description à gros grains listant les logiciels requis et les artefacts applicatifs (i.e. *Tomcat* [tom15], *MySQL*, etc.). Le second modèle, nommé graphe de déploiement, est dérivé du graphe d'application et éventuellement raffiné à l'exécution en fonction des logiques implantées par les composants *Business Logic Manager* et *Action Manager*. Le graphe de déploiement offre la vision de l'architecture du système durant son exécution. *ElaaS* n'introduit pas de limitation quant aux possibilités d'opérations d'élasticité (horizontale, verticale, migration ou externalisation). Cependant celles-ci restent à la charge de l'administrateur de l'application.

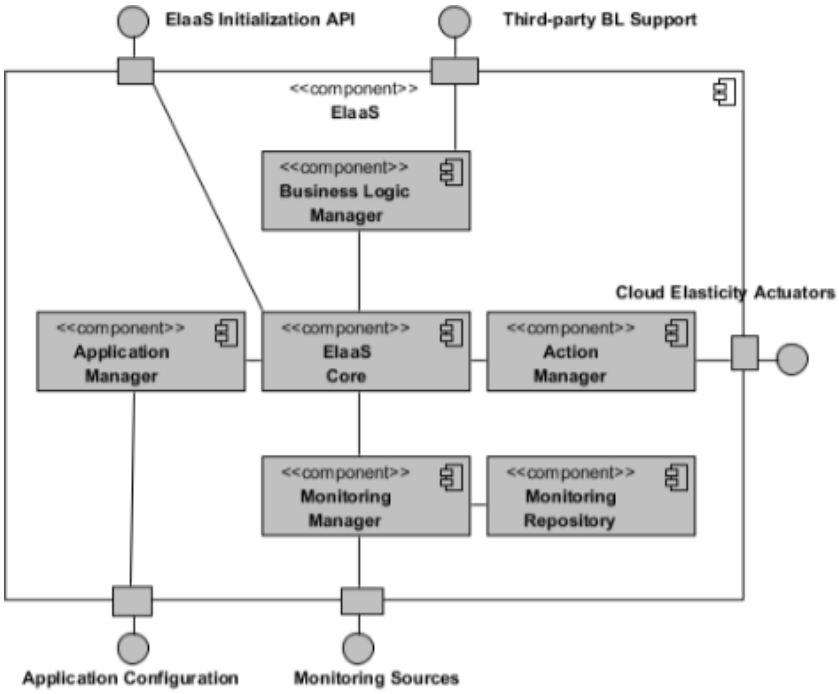


FIGURE 3.20 – Architecture du *framework ElaaS* proposé par [KAMV12].

Marquezan et al [MWM+14] se sont intéressés à l'adaptation automatique des applications Cloud. Néanmoins, contrairement à la plupart des travaux de l'état de l'art qui n'adressent qu'une couche du nuage (e.g. IaaS), ce travail tend à considérer multiples couches en vue de bonifier la capacité d'adaptation globale du Cloud. Ainsi, les auteurs considèrent quatre couches distinctes. Bien que ces couches soient distinctes, la politique d'adaptation et la gestion des ressources d'une couche peuvent avoir une influence sur les couches sous-jacentes (cf. Figure 3.21). Les auteurs partent du constat que la plupart des solutions actuelles n'adressent que partiellement deux points cruciaux dans les environnements Cloud :

— la *multiplicité des ressources Cloud* (e.g. applications, composants, VM, PM, etc.) induit de nombreux mécanismes d'adaptation. Néanmoins, les solutions existantes, en ne considérant qu'une ou deux couche(s) du Cloud, ne bénéficient pas de tous les mécanismes d'adaptation offerts par ce paradigme.
— l'existence d'*interférences entre les actions d'adaptation* portant sur différentes couches du Cloud est problématique. En se focalisant sur une unique couche du Cloud, les solutions actuelles prennent généralement des décisions d'adaptation indépendamment des autres couches. Concrètement, le fait d'appliquer de manière indépendante les différents mécanismes conduit souvent à des adaptations globalement mauvaises pouvant affecter négativement la QoS, les performances ou encore l'efficience énergétique ou financière du Nuage dans sa globalité.

Marquezan et al ne donnent pas de solution à proprement parlé, néanmoins, leur travail répertorie un ensemble de techniques et mécanismes d'adaptation en les rattachant aux différentes couches d'un environnent Cloud. De plus, ils dressent les éventuelles corrélations entre les ressources et les mécanismes provenant des différents couches. Le résultat de ce travail est un modèle conceptuel d'adaptation du Cloud mettant en lumière les différentes possibilités de reconfiguration offertes. Ce modèle permet d'identifier les potentiels besoins de coordination ou encore les éventuelles synergies entre mécanismes d'adaptation.

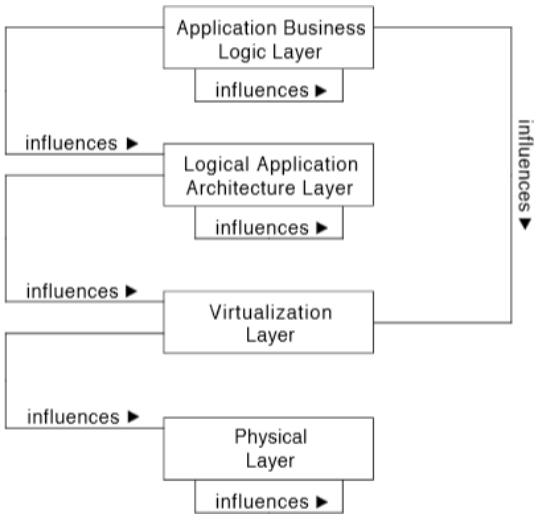


FIGURE 3.21 – Couches distinctes d'un environnement Cloud selon [MWM+14].

Truong et al [TDC+14] se sont intéressés à l'élasticité multi-couche dans le Cloud. Les auteurs indiquent que l'un des principaux challenges des futures offres PaaS est d'adresser l'élasticité des *software-defined elastic systems* (*SES*). Les *SES* sont définis comme des systèmes Cloud faisant intervenir des ressources diverses et variées (e.g. services Web, conteneurs logiciels, VMs, etc.) et dont l'élasticité est contrôlée par des APIs admettant différents niveaux d'abstraction. Comme dans leurs travaux précédents [CMTD13b] [MCTD13] [CMTD13a], Truong et al adressent l'élasticité selon trois dimensions (i.e. *resource*, *cost* et *quality*). Dans leur travail, les auteurs répertorient un ensemble d'opérations d'élasticité existantes (i.e. *elasticity primitive operations*) concernant différents types de ressources (i.e. VM, conteneur logiciel, etc.) et accessibles via des APIs (cf. Figure 3.22). De plus, ils indiquent les relations entre ces multiples opérations. Dans le but de contrôler, surveiller et tester l'élasticité des *SES*, Truong et al ont proposé *CoMoT*, une plate-forme en tant que service (i.e. PaaS) adressant l'élasticité multi-couche dans le Cloud. La solution s'appuie sur différents outils développés par les auteurs dans des travaux précédents. On peut noter l'utilisation de *MELA* [MCTD13] pour la surveillance du système ou encore l'utilisation de *SYBL* [CMTD13b], un langage d'élasticité pour le contrôle de l'élasticité. Nous présenterons davantage ces outils dans la section 3.4.

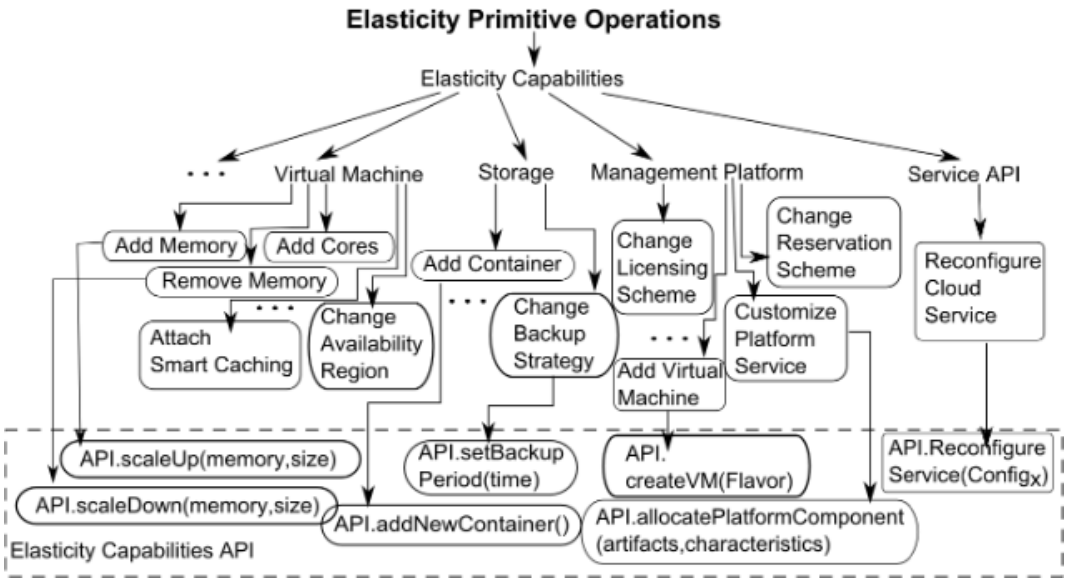


FIGURE 3.22 – Opérations primitives d'élasticité selon [TDC+14].

3.4 Administration de l'élasticité

Nous répertorions dans cette section de nombreux travaux portant sur l'administration de l'élasticité dans le Cloud. Nous classifions ces travaux en quatre catégories. La première catégorie concerne les travaux s'intéressant à résoudre les compromis induits par l'élasticité (e.g. coût-performance). La seconde catégorie adresse la configuration de l'élasticité, c'est-à-dire comment celle-ci peut être paramétrée et configurée. La troisième catégorie, liée à la précédente, s'intéresse aux travaux donnant des indications quant à la place de l'administrateur humain pour la configuration de l'élasticité. Enfin, la dernière catégorie regroupe des travaux dont les solutions visent à outiller le processus d'élasticité avec des langages dédiés.

3.4.1 Compromis induits par l'élasticité

Jung et al ont proposé le *framework Mistral* [JHJ+10] visant à optimiser la consommation énergétique, les performances des applications ainsi que les coûts d'adaptation. Les auteurs considèrent de multiples actions d'adaptation portant uniquement sur la couche IaaS à savoir le dimensionnement horizontal et vertical, la migration ainsi que le fait d'allumer/éteindre des machines physiques. La solution de Jung et al peut être qualifiée de prédictive et repose sur certaines techniques de la théorie du contrôle ainsi qu'un modèle de file d'attente (i.e. *Layered Queuing Network*) pour prédire les performances de l'application, un modèle analytique pour prédire la consommation énergétique et des expérimentations *offline* pour prédire les coûts dépendamment de la charge de travail et de la configuration. De plus, les auteurs s'intéressent à prédire la charge de travail en s'appuyant sur l'historique de la demande passée. Les auteurs partent du principe que la plupart des travaux de l'état de l'art ne s'intéressent qu'à la résolution de compromis faisant intervenir seulement deux critères (e.g. coût-performance). Les expérimentations effectuées dans [JHJ+10] visent donc à comparer le *framework Mistral* à trois stratégies cherchant chacune à optimiser deux critères parmi les performances, la consommation énergétique et les coûts. Les résultats montrent que *Mistral*, en considérant à la fois les performances, la consommation énergétique et les coûts dans son contrôle de l'élasticité, surpasse globalement les autres stratégies qui tendent à optimiser un sous ensemble de ces critères.

Dans leur travail, Roy et al [RDG11] décrivent un algorithme pour l'allocation pro-active de ressources (i.e. dimensionnement horizontal) pour les applications multi-tier. La prédiction de la charge de travail repose sur l'analyse des séries chronologiques et s'appuie sur le modèle *ARMA* (i.e. *AutoRegressive Moving Average*). Le modèle *ARMA* permet de traiter les séries dites stationnaires (i.e. dont la structure du processus sous-jacent reste la même au cours du temps) et est composé d'une partie auto-régressive (*AR*) et d'une partie moyenne-mobile (*MA*). Le but de l'algorithme proposé est de prédire la demande future à court terme en s'appuyant sur un historique limité de la charge de travail (i.e. 3 dernières observations), puis à partir de cette prédiction, estimer la QoS future (i.e. temps de réponse) en vue d'ajouter/retirer la bonne quantité de ressource (i.e. VMs) juste à temps (i.e. *just-in-time resource allocation*). La solution intègre la notion de SLA (i.e. temps de réponse) et tend naturellement à minimiser les coûts du dimensionnement automatique tout en assumant les SLAs signés. Un contrôleur a la charge, à partir des temps de réponse estimés, de trouver la meilleure configuration en termes de coût. Les auteurs proposent une fonction de coût intégrant trois composantes : le coût des ressources, les pénalités en cas de manquement au SLA et le coût de la reconfiguration. Chacune de ces composantes de la fonction de coût se voit associer un poids. Cela permet au système d'orienter la prise de décision pour satisfaire certaines préférences. Ainsi, on peut par exemple augmenter le poids associé aux pénalités de SLA si l'on souhaite minimiser les violations de contrat de niveau de service. Les expérimentations montrent des résultats intéressants aussi bien pour le fournisseur de service (i.e. réduction des coûts) que pour les utilisateurs (i.e. peu de violation SLA). Néanmoins, les expérimentations sont réalisées sur un environnement limité (i.e. 2 à 4 VMs) et aucune garantie n'est donnée sur le passage à l'échelle de l'algorithme lui-même.

Dans ses travaux de thèse [Jun13], Frederico Alvares s'est intéressé à la gestion autonome des applications et des infrastructures pour améliorer la QoS et l'efficacité énergétique de l'informatique en nuage. Le constat est que chaque application peut avoir ses propres objectifs et exigences, de même que l'infrastructure. 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 ne sont pas toujours visibles au niveau de l'infrastructure. De la même manière, les applications ne connaissent pas nécessairement les détails de l'infrastructure. L'approche retenue est la mise en place de plusieurs gestionnaires autonomes (i.e. boucles *MAPE - K*). Ainsi, chaque application est équipée de son propre gestionnaire en charge de déterminer la quantité de ressources nécessaires pour offrir la meilleure QoS possible. Un autre gestionnaire, au niveau de l'infrastructure, gère les ressources physiques en prenant en considération le critère de consommation énergétique. Afin de gérer le manque de synergie entre les applications et l'infrastructure, cette thèse s'intéresse à 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 (e.g. prise de décision d'adaptation contradictoire). Cette thèse vise aussi à étendre les capacités de l'élasticité des ressources en traitant les applications comme des boîtes blanches sur lesquelles il est possible d'effectuer des opérations de reconfiguration. En conséquence, les applications deviennent plus sensibles et réactives à l'évolution de l'environnement d'exécution, peu importe si les changements se produisent au niveau de l'application elle-même (e.g. augmentation de la demande) ou au niveau de l'infrastructure (e.g. restriction des ressources).

Moldovan et al ont présenté *MELA* [MCTD13], un *framework* pour la surveillance et l'analyse de l'élasticité des services du Cloud. Le *framework* offre la possibilité aux développeurs et fournisseurs de services de surveiller et d'analyser le comportement des différentes ressources du nuage (e.g. VMs, applications, etc.) en vue de contrôler l'élasticité du système dans sa globalité. Moldovan et al présentent *MELA* comme un service de surveillance et d'analyse de l'élasticité à la demande (i.e. *elasticity space monitoring and analysis as a service*). Les auteurs proposent un ensemble de métriques à surveiller pour l'analyse d'un comportement élastique (i.e. *elastic behavior*). Ils regroupent ces différentes métriques selon trois catégories (i.e. *elasticity dimensions*) : *Cost*, *Quality* et *Resource*.

La Figure 3.23 donne un aperçu de cette catégorisation. En se basant sur ce modèle, l'utilisateur de *MELA* (e.g. administrateur, développeur) peut définir les métriques concernées pour différents types de ressources. De plus, celui-ci peut spécifier ses besoins d'élasticité (i.e. *elasticity requirements*) sous forme de seuils (i.e. *boundary*) associés aux métriques lui permettant de capturer le comportement élastique de celles-ci. La solution offre aussi la possibilité de visualiser dynamiquement la topologie des ressources ainsi que l'état courant des différentes métriques observées.

3.4.2 Configuration de l'élasticité

Maurer et al [MBS12] ont proposé une approche réactive de dimensionnement automatique adressant trois objectifs parfois contradictoires. Il s'agit de minimiser le nombre de violation de contrats de service, maximiser l'utilisation des ressources et enfin minimiser l'utilisation d'actions de reconfiguration chronophages et énergivores. La solution repose sur les règles à base de seuils. Néanmoins, contrairement à l'approche standard, la valeur des seuils n'est pas statique. En effet, l'objectif de Maurer et al est de paramétrer dynamiquement et automatiquement la valeur des seuils dépendamment de la charge de travail observée. Il s'agit dans un premier temps de classifier la charge de travail en différentes catégories (i.e. *Workload Volatility Class*).

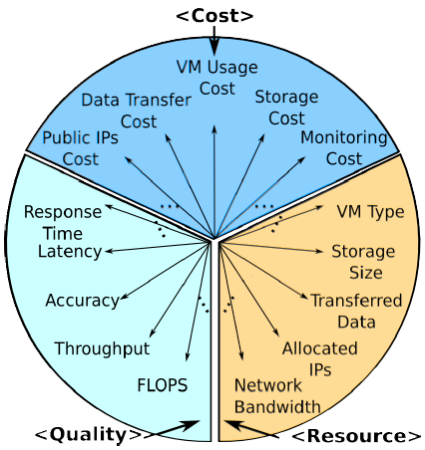


FIGURE 3.23 – Dimensions d'élasticité et métriques associées selon [MCTD13].

Les auteurs considèrent quatre catégories de charge de travail (i.e. *Low, Medium, Medium-High* et *High*) auxquelles vont être associés différents seuils. Ainsi, à l'exécution, suivant la charge de travail observée, on va identifier la catégorie de celle-ci et appliquer aux règles de dimensionnement automatique les seuils correspondants. Cela a un impact sur la "politique de reconfiguration" qui sera plus ou moins agressive en fonction des seuils appliqués. Les résultats des expérimentations, portant sur des traces réelles de charge de travail, montrent que l'approche de paramétrage dynamique des seuils offre généralement de meilleurs résultats que l'approche standard, admettant des seuils statiques.

Casalicchio et al [CS13] ont développé deux architectures pour le dimensionnement autonomique de ressources IaaS en s'appuyant sur les boucles de contrôle *MAPE-K*. Les auteurs s'intéressent au passage à l'échelle horizontal en s'appuyant sur les services de *Amazon EC2* [EC215b]. Les deux architectures sont appelées *Limited ASP control architecture* et *Partial ASP control architecture* (cf. Figure 3.24). Le terme *ASP* (*Application Service Providers*) désigne ici le fournisseur SaaS. La première architecture (*Limited* - Figure 3.24 de droite) s'appuie pleinement sur les services de *Amazon EC2* pour gérer l'adaptation à savoir *Amazon Auto Scaling* [EC215a] pour la définition des règles et *CloudWatch* [clo15b] pour la partie surveillance (*M* de *MAPE-K*). Le fournisseur SaaS doit paramétrer ses politiques de dimensionnement automatique en spécifiant les règles d'adaptation (i.e. seuils et actions) mais le contrôle reste à la charge du IaaS. Dans le cas de la seconde architecture (*Partial* - Figure 3.24 de gauche), le fournisseur SaaS a un contrôle total sur la décision d'adaptation (*A* et *P* de *MAPE-K*) bien qu'il s'appuie sur les services du fournisseur IaaS pour ce qui relève de la surveillance et de l'exécution de l'adaptation (*M* et *E* de *MAPE-K*). Les auteurs proposent aussi cinq politiques réactives d'allocation de ressource. Quatre de ces politiques s'appuient pleinement sur les services de *Amazon EC2* évoqués précédemment (i.e. architecture dite *Limited*). Il s'agit d'exécuter l'adaptation après une (ou deux) alarme(s) déclenchée(s) lorsque la valeur d'une métrique (i.e. utilisation de ressource ou temps de réponse) dépasse un seuil prédéfini. La dernière politique, applicable uniquement dans le cas de la seconde architecture qui donne le contrôle de l'adaptation au fournisseur SaaS, est plus complexe et s'intéresse à la demande moyenne (*average arrival rate*) en la recoupant avec un SLO sur les temps de réponse pour estimer l'adaptation à entreprendre à savoir la quantité de ressource à ajouter/retirer (contrairement à une règle de dimensionnement automatique classique dont l'action s'avère statique). Les auteurs ont expérimentés les deux architectures et les cinq politiques sur un vrai *testbed* avec des charges de travail réalistes (i.e. traces de *Wikipedia*). Les résultats montrent que les possibilités offertes par l'architecture qualifiée de *Partial* permettant aux utilisateurs (i.e. fournisseurs SaaS) de définir leurs propres politiques d'adaptation offrent globalement de meilleurs résultats en termes de performance et de coût en comparaison aux services limités de dimensionnement automatique proposés par les fournisseurs IaaS.

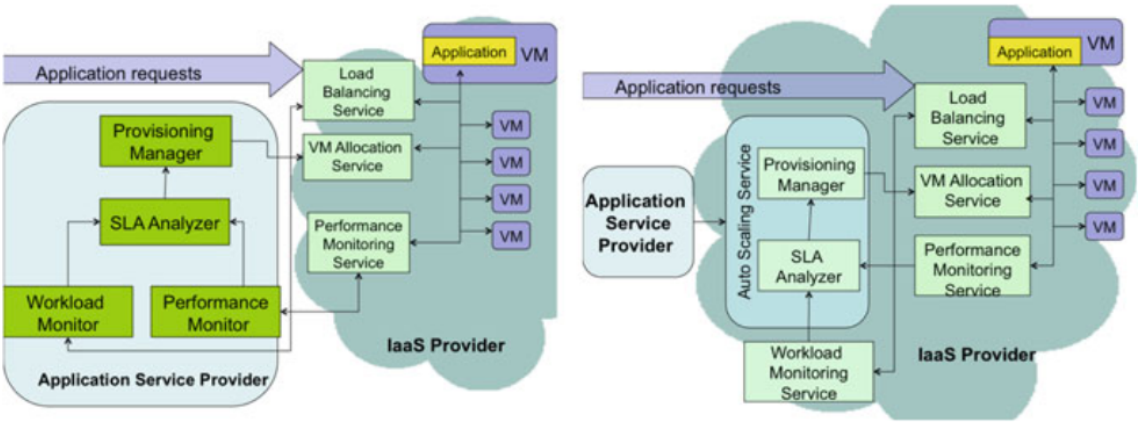


FIGURE 3.24 – *ASP control architectures* (*Partial* et *Limited*) proposées par [CS13].

Yousri Kouki a proposé une solution complète pour la gestion des contrats de service (SLA) du Cloud [Kou13]. Les contributions de cette thèse sont :

— un langage dédié à la définition de contrats SLA dans le Cloud, nommé *CSLA* (*Cloud Service Level Agreement*). Le langage adresse toutes les couches du Cloud (i.e. IaaS/PaaS/SaaS) et permet de gérer finement les violations SLA. En effet, en plus des caractéristiques classiques de définition des SLOs, *CSLA* propose un modèle pour gérer plus finement les violations. Celui-ci repose sur la dégradation fonctionnelle (i.e. modes de service), la dégradation de QoS ainsi qu'un modèle de pénalité avancé.

— un *framework* de dimensionnement automatique dirigé par les SLA, nommé *HybridScale*. L'architecture conceptuelle de ce *framework* repose sur une boucle de contrôle autonomique *MAPE-K*. *HybridScale* suit une approche hybride (i.e. réactive-proactive) et considère des actions d'adaptation multi-couche à savoir le dimensionnement des ressources IaaS ainsi que la reconfiguration de l'application SaaS à travers la dégradation fonctionnelle et la dégradation de QoS. Le tout est paramétrable via des politiques prédéfinies de gestion d'élasticité (i.e. *Best Policies*).

Fawaz Paraiso, dans son travail de thèse [Par14], a proposé un modèle d'applications qui adresse quatre défis : la portabilité, l'approvisionnement, l'élasticité et la haute disponibilité des applications Cloud. Ce modèle est basé sur le standard *SCA* (*Service Component Architecture* [MR09]) du consortium *OASIS* et vise à concevoir de manière simple et cohérente des applications distribuées à large échelle pour un environnement multi-nuages. Fawaz Paraiso a ainsi proposé *soCloud*, une plateforme multi-nuages à base de composants et services qui repose sur le modèle évoqué précédemment et adresse le déploiement, l'exécution et la gestion des applications réparties à travers plusieurs nuages. Elle est constituée de deux parties qui communiquent ensemble : l'*agent* et le *master*. L'*agent* permet d'héberger, de surveiller et d'exécuter les applications en tant que service. Le *master*, quant à lui, gère le déploiement, l'approvisionnement de ressources, la coordination, la disponibilité ainsi que la surveillance des changements d'états des applications déployées. Une autre contribution de cette thèse est la proposition d'un langage dédié à l'élasticité pour exprimer efficacement l'élasticité d'applications multi-nuage par l'abstraction. Ce langage permet de contrôler l'élasticité non seulement en termes de ressources mais aussi en termes de coût et de qualité de service. Il s'agit d'un langage de haut niveau, guidé par les événements, permettant au développeur d'exprimer de manière intuitive le comportement élastique de chaque composant de son application sous forme de règles. Le Code 3.1 donne un exemple d'expression du langage définissant une action qui augmente la capacité du processeur (i.e. dimensionnement vertical) lorsque le pourcentage d'utilisation de ce dernier est supérieur à 80% sur une période de 2 minutes.

```
scaling up when (average(cpuUsage, 120s) > 80%)
```

Code 3.1 – Expression d'une action avec le *DSL* proposé par [Par14].

Loic Letondeur a proposé *Vulcan* [Let14], un outil de spécification de l'élasticité. *Vulcan* repose sur un modèle d'applications élastiques, lui-même composé de deux sous-modèles : un modèle par extension et un modèle par intention. Le modèle par extension décrit l'état courant de l'architecture de l'application. L'architecture de l'application est modélisée sous forme de composants et des conteneurs auxquels viennent s'ajouter des informations relatives aux placements de ceux-ci, à leurs liaisons respectives ainsi que des paramètres de configuration. Le but est de trouver une configuration cible respectant certaines contraintes définies dans le modèle par intention.

Le modèle par intention, quant à lui, concerne la description des contraintes devant être satisfaites durant les calculs d'architectures cibles. Il utilise un formalisme sous forme de requêtes ensemblistes pour l'écriture des contraintes. Ces requêtes décrivent les modifications à opérer sur des composants ou conteneurs du modèle par extension en vue de satisfaire les exigences de l'utilisateur. Le modèle distingue quatre types de contraintes : l'héritage, les liaisons, les placements et les configurations. Chaque type de contrainte induit un ensemble fini de modifications sur l'architecture courante (i.e. modèle par extension).

Une autre contribution de cette thèse est un algorithme de planification permettant la résolution des différents paramètres de l'élasticité. Celui-ci repose sur le raisonnement révisable issu du domaine de l'intelligence artificielle. Ce mode de raisonnement permet la composition automatique de contraintes simples et intelligibles en de nouvelles contraintes complexes.

3.4.3 Place de l'administrateur

Saleh et al [SGB+13] se sont intéressés à la surveillance et au dimensionnement automatique d'environnement Cloud comme cas d'utilisation pour l'emploi du *Complex Event Processing* (*CEP*). Dans ce travail, les auteurs s'intéressent essentiellement à la surveillance et à l'adaptation automatique des ressources de la couche IaaS. La solution repose sur la définition de règles de type "si événement alors action" (cf. Code 3.2). La partie "événement" correspond à une règle *CEP* ou *event pattern*. La partie "action" correspond au plan de reconfiguration à mettre en place face à cet événement. Les événements et les plans d'action définis peuvent être plus ou moins complexes. Les actions d'adaptation considérées concernent le dimensionnement horizontal, le dimensionnement vertical et la migration.

Dans leur travail, Saleh et al proposent un *framework* permettant aux administrateurs Cloud de spécifier de telles règles de dimensionnement automatique. Le *framework* propose une interface graphique offrant une configuration "aisée" du dimensionnement automatique. En ce sens, il s'agit d'un service de dimensionnement automatique reposant sur *CEP*, qui contrairement aux solutions industrielles actuelles, permet à l'administrateur de définir finement les situations pour lesquelles le système nécessite une adaptation.

```
ON PATTERN (complex_event_expression)
DO ACTION (action_1, action_2, ..., action_n)
```

Code 3.2 – Template de règle d'adaptation proposé par [SGB+13].

Sharma et al [SSS13] se sont intéressés au contrôle de l'élasticité dans le but de simplifier la configuration et la gestion de celle-ci dans le contexte de Cloud privés. L'approche revient à rendre le contrôleur d'élasticité lui-même élastique (i.e. *adaptative control plane*). À l'instar du dimensionnement automatique des applications SaaS destinées aux utilisateurs finaux du Cloud, ce travail vise à ajuster la quantité de ressource nécessaire à la gestion de l'élasticité elle-même.

La solution repose sur un modèle de régression logistique (i.e. *logistic regression model*) pour estimer la capacité de ressource nécessaire en vue de respecter les SLO face à une certaine charge de travail. L'avantage de cette technique est que le temps d'apprentissage est relativement faible pour modéliser le comportement du service (i.e. volume de données raisonnable). De plus, les auteurs présentent deux méthodes pour le dimensionnement. Une méthode dite réactive qui vise à détecter la violation de SLO (i.e. valeur seuil sur une métrique) et une méthode pro-active qui tend à estimer la demande future (i.e. analyse des séries chronologiques) pour prévoir à l'avance les violations de SLO. Les auteurs ont implémenté deux services élastiques pour la gestion de l'élasticité des Cloud privés reposant sur *OpenStack* [ope15b]. Le premier est un service de "surveillance" (i.e. *monitoring control plan service*) s'appuyant sur *Ganglia* [gan15] tandis que le second est un service de "message" (i.e. *messaging control plan service*) implémenté par un système de files d'attentes (i.e. *message queuing system* [Vin06]) permettant la communication entre les différents services d'élasticité (e.g. surveillance, allocation de ressources, etc.). Les résultats des expériences montrent que l'approche permet de dimensionner convenablement les services d'élasticité pour des Cloud privés de différentes tailles.

Copil et al [CMTD13a] se sont intéressés au contrôle de l'élasticité multi-niveau et multi-dimension. L'aspect *multi-niveau* correspond au fait que les auteurs adressent différents types de ressources à granularité variable (e.g. cluster, VM, code, etc.) tandis que le *multi-dimension* fait référence aux dimensions de coût, de qualité et de ressource évoquées précédemment [MCTD13]. La solution suit une approche réactive et s'appuie sur le langage *SYBL* [CMTD13b], développé par les mêmes auteurs dans un travail précédent. *SYBL* est un langage dédié à l'élasticité permettant de définir des exigences en termes d'élasticité (cf. sous-section 3.4.4). Le but est de contrôler l'élasticité à différents niveaux en considérant l'observation et l'adaptation de manière multi-couche (i.e. différents types de métriques, différentes actions d'adaptation). Pour ce faire, le système proposé repose sur un modèle de ressources permettant de générer un graphe de dépendance à l'exécution. Ce graphe représente l'état courant du système à l'exécution (i.e. ressources actuelles, valeurs des métriques, etc.). L'objectif est de trouver un plan de reconfiguration satisfaisant un maximum de contraintes d'élasticité (i.e. définies sous forme de directives *SYBL*), idéalement toutes, avec un nombre minimum d'action d'adaptation. Copil et al indiquent que le fait de considérer l'élasticité de manière multi-couche et multi-dimension peut permettre aux fournisseurs de Cloud de "vendre" l'élasticité sous forme de service (i.e. *elasticity as a service*).

Jamshidi et al [JAP14] se sont intéressés au dimensionnement automatique de ressources. L'objectif de ce travail est de rendre le système capable de s'adapter à l'environnement dynamique incertain dans lequel il évolue (e.g. pic de charge inattendu) en fournissant un niveau de service de performance correcte aux utilisateurs tout en augmentant le profit du fournisseur de service par la réduction des coûts d'infrastructure. La solution proposée est un contrôleur d'élasticité hybride, appelé *RobusT2Scale* (cf. Figure 3.25) regroupant les tâches *A* et *P* de *MAPE-K*. Ce contrôleur s'appuie sur la logique floue [KML99] (i.e. réactif) permettant de prendre en charge de l'incertitude, ainsi que sur l'analyse des séries chronologiques (i.e. pro-actif) pour spécifier de manière qualitative les règles à base de seuils. L'approche de Jamshidi et al pour constituer la base de connaissance floue est intéressante. Les règles de dimensionnement automatique sont définies en s'appuyant sur l'expertise des opérateurs humains (e.g. architectes, administrateurs). Il s'agit de collecter leur savoir sous forme de questionnaire à choix multiples. Dans ce travail [JAP14], un groupe de 10 experts Cloud est soumis à un questionnaire contenant des questions du type : "Si la charge de travail courante est élevée et que les temps de réponse observés sont bas, quelle action de reconfiguration entreprendriez-vous ?". Ainsi, plusieurs choix sont possibles : ajouter/retirer une (ou deux) instances(s) ou ne rien faire (cf. Figure 3.26a). De même, les experts sont interrogés sur leur définition d'une "charge de travail élevée" ou encore de "temps de réponse bas", en positionnant des intervalles compris entre 0 et 100 sur différentes classes de charge de travail ou de temps de réponse (cf. Figure 3.26b). Les différentes réponses sont ensuite analysées pour constituer les règles.

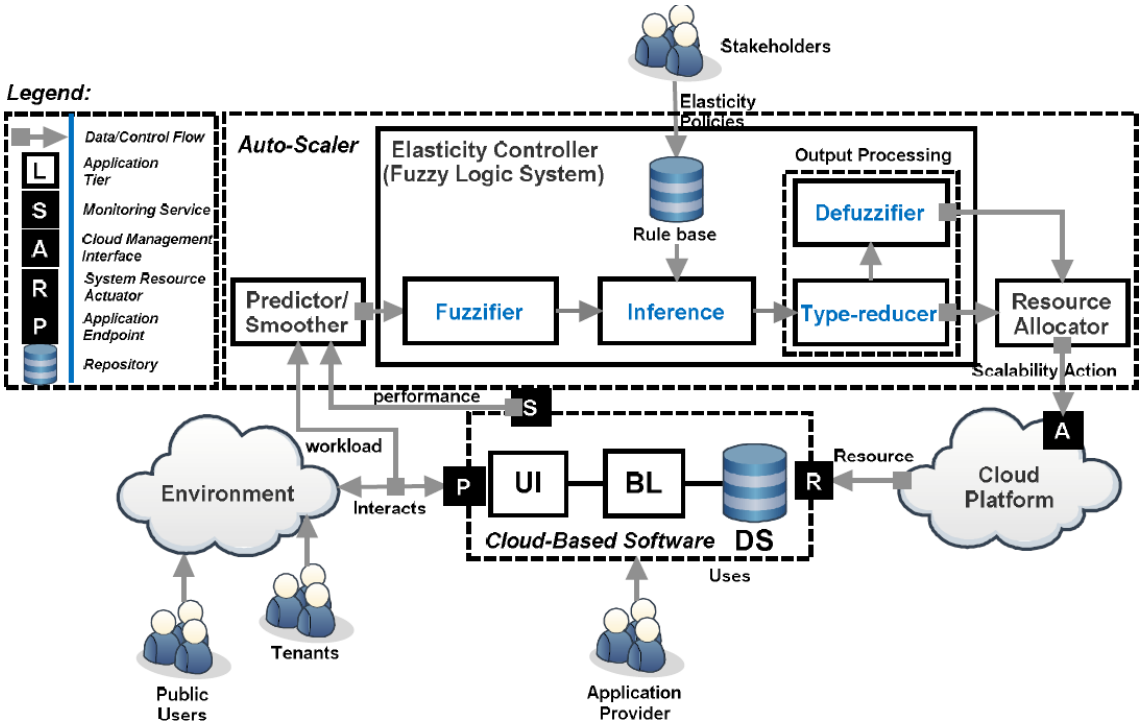


FIGURE 3.25 – Architecture de *RobusT2Scale* proposé par [JAP14].

Rule (l)	Antecedents		Consequent					c^l_{avg}
	Workload	Response-time	-2	-1	0	1	2	
1	Very low	Instantaneous	7	2	1	0	0	-1.6
2	Very low	Fast	5	4	1	0	0	-1.4
3	Very low	Medium	0	2	6	2	0	0
4	Very low	Slow	0	0	4	6	0	0.6
5	Very low	Very slow	0	0	0	6	4	1.4
6	Low	Instantaneous	5	3	2	0	0	-1.3
7	Low	Fast	2	7	1	0	0	-1.1
8	Low	Medium	0	1	5	3	1	0.4
9	Low	Slow	0	0	1	8	1	1
10	Low	Very slow	0	0	0	4	6	1.6
11	Medium	Instantaneous	6	4	0	0	0	-1.6
12	Medium	Fast	2	5	3	0	0	-0.9
13	Medium	Medium	0	0	5	4	1	0.6
14	Medium	Slow	0	0	1	7	2	1.1
15	Medium	Very slow	0	0	1	3	6	1.5
16	High	Instantaneous	8	2	0	0	0	-1.8
17	High	Fast	4	6	0	0	0	-1.4
18	High	Medium	0	1	5	3	1	0.4
19	High	Slow	0	0	1	7	2	1.1
20	High	Very slow	0	0	0	6	4	1.4
21	Very high	Instantaneous	9	1	0	0	0	-1.9
22	Very high	Fast	3	6	1	0	0	-1.2
23	Very high	Medium	0	1	4	4	1	0.5
24	Very high	Slow	0	0	1	8	1	1
25	Very high	Very slow	0	0	0	4	6	1.6

(a)

Linguistic		Means		Standard Deviations	
		Start (a)	End (b)	Start (σ_a)	End (σ_b)
Workload	Very low	0	27	0	8.23
	Low	22	41.5	7.15	7.09
	Medium	36.5	64	5.80	3.94
	High	61	82.5	4.59	6.77
	Very high	78	100	6.32	0
Response-time	Instantaneous	0	7.2	0	5.20
	Fast	6.1	20	4.07	5.27
	Medium	18.2	41.5	5.59	8.51
	Slow	38.5	63.5	7.09	9.44
	Very slow	60	100	7.82	0

(b)

FIGURE 3.26 – Questionnaire adressé aux experts Cloud : *RobusT2Scale* [JAP14].

3.4.4 Langages pour la gestion de l'élasticité

SRL : A Scalability Rule Language for Multi-cloud Environments

Kritikos et al ont proposé *SRL* (*Scalability Rule Language*) [KDR14] [DKR14], un langage de règles de dimensionnement permettant de spécifier des scénarii complexes d'adaptation dans des environnements multi-nuage. Le langage peut être défini de langage dédié (i.e. *Domain Specific Language - DSL*) au domaine de l'élasticité. Les auteurs partent du constat que les services de dimensionnement automatique actuels supportent uniquement des scenarii d'adaptation simples faisant intervenir qu'un seul type d'action de dimensionnement dans la définition des règles d'élasticité, comme le *Scale Out* (e.g. démarrer une nouvelle VM). De plus, la partie "condition" des règles de dimensionnement automatique ne permet d'adresser qu'un ensemble restreint de métriques portant généralement sur la couche IaaS (e.g. consommation CPU). Cela s'avère généralement limitant pour la définition de règles d'élasticité pertinentes qui dépendent de métriques liées à l'application elle-même.

SRL est inspiré du langage *OWL-Q* [KP06] pour la spécification des métriques de QoS et du langage *EPL* (*Esper Processing Language* [esp15b]) pour la définition des événements sous forme de règles *CEP* (*Complex Event Processing*). Le langage permet de définir des règles connues sous le nom de règles *ECA* (*Event-Condition-Action*). Chaque composante de ces règles, à savoir les événements, les conditions et les actions peuvent être à la fois simples ou complexes (i.e. composites), contrairement à ce que permettent les solutions industrielles actuelles (e.g. *Amazon Auto Scaling* [EC215a], *Microsoft Autoscaling Application Block* [Mic15], etc.). Les événements considérés sont divers et variés et peuvent concerner différentes métriques provenant de différentes couches (i.e. IaaS, PaaS, SaaS). Ces métriques, fonctionnelles ou non, peuvent être agrégées dans le temps selon différents niveaux de granularité (e.g. seconde, minute, jour, etc.).

Les actions de dimensionnement considérées concernent le dimensionnement horizontal (i.e. *Scale Out/In*) et vertical (i.e. *Scale Up/Down*). Le résultat d'une action peut aussi être la création d'un nouvel événement. L'application des règles de dimensionnement est aussi régulée par des politiques (i.e. *Scaling Policy*) indiquant par exemple la capacité maximum de mémoire ou de stockage dans le cas du dimensionnement vertical ou encore le nombre maximum/minimum d'instance pour le dimensionnement horizontal. Les auteurs ont aussi développé un outillage autour du langage basé sur l'environnement de développement *Eclipse* [ecl15a]. Cette outillage permet de s'assurer que les règles de dimensionnement automatique définies par l'administrateur Cloud sont correctes en les validant syntaxiquement et sémantiquement. Pour ce faire, la solution s'appuie sur le projet *Eclipse Model Development Tools* (*MDT* - [mdt15]). Ce projet inclut *Eclipse OCL* [ecl15b] qui offre une implémentation du langage déclaratif *OCL* (*Object Constraint Language* [ocl15]).

SYBL : An extensible language for controlling elasticity in cloud applications

Copil et al [CMTD13b] ont présenté *SYBL* (*Simple Yet Beautiful Language*), un langage pour la spécification de l'élasticité dans le Cloud sous forme de directives. Le langage est associé à un modèle *runtime* chargé de contrôler l'élasticité à l'exécution en considérant les spécifications d'élasticité définies avec *SYBL*. Le langage permet de spécifier précisément l'observation du système à travers la surveillance de métriques (e.g. temps de réponse). Ces métriques peuvent concerner différents types de ressources Cloud (e.g. application globale, composants applicatifs, etc.). De plus, elles peuvent concerner différentes dimensions d'élasticité (i.e. *cost*, *resource*, *quality*) comme définies par les auteurs (cf. Figure 3.27).

SYBL offre aussi la possibilité de définir des contraintes sur le système. Il s'agit de conditions à satisfaire pour la mise en place d'action de reconfiguration. Néanmoins, contrairement aux règles de dimensionnement automatique usuelles où il est parfois nécessaire de réécrire plusieurs fois la même condition, ces contraintes sont réutilisables et composables en vue de les associer à différentes stratégies d'élasticité.

Une stratégie d'élasticité est définie par les auteurs comme une recette à suivre dans un cas particulier. Il s'agit concrètement d'un couple "condition-action" où la partie "condition" peut être une composition de différentes contraintes définies au préalable. La partie "action" concerne les actions d'adaptation habituelles de la couche IaaS (e.g. *Scale Out/In*) mais aussi d'autres actions possibles (e.g. changer le prix d'un service). De plus, le langage permet de définir des priorités entre stratégies dans le cas où la partie "condition" d'une stratégie "recouvre" celle d'une seconde stratégie (cf. Code 3.3). *SYBL* permet donc de définir des directives pour le contrôle de l'élasticité. Ces directives peuvent représenter des objectifs de haut-niveau définis par exemple par le fournisseur de service (cf. Code 3.3) mais aussi des objectifs plus techniques en termes d'utilisation de ressource ou de dimensionnement de l'infrastructure pouvant être définis par des développeurs ou des administrateurs Cloud (cf. Code 3.4). Le langage permet ainsi à différents profils d'utilisateurs (e.g. fournisseurs de service, développeurs, utilisateurs, etc.) de spécifier le comportement qu'ils attendent du système en termes d'élasticité.

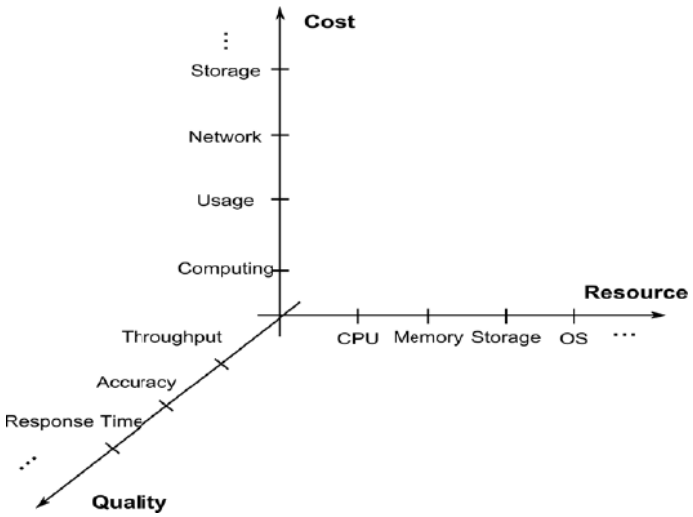


FIGURE 3.27 – Dimensions d'élasticité et propriétés associées selon [CMTD13b].

```
#SYBL.ApplicationLevel
Str1: STRATEGY CASE availability > 98% setPrice(100)
Str2: STRATEGY CASE availability > 99% setPrice(300)
Priority(Str1) < Priority(Str2)
```

Code 3.3 – Exemple de directives *SYBL* définies par le fournisseur de service

```
#SYBL.ApplicationLevel
Mon1: MONITORING rt = Quality.responseTime
Cons1: CONSTRAINT rt < 2 ms. when nbOfUsers < 1000
Cons2: CONSTRAINT rt < 4 ms. when nbOfUsers < 10000
Cons3: CONSTRAINT totalCost < 800 Euro
Str1: STRATEGY CASE Violated(Cons1) OR Violated(Cons2): ScaleOut
Priority(Cons1) = 3, Priority(Cons2) = 5
#SYBL.ComponentLevel
ComponentID = Component3; ComponentName = Data Engine
Cons4: CONSTRAINT totalCost < 600 Euro
#SYBL.ComponentLevel
ComponentID = Component2 ComponentName = Computing Engine
Cons5: CONSTRAINT cpuUsage < 80%
#SYBL.ProgrammingLevel
Cons6: CONSTRAINT dataAccuracy > 90% AND cost < 400
```

Code 3.4 – Exemple de directives *SYBL* définies par un développeur.

3.5 Conclusion

Dans le Cloud computing, l'élasticité correspond au degré selon lequel un système est capable de s'adapter à une demande variable en ajoutant ou retirant des ressources de manière automatique afin que la quantité de ressource disponible soit en permanence au plus proche de la demande [HKR13]. On distingue deux caractéristiques essentielles attendues d'un processus de gestion de l'élasticité. Il s'agit de la *précision* du dimensionnement et de la *réactivité* du dimensionnement [WHGK14]. Il est nécessaire d'adresser ces deux problématiques afin d'éviter les aspects négatifs du sur-dimensionnement et du sous-dimensionnement.

Les travaux existants, portant sur le concept d'élasticité dans le Cloud, tendent à considérer ces deux aspects essentiels que sont la *réactivité* et la *précision*. Cependant, les solutions proposées n'adressent que rarement les deux problématiques en même temps. En ce qui concerne la *réactivité* d'adaptation, certaines solutions reposent sur l'utilisation de la méthode de dimensionnement vertical pour être plus réactif face aux changements (i.e. par rapport au dimensionnement horizontal), tandis que d'autres travaux s'appuient sur une approche pro-active, reposant sur des modèles prédictifs, visant à anticiper le besoin de ressource en vue d'appliquer les reconfigurations en amont et ainsi couvrir le temps d'initiation non-négligeable des ressources IaaS (i.e. dimensionnement horizontal). La précision du dimensionnement est quant à elle adressée par des travaux utilisant notamment l'analyse des séries chronologiques ou encore l'apprentissage par renforcement pour identifier la juste quantité de ressource à mettre en place pour une certaine charge de travail en vue de satisfaire certaines contraintes en termes de QoS ou performance (i.e. SLA).

Il faut préciser que les modèles prédictifs et les différentes théories évoquées ci-dessus sont majoritairement réservées au monde scientifique. En effet, les solutions industrielles préfèrent une approche réactive reposant sur les règles à base de seuil. Cela s'explique du fait que ce type de solution est plus simple à mettre en œuvre et plus facilement compréhensible par une communauté non scientifique qui peine à appréhender des théories trop éloignées du monde industriel. De plus, bien que nécessitant une expertise du domaine et une connaissance accrue de l'application pour la définition des seuils, l'approche réactive est moins sujette à l'évolution de l'application (courant dans le monde industriel), qui s'avère être un cas rédhibitoire pour l'utilisation de modèles prédictifs basés sur l'apprentissage.

Le Tableau 3.1, récapitulatif de l'état de l'art basé sur les critères énoncés dans la section 3.1, montre que la grande majorité des travaux s'intéressent uniquement à la couche IaaS en considérant des actions d'adaptation sur cette couche seulement. De plus, la plupart des solutions ne considère qu'une unique méthode de dimensionnement (e.g. horizontal ou vertical). Une autre remarque est que la majorité des travaux, dans leurs intentions, ne considèrent qu'un ou deux aspect(s) de l'élasticité parmi lesquels figurent le coût (i.e. budget), la QoS (e.g. temps de réponse, disponibilité) et les ressources (e.g. énergie). En ce sens, les travaux existants ne traitent qu'un sous ensemble des compromis induits par l'élasticité [JHJ+10]. Plusieurs travaux se sont penchés sur l'adaptation des ressources logicielles dans de nombreux domaines (e.g. lignes de produits logiciels, architecture orientée services ou composants [BCL+06], etc.), néanmoins, on ne dénombre que peu de travaux portant sur l'adaptation des couches hautes du Cloud (i.e. majoritairement le *middleware* [MRS11], couche PaaS). En effet, très peu de solutions se sont intéressées à adapter la couche SaaS ou encore à considérer l'élasticité de manière multi-couche [Dau13] aussi bien pour l'observation (i.e. métriques de différents niveaux), la prise de décision ou l'adaptation elle-même (i.e. actions). Nous considérons que l'adaptation des applications SaaS dans un environnement Cloud, et plus précisément leur paramétrisation, s'apparente à un nouveau type d'élasticité que nous appelons *élasticité logicielle*. L'*élasticité logicielle* revient à fournir des applications offrant différents niveaux de service, plus ou moins consommateurs de ressources (i.e. CPU, RAM, énergie, etc.) et servant de variable d'ajustement pour rendre le système capable de s'adapter rapidement au contexte dynamique dans lequel il évolue (cf. sous-section 5.1.1).

Tableau 3.1 – Tableau récapitulatif des travaux étudiés.

Référence	Approche		Observation		Décision					Adaptation		Intention	SLA
	R	*P*	*IaaS*	*P/S aaS*	*TBR*	*RL*	*QT*	*CT*	*TSA*	*IaaS*	*P/S aaS*		
[LBCP09]	X		X	X	X			X		H		performance	-
[FGG10]	X		X	X						H		haute disponibilité	temps de réponse
[GSLI11]	X		X		X			X		H		-	-
[MBS11]	X		X		X					V,H,M		énergie	utilisation des ressources
[HGGG12]	X		X	X	X					H,V		performance	temps de réponse
[MBS12]	X		X		X					V		performance,coût,énergie	cpu,mémoire,disque,réseau
[MH13]	X		X	X						H,V		performance,coût	budget
[CS13]	X		X	X	X		X			H		performance	temps de réponse
[HGG+14]	X		X	X			X			H		coût	temps de réponse,budget
[XZF+07]		X	X	X				X		V		performance,coût	disponibilité
[CHL+08]		X	X	X					X	H		énergie	disponibilité,conso. énergétique
[RBX+09]		X	X	X		X				V		performance	débit,temps de réponse
[PHS+09]		X	X	X				X		V		performance	temps de réponse
[GGW10]		X	X						X	V		performance	temps de réponse
[JHJ+10]		X	X	X			X	X	X	V,H,M		performance,énergie,coût	-
[RDG11]		X		X					X	H		performance,coût	temps de réponse
[SSSS11]		X	X	X						H,M		performance,coût	temps de réponse
[GSL+12]		X	X		X			X		H		politiques haut niveau	temps de réponse
[VNM+12]		X	X	X		X				H,V		performance	temps de réponse
[FLWC12]		X	X	X					X	H,V,M		précision prédiction	-
[IKLL12]		X	X						X	H		précision prédiction	-
[DGVV12]		X	X	X					X	H,V		performance,coût	temps de réponse
[BHD13]		X	X			X				H		performance,coût	temps de réponse,budget
[JLZL13]		X		X			X		X	H		performance,coût	temps de réponse
[USC+08]	X	X	X	X	X		X			H		performance	temps de réponse
[SSGW11]	X	X	X						X	V,M		performance,énergie	temps de réponse,taux avancement
[IDCJ11]	X	X	X	X	X				X	H		performance	temps de réponse
[AETE12]	X	X	X	X			X	X		H		performance	disponibilité
[LRME13]	X	X	X	X	X				X	H		performance	cpu
[JAP14]	X	X		X	X				X	H		performance,profit	temps de réponse
[KMÅHR14]	X			X				X			P	disponibilité,QoF	-
[NKHR14]	X	X	X	X				X			P	performance,coût,QoF	temps de réponse,QoF
[GCH+04]*	X		X	X	X					H	A	performance	temps de réponse
[DTM11][MDT13]		X	X	X				X		V	P	performance	temps de réponse
[ZA10]		X	X	X		X		X		V	P	performance,coût	budget,réactivité
[ZHLM10]	X		X	X	X			X		V	P	performance,efficience	-
[XRB12]		X	X	X		X				V	P	performance	temps de réponse
[BRX13]		X	X	X		X				V	P	performance	temps de réponse,débit
[CGS06]	X		X	X	X					H	P	performance,coût,QoF	budget,temps de réponse
[GSC09]	X		X	X	X					H	A	performance,coût,QoF	budget,temps de réponse
[CGS09]	X		X	X	X					H	A	performance,coût,QoF	budget,temps de réponse
[AB14]*	X		*	*						*	*	réutilisabilité	temps de réponse
[KAMV12]	X	X	X	X						*	*	outillage élasticité	*
[CMTD13a]	X		X	X	X					*	*	performance,coût	-
[MWM+14]			*	*						*	*	coordination multicouche	-
[MCTD13]	X		X	X	X							surveillance	-
[SGB+13]	X		X		X					H,V,M		outillage élasticité	-
[CMTD13b]	X		X	X	X					*	*	outillage élasticité	-
[SSS13]	X	X	X	X	X				X	H		contrôleur élastique	temps de réponse,utilisation ressources
[KDR14]	X		X	X	X					H,V		outillage élasticité	-
[TDC+14]	X		X	X	X					*	*	contrôle élasticité	-

Notre intuition est que l'*élasticité logicielle* offre une capacité d'adaptation plus rapide (i.e. *réactivité*) et à granularité plus fine (i.e. *précision*) que celle de l'infrastructure. Cette souplesse dans la gestion des ressources permettrait selon nous de se prémunir de l'utilisation des modèles prédictifs complexes de l'approche pro-active. De même, l'*élasticité logicielle*, de par la souplesse de dimensionnement qu'elle procure, peut s'avérer bénéfique dans le cadre d'une approche réactive implémentée à l'aide de règles à base de seuils. En effet, la granularité fine des actions d'adaptation permet de paramétrer plus finement les seuils des règles plutôt que d'avoir recours à la pratique usuelle qui vise à anticiper le temps non-négligeable des actions d'adaptation (e.g. *Scale Out*) en admettant des seuils plus lâches (e.g. CPU > 70 % au lieu de 80%) ce qui peut s'avérer néfaste et entrainer un sur-dimensionnement en allumant des ressources qui ne seront pas sollicitées (i.e. coûts inutiles).

Dans cette thèse, nous proposons de mettre à profit les avantages offerts par l'*élasticité logicielle* en vue d'améliorer la capacité d'adaptation du Cloud dans sa globalité. L'*élasticité logicielle* n'a pas pour vocation de remplacer l'élasticité de l'infrastructure, inhérente au Cloud, mais plutôt de pallier ses manques en termes de réactivité, de flexibilité (i.e. granularité) et d'empreinte énergétique. La force de notre proposition réside dans la complémentarité entre l'élasticité de l'infrastructure et l'élasticité du logiciel. En effet, en considérant l'élasticité de manière transverse (depuis les machines physiques jusqu'à l'utilisateur final des applications SaaS), il devient possible d'agir à tous les niveaux en vue d'optimiser l'utilisation des ressources Cloud au sens large.

Pour atteindre l'objectif fixé, il est indispensable de modéliser cette nouvelle dimension d'adaptation qu'est l'*élasticité logicielle*. De plus, le fait d'étendre le concept d'élasticité aux couches hautes du Cloud implique une élasticité multi-couche qui doit être modélisée et outillée en vue de simplifier sa gestion et ainsi bénéficier de la synergie offerte par les différents types d'élasticité. Le fait de considérer l'élasticité de manière transverse soulève un problème en termes de complexité d'administration qui a été abordé dans la section 3.4. Il devient alors nécessaire d'outiller le processus de gestion de l'élasticité à travers des langages et outils adaptés à ce domaine afin d'en faciliter la tâche de configuration. De plus, il convient de fournir aux administrateurs Cloud de nouveaux moyens pour configurer de manière intelligible les compromis induits par cette élasticité multi-couche et ainsi industrialiser le processus de gestion de celle-ci.

Contributions

4

Élasticité multi-couche : motivation par l'exemple

Dans le cadre de nos travaux portant sur l'élasticité dans le Cloud, nous avons entrepris, dès la première année de thèse, de valider nos intuitions quant au fait d'étendre les capacités d'adaptation du système en adressant les ressources informatiques des couches hautes du Nuage. Ainsi, un projet autour de l'*élasticité logicielle* (i.e. SaaS), faisant office de preuve de concept (*POC - Proof of Concept*), a rapidement été lancé à l'entreprise en vue de confirmer la pertinence de la proposition de la thèse.

Contents

4.1 *TUBA* : une preuve de concept menée à l'entreprise

4.1.1 Objectif initial

L'objectif du projet était de développer un cas concret, proche des activités de *Sigma Informatique*, d'une application mobile de consultation des transports en commun, basée sur les informations mises à disposition librement par *Nantes Métropole* (i.e. *Open data* [ope15a]). Cette application fut l'opportunité pour *Sigma* d'améliorer son savoir-faire (i.e. mobilité, *open data*, etc.) ainsi que de "maltraiter" l'architecture de l'application (i.e. simulations de pannes, montées en charge excessives, etc.) pour vérifier son aptitude à continuer de rendre le service attendu dans des situations critiques, tout en minimisant son empreinte énergétique. Ce projet, faisant office de *Proof-of-Concept* de l'utilisation conjointe de l'élasticité des ressources IaaS et SaaS, a abouti à un livrable s'intitulant *TUBA* (Test d'Utilisation de Boucle Autonomique). Ce démonstrateur, visant à comparer le comportement d'une architecture Cloud face à différents scénarii de montée en charge a donné lieu à de nombreuses communications et retours positifs.

4.1.2 Scénario de motivation concret

Le choix d'une application mobile de consultation des transports en commun, en tant que cas d'utilisation, n'a pas été effectué par hasard. En effet, le 18 janvier 2013, suite à un épisode neigeux important ayant eu lieu à Nantes pendant la nuit, de très nombreux utilisateurs du réseau de transport en commun de l'agglomération nantaise ont sollicité l'application mobile et le site internet de la compagnie de transport afin de connaître l'état du trafic. Cette montée en charge importante, soudaine et imprévisible a entrainé l'interruption totale du service (i.e. application mobile et page web inaccessible). Le cas d'utilisation évoqué ci-dessus traduit un manque d'élasticité du système qui n'a pas été en mesure de passer à l'échelle. Ainsi, les utilisateurs n'ont pas eu accès au service au moment où ils en avaient vraiment besoin. De plus, il s'agissait potentiellement, pour certains utilisateurs, de l'unique fois dans l'année où ils ont sollicité le service ce qui leur donne une image négative du fournisseur bien que le service ait été opérationnel le restant de l'année.

À partir de ce cas d'utilisation, nous avons examiné les raisons de cette interruption de service ainsi que les solutions qui pouvaient être mises en œuvre par le fournisseur de service pour éviter ce genre d'incident. Nous avons ainsi considéré que l'infrastructure et l'application étaient toutes deux fautives vis-à-vis de cet échec. L'infrastructure n'a pas été en mesure de passer à l'échelle ce qui s'est traduit par l'interruption complète du service. Les raisons peuvent être multiples. Par exemple, il se peut qu'aucun mécanisme d'élasticité n'ait été prévu par le SI ou que l'infrastructure informatique soit modeste et de taille limitée. L'application, quant à elle, offrait ce matin là la possibilité de télécharger le plan général complet du réseau de l'agglomération nantaise (i.e. volume supérieur à 5 Mo) ou encore le détail complet des arrêts et horaires d'une ligne de bus selon toutes les périodes de l'année. Il aurait été préférable de désactiver de telles fonctionnalités en vue de se concentrer sur l'essentiel, à savoir informer les utilisateurs des lignes de bus/tramway opérationnelles ou non. Cela aurait permis de soulager les ressources sous-jacentes (i.e. CPU, mémoire, réseau, etc.) en vue d'absorber ce pic de charge imprévisible tout en continuant de fournir le service, bien que dégradé.

Nous avons ainsi entrepris de développer une application mobile de consultation des transports en commun, capable de surmonter des événements imprédictibles bien que récurrents (e.g. montée en charge, pannes, etc.). Il s'agit concrètement d'une application Web suivant une architecture Cloud, en mesure d'ajuster ses ressources sur la couche IaaS (i.e. VMs) et SaaS (i.e. composants logiciels), ce qui la rend plus à même de pouvoir s'adapter à l'environnement dynamique dans lequel elle évolue.

4.1.3 Prototypage d'une application élastique

Architecture de l'application

L'application développée peut être qualifiée d'application *client-serveur multi-tier*. L'architecture logique du système est constituée de trois tiers (cf. Figure 4.1) :

— le tier présentation, ou client, qui va solliciter le service par le biais d'une interface utilisateur (e.g. application du terminal mobile ou navigateur web) ;
— le tier métier, ou serveur d'application, qui est chargé de la mise en œuvre de l'ensemble des règles de gestion et de la logique applicative. Dans notre cas, ce tier est composé de différents composants tels que *Apache Tomcat* [tom15] en tant que serveur *HTTP*, *OpenTripPlanner* [otp15] pour la partie calcul d'itinéraires, etc.
— le tier d'accès aux données, ou serveur de données, qui a pour but de conserver les données et de restituer celles-ci au serveur d'application si besoin.

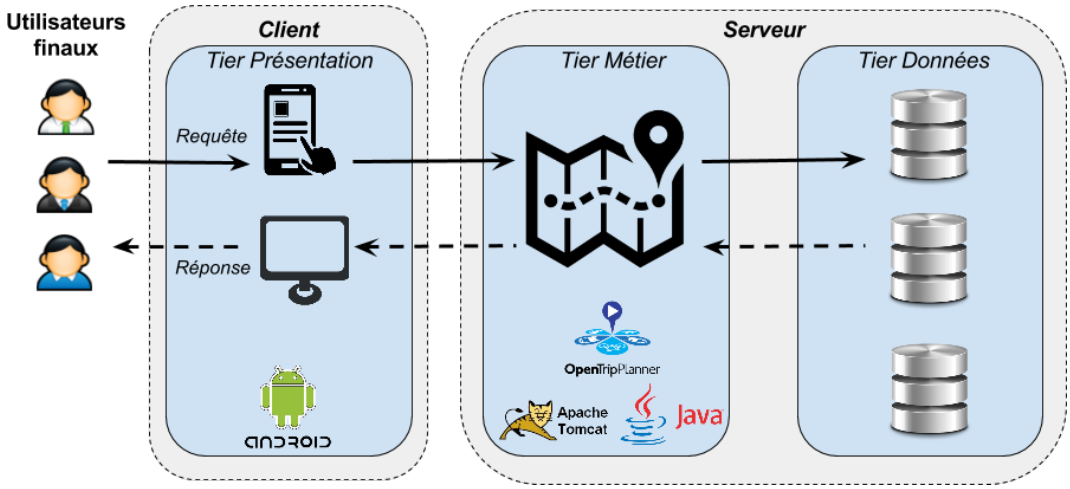


FIGURE 4.1 – Application de consultation des transports en commun : architecture 3-tier.

L'application développée par nos soins offre un service de calcul d'itinéraires des transports publics de l'agglomération nantaise s'appuyant sur les informations mises librement à disposition par *Nantes Métropole* [ope15a] (e.g. positions GPS des arrêts, horaires de passage, etc.). Il s'agit d'un service de type *Google Maps* [goo15a] permettant à l'utilisateur de connaître l'itinéraire pour se rendre d'un point A à un point B en lui indiquant quel(s) arrêt(s) ou ligne(s) de bus/tramway emprunter. Les services de calcul d'itinéraires proposent généralement plusieurs trajets possibles en réponse à une requête (e.g. 3 itinéraires différents proposés par *Google Maps*).

Dans notre cas, nous avons décidé de proposer trois modes distincts pour ce service que nous appelons $mode_1$, $mode_2$ et $mode_3$ et qui vont respectivement calculer et retourner un, deux ou trois itinéraire(s). Une de nos hypothèses de travail est que chacun des modes admet une consommation de ressource distincte. Bien que le mode retournant trois itinéraires offre une meilleure expérience utilisateur, il est aussi plus consommateur de ressource que les autres modes (e.g. CPU, réseau). On peut imaginer d'autres modes en offrant par exemple la possibilité de préciser une (ou plusieurs) escale(s) dans le trajet impliquant un calcul d'itinéraire plus ou moins complexe et consommateur de ressources.

Architecture de l'infrastructure

L'infrastructure repose sur une architecture de type Cloud où les composants applicatifs sont déployés sur des machines virtuelles (VM). La solution s'appuie sur l'outil *Apache CloudStack* [clo15a] pour le déploiement et la gestion des VMs.

Les deux tiers de la partie serveur (cf. Figure 4.1) sont déployés sous forme de VMs. Chacun de ces tiers (i.e. métier et données) se voit attribuer un type de VM, ou plus précisément une image de VM configurée en fonction des composants constituants le tier (e.g. OS, *Java*, *Apache Tomcat*, *OpenTripPlanner* installés et paramétrés pour le *tier Métier*). Une image de VM peut être associée à une classe *Java* permettant d'instancier des objets du même type. Ainsi, il va être possible de dupliquer/répliquer les VMs d'un tier en fonction de sa charge de travail (i.e. dimensionnement horizontal). Chaque tier se voit aussi attribuer un répartiteur de charge (niveau 4 - *TCP*) permettant de partager équitablement les requêtes sur les différentes VMs qui constituent le tier. Le mécanisme de répartition de charge, couplé aux images distinctes de VMs, permet dans notre cas de passer à l'échelle les tiers de manière indépendante. La Figure 4.2 donne une vision générale de l'architecture de l'infrastructure de l'application mobile de consultation des transports en commun.

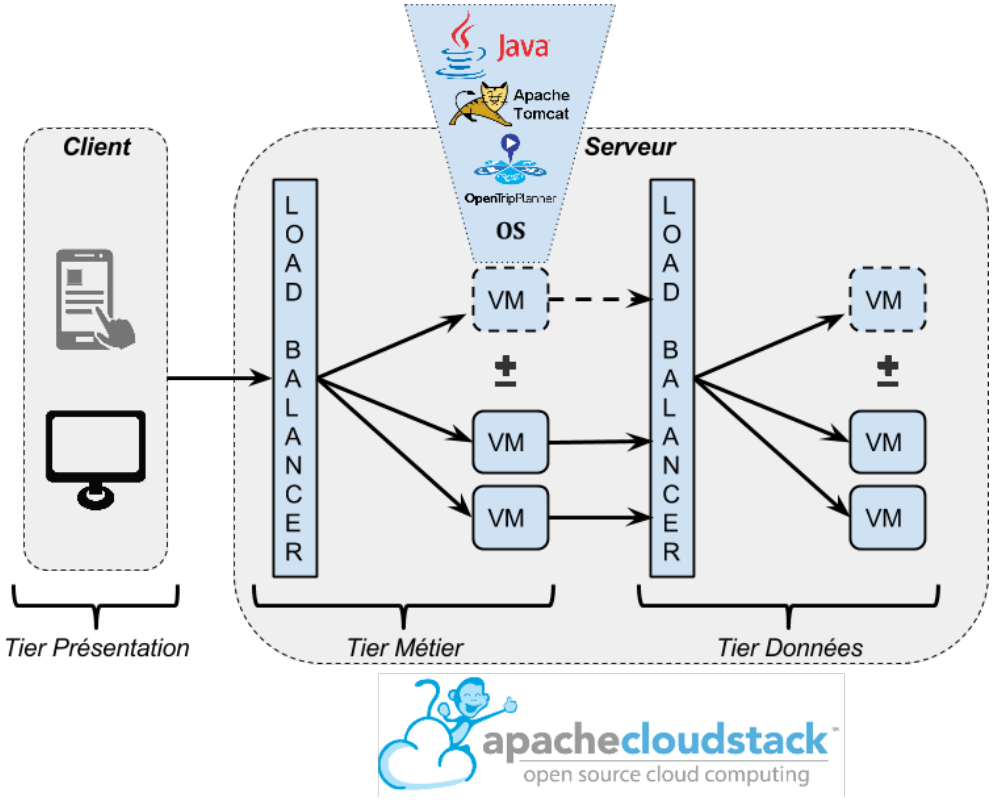


FIGURE 4.2 – Architecture de l'infrastructure : tiers, VMs et répartiteurs de charge.

Gestionnaire autonomique

Nous avons développé un gestionnaire autonomique (cf. Figure 4.3), reposant sur une architecture *MAPE-K* [KC03], permettant de gérer automatiquement le redimensionnement du système au niveau des ressources (i.e. dimensionnement horizontal) et des services (i.e. reconfiguration du service de calcul d'itinéraires). Le système administré correspond aux VMs contenant les composants de l'application. Le gestionnaire autonomique s'interface avec le système administré au travers des sondes et des actionneurs. Du point de vue technique, un bus de message asynchrone a été mis en place pour la communication entre ces deux entités (i.e. *Apache ActiveMQ* [act15]).

La tâche de surveillance *M* est chargée de collecter et éventuellement d'agréger les informations fournies par les sondes. Les sondes concernent des métriques dites systèmes telles que la consommation CPU ou mémoire des VMs (i.e. *syslog CentOS*) mais aussi des métriques pouvant être qualifiées de haut niveau telles que les temps de réponse observés, les codes statut *HTTP*, etc. (i.e. *Access Logs Tomcat*).

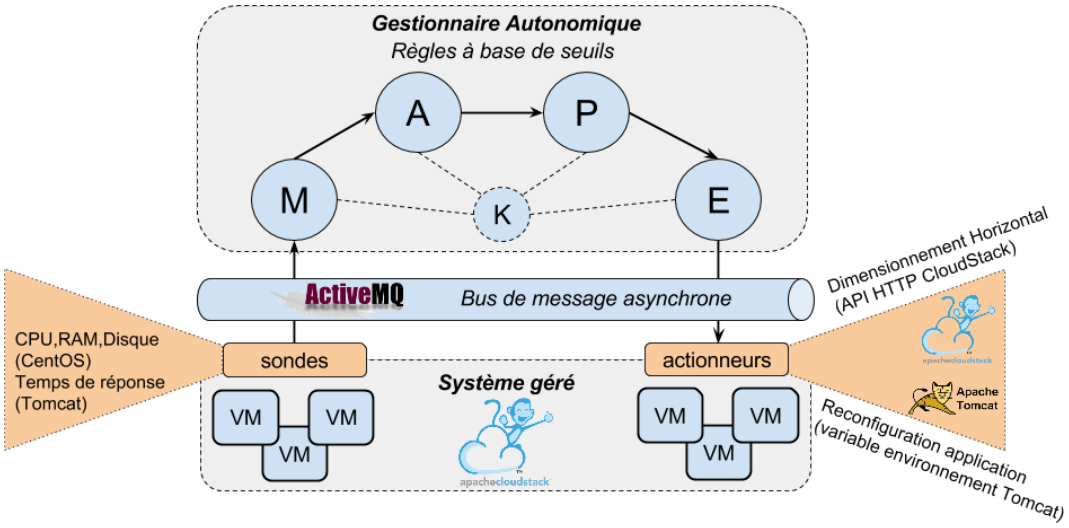


FIGURE 4.3 – Gestionnaire autonomique et système administré.

La tâche d'exécution *E* correspond à l'appel des actionneurs mis à disposition par le système administré. Les actionneurs portent à la fois sur l'infrastructure et l'application. Les actionneurs, que l'on peut qualifier de IaaS, concernent le dimensionnement horizontal. Il s'agit concrètement de l'*API HTTP* offerte par *CloudStack* permettant notamment d'ajouter ou de retirer des VMs. Les actionneurs portant sur l'application offrent la possibilité de changer le mode du service de calcul d'itinéraires en passant par exemple du mode retournant 3 itinéraires en réponse à une requête utilisateur à celui n'en retournant qu'un. D'un point de vue technique, il s'agit de changer la valeur d'une variable d'environnement *Tomcat* à l'exécution (i.e. prise en considération dynamiquement) pour spécifier le mode du service.

Les tâches d'analyse *A* et de planification *P* ont à leur charge la décision d'adaptation en s'appuyant sur les données transmises par la tâche *M*. Dans notre cas, la décision s'appuie sur des règles à base de seuils [LBMAL12]. On distingue ici deux types de règles de dimensionnement automatique. Les règles IaaS, faisant intervenir des métriques (i.e. consommation CPU, RAM) et des actionneurs relatifs à l'infrastructure (i.e. *Scale Out*, *Scale In*) et les règles SaaS, qui concernent quant à elles des métriques (i.e. temps de réponse) et des actionneurs (i.e. changement de mode du service) de haut niveau. Les règles de dimensionnement IaaS et SaaS sont ici indépendantes, c'est-à-dire qu'elles sont évaluées indépendamment et ne concernent ni les mêmes métriques, ni les mêmes actions. Ainsi, les règles SaaS, comme les règles IaaS, font intervenir des métriques et des actions propres à leur couche. Il faut préciser que la coordination entre règles n'était pas l'objectif de ce travail. En ce sens, aucun effort n'a été réalisé en vue de synchroniser les actions d'adaptation. Nous verrons dans la suite de ce chapitre (cf. section 4.2) que cette problématique a été adressée dans un travail ultérieur.

4.1.4 Impact de l'*élasticité logicielle*

Les premières expérimentations menées visaient à identifier les gains potentiels de l'*élasticité logicielle* au travers du cas d'utilisation de l'application mobile de consultation des transports en commun. Pour ce faire, nous avons considéré une infrastructure non élastique de type *Cluster* constituée de 4 VMs pour le *tier Métier* et d'une VM pour le *tier Données*. Les VMs sont dotées de deux cœurs CPU de 2 GHz, 4 Go de mémoire et reposent sur le système d'exploitation *CentOS 6.0 (64-bit)*. Le gestionnaire autonome, implémenté en *Java*, est déployé sur un poste accessible via le réseau (i.e. bus de message asynchrone implémenté avec *ActiveMQ* [act15]).

Nous avons soumis l'application à un scénario de montée en charge et nous avons réalisé trois expériences en vue de passer au banc d'essai les trois modes (i.e. $mode_1$, $mode_2$ et $mode_3$) du service de calcul d'itinéraires calculant et retournant respectivement 1, 2 ou 3 itinéraires en réponse à une requête utilisateur. Il faut préciser qu'il s'agit ici d'un service sans état ou *stateless*, c'est-à-dire que chaque requête est traitée comme une transaction indépendante sans relation avec les requêtes précédentes (i.e. pas de session utilisateur).

Nous avons eu recours à l'outil *Apache JMeter* [jme15] en tant qu'injecteur de charge pour simuler les requêtes *HTTP* clientes et stresser l'application. La Figure 4.4 donne un aperçu du scénario de montée en charge appliqué dans le cadre de ces expériences. La montée en charge est réalisée en faisant croître le nombre de *Thread*, c'est-à-dire le nombre de client accédant simultanément à l'application. Chaque *Thread* créé va exécuter une requête, puis attendre la réponse du serveur avant d'en soumettre une autre. Ce processus est répété de manière itérative par tous les clients pendant toute la durée de l'expérience (i.e. 27 min).

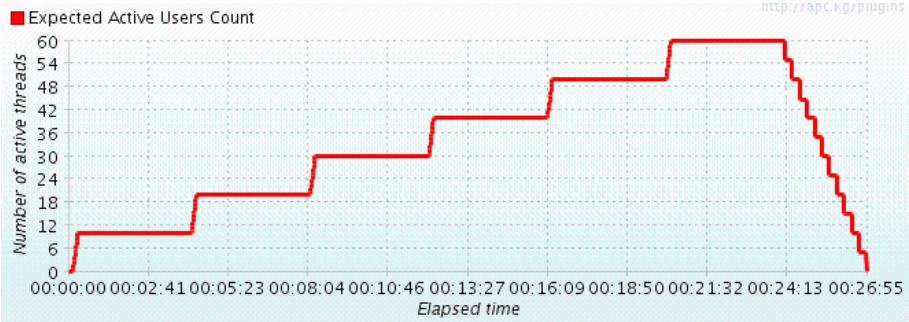


FIGURE 4.4 – Scénario de montée en charge appliqué par *JMeter*.

La Figure 4.5 correspond aux rapports consolidés obtenus par l'outil *JMeter* à la fin des expériences. Les trois Tableaux 4.5a, 4.5b et 4.5c représentent respectivement les résultats obtenus pour l'application paramétrée avec le $mode_3$, le $mode_2$ et le $mode_1$. Le tableau du rapport consolidé fourni par *JMeter* offre les informations suivantes :

- colonne 1 (*Label*) : la phase de l'expérimentation (une seule dans notre cas) ;
- colonne 2 (*#Samples*) : le nombre total de requêtes traitées pendant l'expérience ;
- colonne 3 (*Average*) : la durée moyenne de traitement d'une requête (milliseconde) ;
- colonne 4 (*Min*) : la durée minimum de traitement d'une requête (milliseconde) ;
- colonne 5 (*Max*) : la durée maximum de traitement d'une requête (milliseconde) ;
- colonne 6 (*Std. Dev.*) : l'écart type (*standard deviation*) des temps de réponse ;
- colonne 7 (*Error %*) : le pourcentage de requêtes tombées en erreur ;
- colonne 8 (*Throughput*) : le débit de sortie de l'application (requêtes/seconde) ;
- colonne 9 (*KB/sec*) : le volume de données transitant sur le réseau (ko/seconde) ;
- colonne 10 (*Avg. Bytes*) : le volume moyen de données des requêtes (octet).

Les résultats montrent que le fait de changer le mode de l'application a des répercussions sur les performances (i.e. temps de réponse) ainsi que sur l'utilisation des ressources (i.e. réseau). Il faut préciser que les colonnes 2, 3 et 8, correspondant respectivement au nombre de requêtes traitées, au temps de réponse moyen des requêtes et au débit de sortie sont étroitement liées. On constate notamment un gain de l'ordre de 21% entre le $mode_1$ et le $mode_3$ pour ces différentes métriques. En ce qui concerne l'utilisation des ressources, les résultats montrent un gain non négligeable en termes de bande passante. Comme nous le verrons par la suite, le réseau n'est pas la seule ressource concernée par cette tendance. Si l'on ramène cela à la durée totale de l'expérience, le $mode_1$, bien que permettant d'exécuter 21% de requête supplémentaire, permet de soulager le réseau de l'ordre de 47% par rapport au $mode_3$. De plus, le taux de requêtes en erreur observé pour le $mode_1$ est plus faible que les autres modes ce qui se traduit par une meilleure disponibilité du service (i.e. *availability*), qui s'avère être un critère essentiel de QoS.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Through...	KB/sec	Avg. Bytes
Requête ...	60793	905	62	50009	1171.75	0.04%	37.9/sec	48.18	1302.6
TOTAL	60793	905	62	50009	1171.75	0.04%	37.9/sec	48.18	1302.6

(a)

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Through...	KB/sec	Avg. Bytes
Requête ...	70295	783	54	50006	1209.82	0.05%	43.8/sec	39.63	926.6
TOTAL	70295	783	54	50006	1209.82	0.05%	43.8/sec	39.63	926.6

(b)

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Through...	KB/sec	Avg. Bytes
Requête ...	77164	711	50	50007	867.64	0.02%	48.0/sec	25.89	552.2
TOTAL	77164	711	50	50007	867.64	0.02%	48.0/sec	25.89	552.2

(c)

FIGURE 4.5 – Rapports consolidés *JMeter* obtenus pour les différents modes de l'application.

Il est possible de constater que la colonne représentant la valeur maximum observée pour les temps de réponse est presque la même pour les trois expériences (i.e. environ 50000ms). Cela s'explique par le fait que notre serveur *HTTP Tomcat* est paramétré pour considérer les requêtes mettant plus de 50 secondes avant d'être traitées (i.e. en file d'attente) comme des requêtes tombées en erreur (i.e. *code HTTP 408*, *request timeout*). Le Tableau 4.1 offre un comparatif des résultats obtenus pour les trois modes du service de calcul d'itinéraire. Le $mode_3$, considéré comme le mode nominal, est pris comme référentiel. Il faut préciser que les gains observés sont fortement dépendants de l'application et du type de requête.

Tableau 4.1 – Tableau récapitulatif des résultats obtenus pour les trois modes du service.

mode service	**nombre requête**	**temps réponse moyen**	**débit sortie**	**gain performance**	**bande passante**	**gain ressource**
$mode_3$	60793	905ms	37.9req/sec	-	78Mo	-
$mode_2$	70295	783ms	43.8req/sec	13.5%	64Mo	18%
$mode_1$	77164	711ms	48req/sec	21%	42Mo	47%

4.1.5 Démonstrateur *TUBA*

Les premières expériences, visant à identifier l'impact de l'*élasticité logicielle* en termes d'utilisation de ressources et de performances, ont montré des résultats intéressants. De ce fait, nous avons entrepris de poursuivre les expérimentations dans le cadre de la mise en place d'un dimensionnement automatique des ressources et des services dans un environnement Cloud. Cela a donné lieu à notre démonstrateur *TUBA*. L'objectif du démonstrateur *TUBA* est de comparer le comportement de plusieurs architectures, face à un environnement dynamique, en termes de performances, de QoS, de consommation de ressources, etc. Dans notre cas, nous avons comparé trois architectures :

— architecture Cluster (*archi Cluster*) : infrastructure non-élastique dimensionnée en amont ;
— architecture Cloud "classique" (*archi Cloud*) : infrastructure élastique permettant d'ajuster dynamiquement le nombre de VM (i.e. dimensionnement horizontal) ;
— architecture dite "Éco-Élastique" (*archi Éco-Élastique*) : infrastructure élastique (i.e. dimensionnement horizontal) et application élastique (i.e. changement de mode du service).

La Figure 4.6 donne certains résultats obtenus par notre démonstrateur *TUBA*. Trois expériences de 30 minutes ont été réalisées pour tester le comportement des trois architectures face à un même scénario de montée en charge (cf. courbe noire hachurée). On peut constater que le scénario de charge correspond à une augmentation de la demande. Il s'agit concrètement de faire grimper le nombre de clients par pallier successifs (environ 5 minutes) de 10 *threads*. Les courbes rouge, bleue et verte représentent respectivement les résultats obtenus pour l'architecture *Cluster*, *Cloud* et *Éco-Élastique* en termes de temps de réponse (cf. Figure 4.6a) et de débit de sortie (cf. Figure 4.6b).

L'élasticité de l'infrastructure est mise en place par le biais de règles de dimensionnent automatique faisant appel à l'*API HTTP* de *CloudStack*. Il s'agit ici de faire varier le nombre de VMs du *tier Métier* de l'application de consultation des transports en commun. On considère une infrastructure limitée ne pouvant excéder 4 VMs pour le *tier Métier*. L'*élasticité logicielle* est elle-aussi implémentée à l'aide de règles à base de seuils. La partie "action" de ces règles revient à changer la valeur de la variable d'environnement *Tomcat* des VMs concernées (i.e. 3 valeurs possibles correspondants aux 3 modes du service). Comme pour les expériences précédentes, le $mode_3$ est considéré comme étant le mode par défaut.

La configuration initiale des expériences pour les différentes architectures est la suivante :

— *archi Cluster* : 4 VMs allumées dès le départ pour le *tier Métier* avec le service de calcul d'itinéraires paramétré avec le $mode_3$. Cette configuration reste la même tout le long de l'expérience (i.e. architecture non-élastique) ;

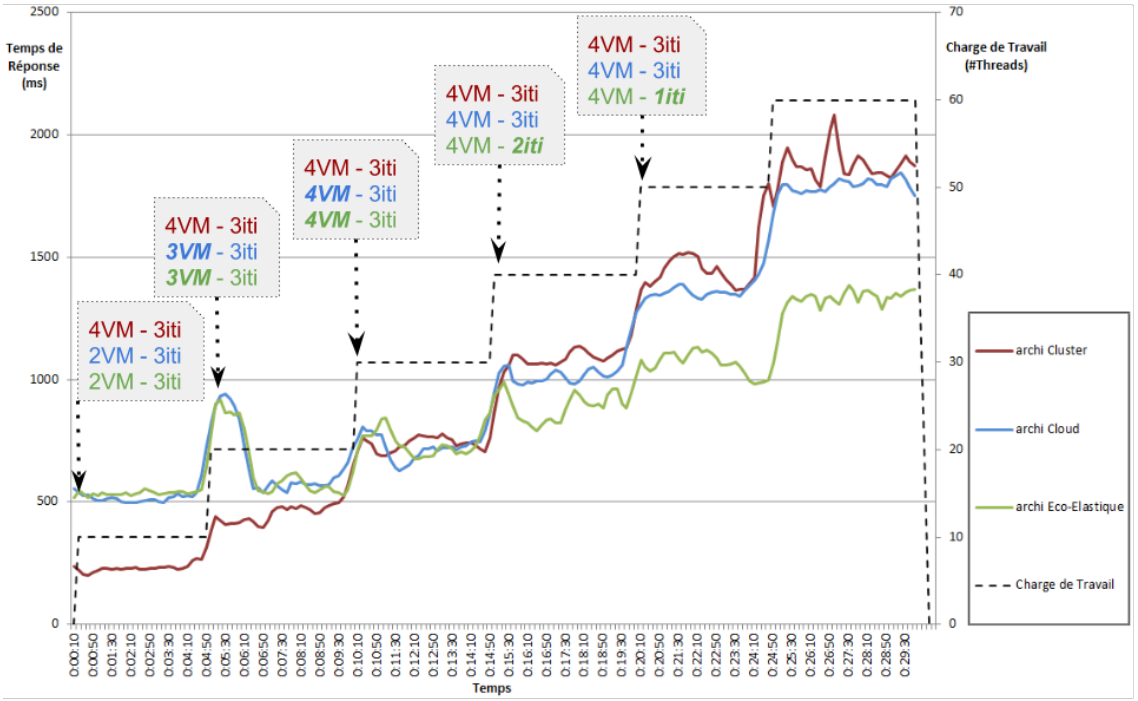
— *archi Cloud* : 2 VMs allumées au départ et le service de calcul d'itinéraires paramétré avec le $mode_3$. Le nombre de VMs est ajusté en fonction de la charge ;

— *archi Éco-Élastique* : configuration initiale identique à celle de l'*archi Cloud* (2 VMs, $mode_3$). Néanmoins, le nombre de VMs et le mode du service de calcul d'itinéraires sont ajustés.

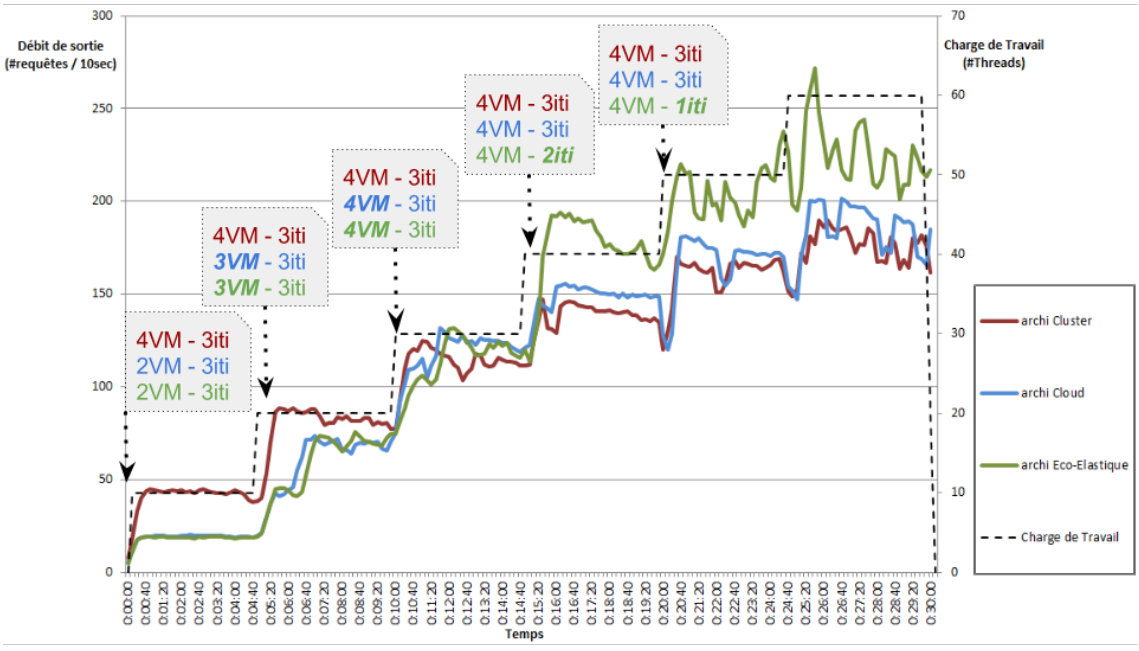
Dans le cadre des expériences présentées ici, nous avons donné une priorité à l'élasticité de l'infrastructure pour l'*archi Éco-Élastique*, c'est-à-dire que l'on fait d'abord appel au dimensionnement horizontal avant d'adapter l'application. L'*élasticité logicielle* est ainsi utilisée comme une capacité de passage à l'échelle supplémentaire dans le cas où les ressources IaaS viennent à manquer. Le fait de privilégier l'adaptation des ressources IaaS à celle des ressources logicielles représente un choix quant à la configuration de l'élasticité. En ce sens, il peut s'agir d'une préférence d'adaptation (i.e. stratégie) souhaitée et spécifiée par l'administrateur Cloud.

Une première remarque concerne les coûts induits par l'infrastructure. L'*archi Cluster* représente 4 VMs allumées tout le long de l'expérience tandis que les deux autres architectures, grâce à l'élasticité des ressources IaaS, permettent d'ajuster le nombre d'instances en fonction de la demande. Si l'on considère un SLO sur le temps de réponse indiquant par exemple que les requêtes doivent être traitées en moins d'une seconde (cf. Figure 4.6a), on peut affirmer que l'*archi Cluster* est surdimensionnée les 10 premières minutes de l'expérience puis sous-dimensionnée le reste de l'expérience. L'*archi Cluster* offre de meilleures performances les 10 premières minutes que les autres architectures mais cela se fait au détriment des coûts énergétiques et financiers.

Les trois architectures admettent les mêmes configurations (4 VMs / $mode_3$) entre la 10^{eme} et la 15^{eme} minutes ce qui se traduit par des performances semblables à savoir des temps de réponses de l'ordre de 700ms. À ce stade de l'expérience, la configuration maximale de l'infrastructure est atteinte pour les trois architectures (i.e. 4 VMs pour le tier Métier) ce qui est problématique du fait que le pic de charge n'a pas encore atteint son maximum. Dans ce cas, l'*archi Cloud*, tout comme l'*archi Cluster*, vont se retrouver limitées en termes d'adaptation.



(a)



(b)

FIGURE 4.6 – Résultats du démonstrateur *TUBA*.

L'*archi Éco-Élastique*, bien qu'ayant atteint sa configuration maximale en termes de ressources IaaS est encore en mesure d'adapter sa couche applicative en changeant le mode du service de calcul d'itinéraires. On peut voir une première reconfiguration de l'application à la 15^{eme} min visant à basculer du $mode_3$, calculant et retournant trois itinéraires, au $mode_2$, moins consommateur de ressources. Ainsi, entre la 15^{eme} et la 20^{eme} min, on constate des performances sensiblement meilleures pour l'*archi Éco-Élastique* qui admet des temps de réponse de l'ordre de 850ms là où les autres architectures dépassent la seconde.

La différence en termes de performances se fait fortement ressentir à partir de la 20^{eme} min, où l'*archi Éco-Élastique* passe du $mode_2$ au $mode_1$ pour absorber le pic de charge en soulageant davantage les ressources. En effet, on constate des temps de réponse de l'ordre de 1050ms (cf. 20-25 min) puis 1300ms (cf. 25-30 min) pour l'*archi Éco-Élastique* alors que les autres architectures voient leurs temps de réponse grimper de manière importante à 1400ms puis 1800ms.

En conclusion, l'*élasticité logicielle* permet de limiter l'impact de la montée en charge sur les performances en soulageant les ressources IaaS. Cela peut notamment permettre d'assumer les SLOs signés en termes de performances (e.g. temps de réponse) et ainsi éviter de payer des pénalités importantes en cas de violation de SLA. Un autre intérêt identifié de l'*élasticité logicielle* est le fait de pouvoir accepter davantage de clients avec la même infrastructure c'est-à-dire à coût constant. Ainsi, à configuration égale, on peut voir sur les 5 dernières minutes de l'expérience (cf. Figure 4.6b) que l'*archi Éco-Élastique* paramétrée avec le $mode_1$ offre un débit de sortie de l'ordre de 23req/sec contre 17-18req/sec pour les autres architectures paramétrées avec le $mode_3$.

4.2 *SCUBA* : vers une gestion de l'élasticité multi-couche

Les expériences menées avec le démonstrateur *TUBA* nous ont permis d'identifier certains points d'amélioration quant à notre gestionnaire autonomique (i.e. boucle *MAPE-K*). Ainsi, nous avons entrepris de développer un *framework* de gestion de l'élasticité multi-couche : *SCUBA*.

SCUBA est la continuité du projet *TUBA* évoqué précédemment. Il s'agit d'un *framework*, reposant sur une architecture de type *MAPE-K*, permettant de gérer automatiquement le redimensionnement du système au niveau des ressources IaaS et SaaS. Un effort important a été effectué concernant la partie surveillance (i.e. *monitoring*) de la boucle autonomique qui offre désormais la possibilité de collecter et d'agréger des métriques provenant de différents couches. De plus, contrairement au démonstrateur *TUBA*, une étude a été menée sur la coordination des actions de dimensionnement (i.e. adaptation multi-couche). En outre, le *framework SCUBA* est extensible et peut ainsi s'interfacer avec différents systèmes IaaS ou applications SaaS, contrairement au démonstrateur *TUBA* qui se limitait au cas d'utilisation de l'application de calcul d'itinéraires. Enfin, *SCUBA* est davantage *scalable* du fait que le gestionnaire autonomique peut lui-même être déployé sur une infrastructure virtualisée élastique et ainsi assurer efficacement la gestion du processus d'élasticité de systèmes à taille variable.

4.2.1 Architecture du *framework*

Le *framework SCUBA* offre une solution d'élasticité de bout en bout, depuis la collecte des données, traduisant l'état de santé du système, jusqu'à la reconfiguration de celui-ci. La Figure 4.7 donne une vision de l'architecture globale du *framework* en mettant l'accent sur la partie observation (i.e. *M* de *MAPE-K*) et les différentes technologies utilisées.

L'observation, dans notre solution, repose sur les *logs* et plus particulièrement sur *Logstash* [log15]. *Logstash* est un outil de collecte, analyse et stockage de *logs*, développé en *Java*. *Logstash* est parfois présenté comme un *ETL* (i.e. *Extract Transform Load*), outil permettant d'effectuer des synchronisations massives d'information d'une source de données vers une autre.

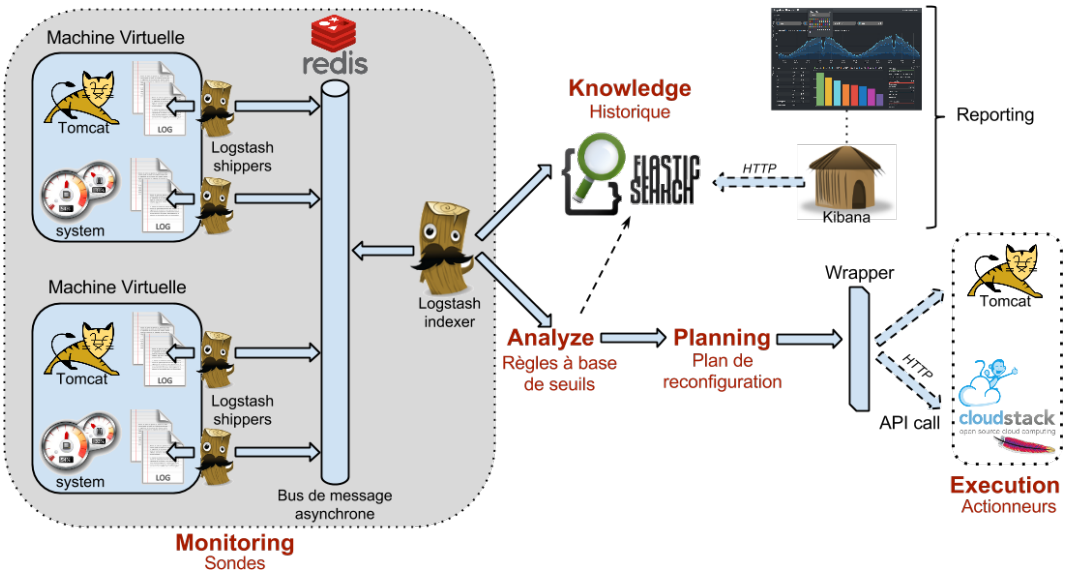


FIGURE 4.7 – Architecture générale du *framework SCUBA*.

Pour la collecte, des agents *Logstash* (i.e. *shipper*) sont déployés sur chacune de nos VMs. Dans notre exemple, on peut voir qu'un agent est chargé de la collecte des métriques de bas niveau (i.e. utilisation CPU et RAM) lues dans les fichiers de *logs* systèmes (i.e. *syslog*) tandis qu'un second agent est chargé de lire les fichiers de *access logs* de notre serveur d'application *Tomcat* contenant des métriques de haut niveau (i.e. temps de réponse, codes de retour *HTTP*, etc.). Concrètement, un *shipper Logstash* est chargé de lire les entrées dans des fichiers de *logs* perçues comme des événements. Ensuite, il est possible d'analyser ces événements et de les mettre en forme à l'aide de filtres (e.g. standardisation de la date, découpage/structuration du message, etc.). Après filtrage, on obtient un message clair et concis, avec des couples clé-valeur facilement exploitables.

Les *shippers* peuvent ensuite pousser ces messages sur différents canaux (i.e. email, sortie standard, fichier texte, etc.). Dans notre cas, il s'agit d'un bus de message asynchrone mis en place via un serveur *Redis* [red15]. *Redis* est à l'origine un système de gestion de base de données clé-valeur *scalable*, hautes performances, écrit avec le langage *C*. Le principal avantage de *Redis* (par rapport à *ActiveMQ*) est de conserver l'intégralité des données en RAM ce qui se traduit par d'excellentes performances en évitant des accès disques qui s'avèrent coûteux.

Un second agent *Logstash* (i.e. *indexer*) est chargé d'agréger les données récupérées dans le bus de message en vue d'obtenir des indicateurs de plus haut niveau comme la moyenne d'utilisation CPU d'un tier spécifique ou encore la moyenne des temps de réponse des requêtes sur tous les serveurs d'application *Tomcat* dont les VMs sont reliées à un répartiteur de charge spécifique. En ce sens, on associe parfois *Logstash* à un outil de traitement d'événements complexes (i.e. *Complex Event Processing - CEP*).

L'*indexer Logstash* peut lui aussi transmettre les données agrégées vers différents canaux. Dans notre cas, l'*indexer* pousse à la fois vers une base de données *NoSQL ElasticSearch* [ela15], jouant le rôle de base de connaissance (i.e. *Knowledge*) de *MAPE-K* en assurant la persistance des données, et vers le bus *Redis* (i.e. second canal) qui alimentera la tâche d'*Analyse* de la boucle autonomique en lui envoyant des événements de haut niveau comme décrit précédemment.

Les messages (i.e. événements) reçus par la boucle autonomique font l'objet d'un traitement par un outil de *CEP* (i.e. *CEP engine*) appelé *WildCat* [wil15]. Il est chargé de repérer les événements qui vont déclencher une (ou plusieurs) action(s) d'élasticité. Cela correspond à l'évaluation de la partie *condition* des règles à base de seuils. L'avantage d'un tel outil est de pouvoir considérer des conditions diverses et variées faisant intervenir de multiples métriques, provenant éventuellement de plusieurs ressources de différents niveaux (e.g. IaaS ou SaaS) ou encore de considérer la chronologie des événements, etc.

L'exécution du plan de reconfiguration fait appel à l'*API HTTP* de *Apache CloudStack* pour l'adaptation des ressources IaaS. En revanche, une action sur la couche SaaS est gérée à travers l'envoi d'un message (e.g. demande de changement de mode du service de calcul d'itinéraires) sur le bus asynchrone évoqué précédemment suivant un modèle événementiel classique de type *Observer* où l'on va "écouter" les messages circulant dans le bus. Lorsqu'un message est reçu par la VM concernée, on va modifier la valeur de sa variable d'environnement *Tomcat* correspondant au mode du service.

L'architecture mise en place propose aussi une composante de visualisation (i.e. *reporting*) reposant sur *Kibana*. Il s'agit d'une interface web permettant de rechercher des informations stockées par *Logstash* dans *ElasticSearch* et de les mettre en forme à travers des tableaux de bord (i.e. *dashboard*). Cet outil permet notamment de constater en temps réel l'état de santé du système (e.g. utilisation CPU/RAM, temps de réponse des requêtes par VM, par tier, etc.). Un autre avantage de cet outil est de pouvoir comparer le comportement de plusieurs architectures stressées en parallèle ce qui s'avère utile dans une démarche d'expérimentation ou de calibrage.

4.2.2 Expérimentations et validation

Le *framework SCUBA* a servi de socle aux expérimentations menées dans le cadre de notre publication scientifique à *IEEE/ACM CCGrid 2014* [KdODL14]. Le papier propose une solution complète pour la gestion des contrats de service du Cloud. Nous introduisons *CSLA* (*Cloud Service Level Agreement* [Kou13]), un langage dédié à la définition de contrat de service dans le Cloud. Celui-ci permet d'adresser finement les violations SLA via la dégradation fonctionnelle et des modèles de pénalité avancés. Nous proposons ensuite une solution de dimensionnement automatique dirigé par *CSLA* qui implémente l'élasticité de façon multi-couche (i.e. application et infrastructure). Notre solution est validée expérimentalement sur *SCUBA* (cf. Figure 4.7).

Scénario de motivation

Le cas d'utilisation adopté concerne le service de calcul d'itinéraires. Contrairement aux expérimentations menées en interne à l'entreprise, nous admettons ici trois acteurs qui sont le fournisseur IaaS, le fournisseur SaaS proposant le service de calcul d'itinéraires et les utilisateurs finaux (cf. Figure 4.8). Dans ce cas, les utilisateurs finaux correspondent à la fois à des entreprises de transport public ou des internautes.

Dans notre exemple, le fournisseur IaaS propose deux types d'instance (i.e. *offering*) à savoir *Small* et *Large*. De même que pour les *offerings* de VMs, le fournisseur SaaS propose deux types de service de calcul d'itinéraires admettant une QoS ainsi qu'une qualité de rendu différentes. Le premier service *S1* n'est pas élastique, c'est-à-dire qu'il calcule et retourne en permanence 3 itinéraires en réponse à une requête. Le service *S2* est quant à lui élastique et offre 2 modes (i.e. $mode_3$ et $mode_1$ référencés respectivement dans ce papier comme le mode *normal* et *degraded*) avec la possibilité de basculer dynamiquement de l'un à l'autre.

Nous considérons deux types de SLA (cf. Figure 4.8), l'un entre le fournisseur IaaS et le fournisseur SaaS, portant sur la disponibilité (i.e. *availability*) des ressources (i.e. VM) et l'autre, entre le fournisseur SaaS et ses clients, comprenant 3 SLOs sur les métriques de temps de réponse, de disponibilité ainsi que sur l'utilisation des modes du service de calcul d'itinéraires. En effet, nous estimons que l'utilisation des modes doit elle aussi être sujette à des SLOs. Ainsi, il devient possible pour le fournisseur SaaS et ses clients de statuer la proportion d'utilisation d'un mode par rapport à un autre en spécifiant par exemple que 80% du temps, le service répondra 3 itinéraires à une requête (i.e. mode *normal*) mais 20% du temps, seulement 1 itinéraire sera retourné (i.e. mode *degraded*). On peut alors imaginer une facturation à la requête dont le prix varie selon le mode utilisé.

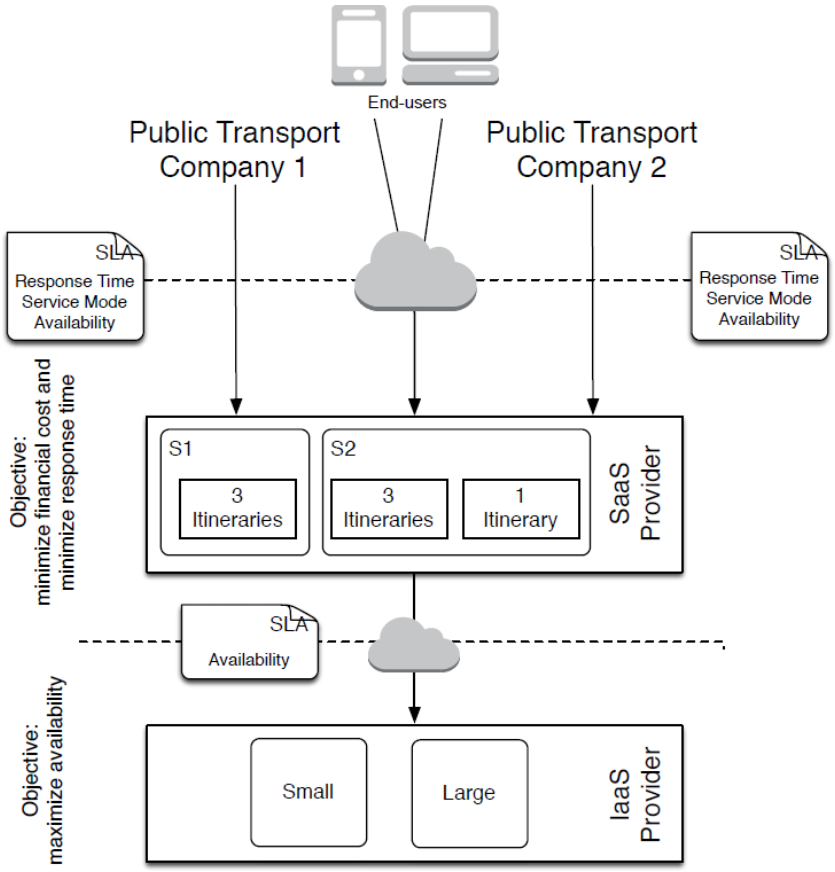


FIGURE 4.8 – Cas d'utilisation considéré dans notre papier [KdODL14] : acteurs, services et SLAs.

Les SLAs considérés ont été spécifiés à l'aide du langage *CSLA* [Kou13]. Les Tableaux 4.2 et 4.3 donnent respectivement les SLAs signés entre les fournisseurs IaaS et SaaS et entre le fournisseur SaaS et ses clients. Bien que nous n'allons par rentrer dans le détail du langage *CSLA* dans cette thèse, il faut simplement préciser qu'un SLO est défini par une contrainte à satisfaire pour une circonstance particulière. Cette circonstance est représentée par une pré-condition. Une contrainte est une expression composée d'une métrique, d'un comparateur et d'un seuil. Cette contrainte peut éventuellement avoir une priorité pour refléter les préférences du client.

Pour faire face aux fluctuations du Cloud et à l'incertitude sur la QoS, l'originalité de *CSLA* est d'introduire dans le contrat SLA les propriétés de *fuzzinessValue*, *fuzzinessPercentage* et *confidence*. Le langage distingue trois états possible pour une requête à savoir : idéal, dégradé et inadéquat. Le terme "idéal" désigne le fait que le seuil du SLO est bien respecté. Le terme "dégradé" signifie une dégradation de QoS. Cette dégradation consiste à utiliser une marge d'erreur *fuzzinessValue*. Au-delà de cette marge, il s'agit d'un état "inadéquat". Un SLO doit être garanti selon un calendrier, pendant la durée de validité de contrat, avec un pourcentage de confiance (i.e. *confidence*). La confiance est le rapport cas adéquats/nombre de cas (où adéquat = idéal ou dégradé).

Les deux fournisseurs ont des objectifs qui leurs sont propres. Le fournisseur IaaS souhaite maximiser la disponibilité de ses ressources en vue de respecter le SLA signé avec le fournisseur SaaS ; ce dernier souhaite quant à lui minimiser les temps de réponse pour respecter les engagements signés avec ses utilisateurs tout en minimisant ses coûts d'hébergement (cf. Figure 4.8).

Tableau 4.2 – SLA entre le fournisseur IaaS et le fournisseur SaaS.

Service	Métrique	Opérateur	Seuil	Fuzz.	% of Fuzz.	Confid.	Prix	Pénalité
Small/Large	Av.	$\geq$	99.95%	0	0	100	0.046$/h	10% if 99.95% $\leq$ Av. $\leq$ 99% 30% if Av. $\leq$ 99%

Tableau 4.3 – SLA entre le fournisseur SaaS et les utilisateurs finaux.

Service	Métrique	Opérateur	Seuil	Fuzz.	% of Fuzz.	Confid.	Pénalité
S1/S2	Temps de réponse	$\leq$	750ms	50ms	16.66%	90%	0.003$/rqt
	Av.	$\geq$	99,5%	0,5%			0,001$/rqt
S2	Mode(degraded)	$\leq$	30%	5%			0,0005$/rqt

Paramètres de l'expérience

Le dimensionnement automatique, qu'il soit au niveau de l'infrastructure ou de l'application elle-même, repose sur les règles à base de seuils. Nous considérons 3 types d'actions : *ScaleOut*, *ScaleIn* et *ScaleApp*. La dernière action revient à changer le mode de l'application de *Normal* à *Degraded* pour une durée spécifique puis de re-basculer dans le mode *Normal*. Dans le cadre de nos expériences, nous spécifions deux règles pour le dimensionnement IaaS et une règle pour l'adaptation SaaS. Les trois règles évoquées sont présentées ci-dessous. Il faut préciser que la valeur des seuils a été fixée en s'appuyant sur les résultats d'une campagne de tests de calibrage effectuée au préalable.

— ***ScaleOutRule*** *:*
If(avg(cpuLoad of VMs)) > 75% for 60sec
Then deploy new VM
— ***ScaleInRule*** *:*
If(avg(cpuLoad of VMs)) < 40% for 60sec
Then remove one VM
— ***ScaleAppRule*** *:*
If (avg(responseTime of Tomcat's VMs)) > 750ms for 30s
Then switch mode from Normal to Degraded for all instances during 3 minutes
Then switch back to Normal mode

Nous distinguons deux implémentations, la première, nommée *withoutSaaSElasticity*, correspond à l'utilisation du service *S1* tandis que la seconde, appelée *withSaaSElasticity*, correspond à l'utilisation du service *S2* offrant différents modes. Les règles d'élasticité *ScaleOutRule* et *ScaleInRule* sont activées pour les deux implémentations. En revanche, seule l'implémentation *withSaaSElasticity* est concernée par la règle *ScaleAppRule*. Il faut aussi préciser que dans le cas de l'implémentation *withSaaSElasticity*, l'action de dimensionnement de la règle *ScaleOutRule* (i.e. ajouter une VM) est suivie de l'action de la règle *ScaleAppRule* (i.e. basculer du mode *Normal* à *Degraded* pendant 3 min) dans le but de soulager les ressources le temps du démarrage de la VM et des services associés, prenant environ 3 min. En ce sens, nous adressons la coordination des actions d'élasticité en considérant une adaptation multi-couche au sein d'une même règle de dimensionnement automatique. L'objectif ici est de bénéficier de la synergie offerte par l'utilisation conjointe de l'élasticité IaaS et SaaS. Nous reviendrons plus tard dans ce manuscrit sur cette notion de synergie.

L'objectif de nos expériences est de comparer le comportement des implémentations *without-SaaSElasticity* et *withSaaSElasticity* selon différents critères (i.e. performances, coût, respect des SLAs, etc.). La Figure 4.9 regroupe certains résultats obtenus dans le cadre des expérimentations en considérant les SLAs décrits précédemment. Les Figures 4.9a, 4.9b, 4.9c, 4.9d et 4.9e illustrent respectivement les résultats observés en termes de temps de réponse, de disponibilité, de mode du service, de coût des ressources et enfin de satisfaction des SLAs.

Le scénario de charge appliqué pour cette expérience est d'une durée de 30 minutes (cf. courbe hachurée noire). L'injection de charge a été réalisée avec *Apache Jmeter*. Nous avons étudié un scénario correspondant à un pic de charge. Ainsi, les 20 premières minutes, 5 clients (i.e. *threads*) sollicitent l'application, puis nous simulons un pic de charge d'environ 5 minutes en faisant tripler la demande (i.e. de 5 à 15 *threads*) durant cette période. Enfin, un retour à la normale de la charge de travail est observé sur les 5 dernières minutes de l'expérience. Les deux implémentations, *withoutSaaSElasticity* et *withSaaSElasticity*, admettent la même configuration initiale. Celle-ci correspond à une infrastructure constituée de 3 VMs allumées dès le départ et le service de calcul d'itinéraires paramétré en mode *Normal* c'est-à-dire retournant 3 itinéraires en réponse à une requête.

Résultats

Une première observation possible, lorsque le pic de charge survient au temps $t = 20min$, est que l'implémentation *withSaaSElasticity* (cf. courbe verte) permet d'éviter l'ajout de ressources IaaS en absorbant l'augmentation de la charge de travail à l'aide de l'*élasticité logicielle* en basculant du mode *Normal* au mode *Degraded*. L'implémentation *withoutSaaSElasticity* (cf. courbe rouge), quant à elle, a nécessité l'ajout d'une nouvelle instance, c'est-à-dire une augmentation des coûts d'infrastructure, pour satisfaire la demande croissante (cf. Figure 4.9d). De plus, on peut constater que l'*élasticité logicielle* a été plus à même de passer à l'échelle rapidement face à l'augmentation brutale de la demande (cf. Figures 4.9a et 4.9b). En ce sens, la réactivité de l'*élasticité logicielle* permet d'approcher l'idéal du dimensionnement juste-à-temps (i.e. *just-in-time provisioning*).

Les temps de réponse et la disponibilité observés pour l'implémentation *withoutSaaSElasticity* dépassent nettement les valeurs fixées dans le cadre des SLOs (i.e. 750 ms et 99,5%) contrairement à l'implémentation *withSaaSElasticity*. Bien que la qualité d'expérience soit dégradée dans le cas de l'implémentation *withSaaSElasticity* (i.e. 1 itinéraire au lieu de 3), la flexibilité et la réactivité offertes par l'*élasticité logicielle* permet de fournir le service avec un niveau de performance et de QoS acceptable sans nécessité l'ajout de ressource IaaS.

La Figure 4.9e, mettant en lumière le respect des SLAs des deux implémentations, considère uniquement la dernière fenêtre de 10 minutes de l'expérience. Pour rappel, l'état *Ideal* indique que le seuil du SLO a été respecté, l'état *Degraded* désigne que la requête reste dans la marge d'erreur possible spécifiée dans le SLA (i.e. *Fuzziness*) et enfin l'état *Inadequate* indique que le SLO a été violé. Nous pouvons voir que l'implémentation *withSaaSElasticity* admet de meilleurs résultats en termes de violation de SLA (i.e. *Inadequate*). De plus, la proportion de requêtes *Ideal* de l'implémentation *withSaaSElasticity* est nettement supérieure à celle de l'implémentation *withoutSaaSElasticity*. Enfin, le débit de sortie (i.e. *throughput*) supérieur de l'implémentation *withSaaSElasticity* permet au fournisseur SaaS d'augmenter son profit en traitant davantage de requêtes (i.e. facturation à la requête) sans nécessairement augmenter ses coûts d'infrastructure.

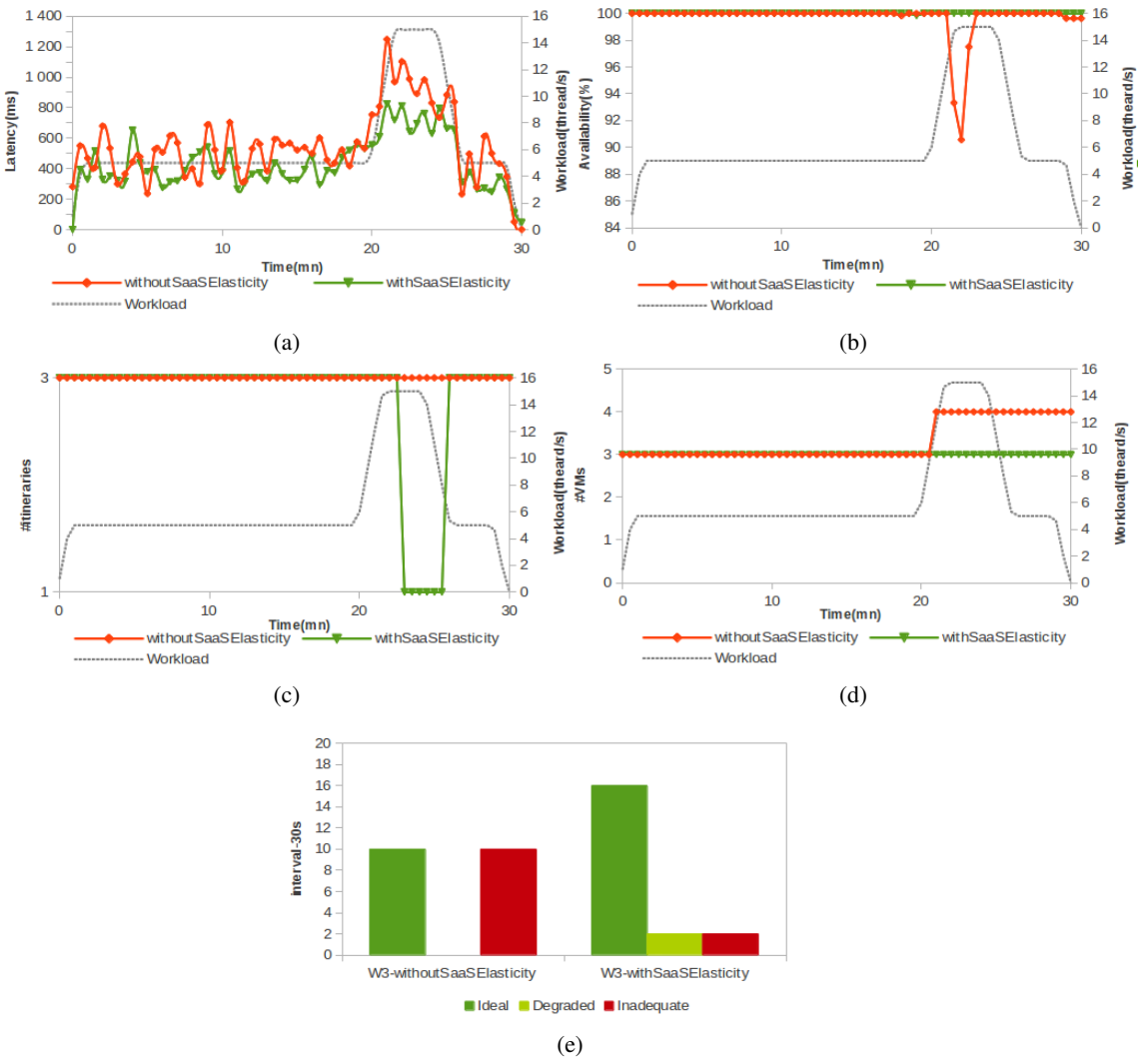


FIGURE 4.9 – Résultats des expérimentations.

Conclusion

Dans ce travail, les deux types d'élasticité n'ont pas fait l'objet d'une véritable coordination (i.e. règles de dimensionnement automatique distinctes), néanmoins, les actions d'adaptation ont bien été appliquées sur les deux couches (i.e. implémentation *withSaaSElasticity*) en considérant les exécutions de certaines actions comme complémentaires (i.e. *ScaleApp* pour soulager les ressources le temps du *ScaleOut*). Nous avons pris conscience qu'il serait intéressant de considérer certaines adaptations composites et multi-couches, qui, bien que nécessitant une coordination des actions de dimensionnement, permettraient de bénéficier davantage de la synergie entre les deux types d'élasticité.

En conclusion, nous avons montré dans ces expérimentations que l'*élasticité logicielle* offre un moyen efficace pour rendre l'adaptation davantage flexible et réactive face à un environnement dynamique (e.g. pic de charge soudain). Cela permet en l'occurrence de maintenir un meilleur rapport QoS/coût. En ce sens, l'*élasticité logicielle* est à la fois une alternative et un complémentaire à l'élasticité de l'infrastructure permettant au fournisseur SaaS de respecter ses engagements en termes de SLAs tout en maximisant son profit par la réduction de ses coûts d'infrastructure.

4.3 Bilan

Les expérimentations réalisées portant sur notre cas d'utilisation d'application de consultation de transports en commun ont montré certains bénéfices de l'*élasticité logicielle*.

4.3.1 Réactivité de l'adaptation

Nous avons évoqué dans la section 3.1.1 que la réactivité de l'adaptation était l'une des caractéristiques essentielles attendues d'un processus de gestion de l'élasticité. Cependant, le temps d'initialisation des ressources informatiques n'est pas négligeable, ce qui induit parfois un manque réactivité face aux changements d'états du système.

Dans le cas de notre application de consultation des transports en commun, nous avons mesuré la durée nécessaire au démarrage d'une VM pour le *tier Métier*. Cela comprend le fait de démarrer la VM via *CloudStack*, puis le système d'exploitation, les différents services (*Tomcat*, *OpenTripPlanner*, etc.) et enfin d'associer la VM au répartiteur de charge (i.e. paramétrage via *CloudStack*). L'ensemble de ces opérations prend environ 3 minutes, ce qui reste raisonnable. En effet, l'entreprise *Sigma Informatique* a mesuré des temps d'initiation pouvant atteindre plus de 10 minutes pour certains environnements. Cependant, même avec une durée de 3 minutes, le *Scale Out* ne permet pas d'être pleinement réactif pour passer à l'échelle ce qui peut s'avérer problématique dans un contexte hautement dynamique.

Dans notre exemple, l'adaptation du logiciel revient à changer une variable d'environnement *Tomcat*, indiquant le mode du service de calcul d'itinéraires. Cette opération nécessite une à deux secondes. Cela permet une réactivité de reconfiguration très appréciable dans le cas d'applications confrontées à des changements fréquents de la demande (i.e. *fluctuating workload*). Bien que les valeurs mesurées pendant nos expérimentations (i.e. 3 minutes pour le *Scale Out* et 2 secondes pour le changement de mode) soient fortement dépendantes de notre cas d'utilisation et ne constituent pas des valeurs de référence, cela reflète tout de même d'une différence notable en termes d'ordre de grandeur de temps de reconfiguration.

4.3.2 Granularité et précision de l'adaptation

Un autre gain identifié concerne la granularité du passage à l'échelle. Il s'agit de considérer la granularité des actions d'adaptation offertes par l'*élasticité logicielle* ainsi que l'impact de la reconfiguration. L'*élasticité logicielle*, en adaptant le contenu plutôt que le contenant, permet la mise en place d'actions d'adaptation à granularité fine (et plus réactive). Il est ainsi possible de soulager subtilement les ressources en changeant par exemple le mode du service de calcul d'itinéraires d'une unique VM. L'*élasticité logicielle* offre ainsi la possibilité d'un passage à l'échelle à granularité fine qui permet de combler les manques concernant la précision de l'adaptation (cf. section 3.1.1).

Bien que l'impact de l'adaptation du logiciel sur l'utilisation des ressources soit variable (i.e. dépendant des applications), l'impact en termes de coûts de reconfiguration (i.e. temps et risque) est moindre comparé à la reconfiguration des ressources IaaS. Par exemple, le fait d'arrêter une instance représente un risque important si la charge de travail est fluctuante. En effet, cela peut nécessiter de rallumer une VM (i.e. effet *ping-pong*) ce qui s'avère couteux en termes d'utilisation de ressources, de temps, d'énergie mais aussi du point de vue financier (e.g. instance facturée à l'heure). En ce sens, l'*élasticité logicielle*, du fait de sa réactivité et de ses actions à granularité fine, diminue ce risque de reconfiguration.

4.3.3 Extension de la capacité du système

Le démonstrateur *TUBA* a aussi montré des résultats intéressants dans le cas de Cloud privé avec une infrastructure limitée. Prenons l'exemple d'une entreprise proposant une application SaaS (e.g. consultation des transports en commun) qu'elle héberge sur sa propre infrastructure. Celle-ci est possiblement modeste et constituée de quelques serveurs impliquant des contraintes sur l'utilisation des ressources (e.g. 4 VMs maximum pour le *tier Métier*). Dans certains cas (e.g. montée en charge soudaine non anticipée, partie de l'infrastructure en cours de mise à jour, etc.), les ressources peuvent venir à manquer. L'*élasticité logicielle*, dans ce cas, est une alternative au débordement de charge (i.e. *Cloud bursting*) qui vise à activer des ressources sur un Cloud externe (i.e. *public*).

L'*élasticité logicielle* offre la possibilité de soulager les ressources IaaS, en suivant une stratégie que l'on peut qualifier de "meilleur-effort" (i.e. *best-effort*), le temps que le système retrouve une activité normale. Il s'agit dans ce cas d'augmenter la capacité de traitement du système en changeant le mode de l'application (e.g. $mode_3$ vers $mode_1$) ce qui permet de traiter davantage de requêtes (cf. Figure 4.5) tout en maintenant un certain niveau de service (i.e. QoS et performance). En outre, cela permet de passer à l'échelle sans nécessairement solliciter des ressources supplémentaires (e.g. contrainte budgétaire).

4.3.4 Frugalité du passage à l'échelle

L'élasticité du Cloud revient actuellement à ajouter ou retirer des ressources au niveau IaaS pour répondre à une charge de travail variable. Bien que le fait de pouvoir ajouter des ressources à des fins de passage à l'échelle (en théorie de manière infinie) soit appréciable, cela implique des coûts énergétiques et financiers non-négligeables. En ce sens, l'*élasticité logicielle* apporte une vision frugale du passage à l'échelle. Nous parlons parfois d'innovation frugale à l'entreprise pour positionner la démarche de l'*élasticité logicielle* dans un contexte plus global. L'innovation frugale est définie comme "une démarche consistant à répondre à un besoin de la manière la plus simple et efficace possible en utilisant un minimum de moyens. Elle est souvent résumée par le fait de fournir des solutions de qualité à bas coût ou d'innover mieux avec moins" [inn15].

Dans le cas de nos travaux, l'*élasticité logicielle* offre une alternative à l'ajout systématique de ressources pour le passage à l'échelle. Les résultats préliminaires ont en effet montré qu'il était possible de continuer de fournir le service aux utilisateurs avec un niveau de service et de performance acceptable sans nécessairement ajouter des ressources, synonyme d'augmentation des coûts (e.g. infrastructure, licences, etc.) et de l'empreinte énergétique. Pour cette raison, nous faisons parfois référence à nos travaux selon le terme *éco-élasticité* qui couvre à la fois les aspects énergétiques et financiers.

4.3.5 Synergie entre l'élasticité des ressources IaaS et SaaS

Le but de l'*élasticité logicielle* n'est pas de supplanter l'élasticité des ressources IaaS, indispensable au domaine du Cloud. L'*élasticité logicielle* peut être vue comme une nouvelle dimension d'élasticité, offrant de nouveaux leviers de reconfiguration qui peuvent être mis à profits pour renforcer la capacité d'adaptation globale du système en apportant davantage de souplesse dans la gestion des ressources Cloud. Les limites actuelles de l'élasticité des ressources IaaS ainsi que les bénéfices de l'*élasticité logicielle* que nous avons identifié lors de nos expériences nous ont amené à considérer la synergie possible entre les deux types d'élasticité. Il s'agit concrètement de mettre à profit la réactivité et la granularité fine du passage à l'échelle offerte par l'*élasticité logicielle* en vue de pallier aux limites de l'élasticité des ressources IaaS.

L'objectif revient à utiliser conjointement les deux types d'élasticité pour bénéficier des avantages de chacun. Il s'agit de coordonner les actions d'élasticité pour en tirer les meilleurs bénéfices. Le choix du terme "synergie" n'est pas anodin car il définit la "mise en commun de plusieurs actions concourant à un effet unique et aboutissant à une économie de moyens" [syn15]. La coordination des actions d'élasticité peut prendre de multiples formes, le but étant de gérer au mieux les différentes situations survenant à l'exécution (e.g. pic de charge soudain, pannes, etc.) et nécessitant une adaptation. Le fait d'étendre le concept d'élasticité aux couches hautes du Cloud (i.e. PaaS, SaaS) et de considérer l'élasticité de manière transverse et multi-couche ouvre la voie vers de nouvelles possibilités d'adaptation.

4.3.6 Pour aller plus loin

Dans le cadre de nos travaux autour de l'élasticité dans le Cloud, nous avons entrepris, dès la première année de thèse, de valider nos intuitions quant au fait d'étendre les capacités d'adaptation du système en adressant les ressources informatiques des couches hautes du Nuage. Ainsi, un projet autour de l'*élasticité logicielle* (i.e. SaaS) a rapidement été lancé à l'entreprise en vue de confirmer la pertinence de la proposition de la thèse. Le démonstrateur *TUBA* résultant de ce projet nous a permis d'identifier les gains potentiels de l'*élasticité logicielle* et plus généralement l'intérêt de considérer l'élasticité de manière multi-couche.

Le travail d'analyse de l'état de l'art, couplé aux résultats encourageants du démonstrateur *TUBA*, nous ont amené à proposer un premier *framework* de gestion autonomique de l'élasticité : *Scuba*. Il s'agit d'une solution de dimensionnement automatique suivant une approche réactive basée sur les règles à base de seuils et considérant des actions d'adaptation sur différentes couches (i.e. IaaS/SaaS). Un effort important a été fourni en ce qui concerne la surveillance des ressources du système. En revanche, bien que *Scuba* nous a permis d'appréhender les tenants et les aboutissants liés à l'adaptation des différentes ressources du Cloud et d'envisager la synergie possible entre les différents types d'élasticité, peu d'efforts ont été fournis dans cette voie à savoir le fait de considérer des adaptations composites et multi-couche.

Grâce à *Scuba*, nous avons identifié de nombreuses pistes d'amélioration pour la mise en place d'une solution autonomique permettant d'assurer une véritable gestion de l'*élasticité multi-couche* de bout en bout, depuis son paramétrage par l'administrateur Cloud jusqu'à son exécution. Cela nous a amené à conceptualiser davantage l'*élasticité logicielle* et plus généralement l'*élasticité multi-couche* (cf. chapitre 5). De plus, *Scuba* fut l'occasion d'identifier un besoin de fournir à l'administrateur Cloud les outils nécessaires pour lui permettre de paramétrer ce nouveau processus de gestion de l'*élasticité multi-couche* (cf. chapitre 6) et ainsi industrialiser celui-ci.

5

Modèle d'élasticité multi-couche

La proposition de cette thèse, visant à étendre le concept d'élasticité aux couches hautes du Cloud, trouve son sens du fait que le modèle actuel de l'élasticité admet certaines limitations en termes de *réactivité* et de *précision* d'adaptation (cf. sous-section 3.1.1). L'*élasticité logicielle*, en considérant les ressources propres à la couche SaaS, offre une capacité d'adaptation à granularité plus fine permettant de pallier aux lacunes de l'élasticité de l'infrastructure, qui bien que inhérente et nécessaire au modèle du Cloud, reste aujourd'hui limitée d'un point de vue physique (e.g. les ressources ne sont pas infinies et peuvent venir à manquer), conceptuel (e.g. modèle de facturation) ou encore technique (e.g. temps d'initialisation d'une instance).

Dans ce chapitre, nous définissons l'*élasticité logicielle* ainsi que les différents concepts relatifs à cette nouvelle capacité d'adaptation. Nous proposons un modèle de l'*élasticité logicielle* s'inspirant de celui de l'élasticité de l'infrastructure, et précisons les tenants et les aboutissants de ce type d'adaptation (cf. section 5.1). Nous nous intéresserons ensuite à la gestion autonomique des ressources Cloud au sens large, ce qui revient à considérer les deux types d'élasticité (i.e. IaaS et SaaS) dans un même processus d'adaptation automatique (cf. section 5.2). Nous présentons alors globalement notre *framework* de gestion de l'élasticité multi-couche *ElaStuff*, reposant sur un gestionnaire autonomique de type *MAPE-K*.

Contents

5.1 Étendre l'élasticité à la couche SaaS

Dans cette section, nous définissons l'*élasticité logicielle* ainsi que les différents concepts liés à celle-ci, puis nous présentons le modèle de l'*élasticité logicielle* ainsi que les différents aspects impactés par cette nouvelle capacité d'adaptation. Enfin, nous nous intéressons à l'intégration de l'*élasticité logicielle* avec le modèle d'élasticité actuel, reposant sur l'adaptation de la couche IaaS, en vue d'appréhender un nouveau modèle d'élasticité multi-couche qui résulte de l'adaptation des ressources Cloud au sens large.

5.1.1 Élasticité logicielle : définitions et concepts

Élasticité logicielle

Nous définissons l'*élasticité logicielle* comme la *capacité d'un logiciel à s'adapter, idéalement de manière autonome, pour répondre aux changements de la demande et/ou aux limitations de l'élasticité des ressources de l'infrastructure.* Par analogie avec l'élasticité de l'infrastructure où les ressources IaaS (e.g. VMs) sont ajustées dynamiquement pour satisfaire les contrats de niveau de service (i.e. SLA), l'*élasticité logicielle* offre les moyens d'ajuster les ressources SaaS (i.e. *composants logiciels*) de manière transparente et rapide afin de respecter davantage les attentes et contraintes du point de vue de la QoS, des performances et des coûts en palliant les manques de l'élasticité de l'infrastructure en termes de *précision* et/ou de *réactivité* d'adaptation.

Différentes configurations

L'*élasticité logicielle* revient à fournir des applications offrant différents niveaux de service, plus ou moins consommateurs de ressources (e.g. CPU, RAM, énergie, etc.). Une application admet plusieurs *configurations* possibles qui résultent du choix des composants qui la constitue, de leur nombre et de leur paramétrage. Contrairement à l'élasticité de l'infrastructure qui revient à ajuster les ressources IaaS (e.g. VM, CPU, RAM, etc.), nous considérons ici les ressources propres aux couches hautes du Cloud (i.e. SaaS, PaaS) à savoir les *composants logiciels*. Ces composants logiciels sont à granularité variable et peuvent concerner le logiciel lui-même (e.g. composant de calcul d'itinéraires, de messagerie instantanée, etc.) ou le *middleware* (e.g. serveur *HTTP*, *SGBD*, etc.).

Les multiples *configurations* possibles d'une application peuvent être le résultat de changements *architecturaux* des éléments qui la constitue (i.e. le choix et l'agencement des composants logiciels) ou de *paramétrisation* des composants existants (e.g. changer la valeur du paramètre *MaxClients* d'un serveur *Apache* [apa15] permettant de fixer le nombre d'utilisateurs simultanés que le serveur peut prendre en charge). Ainsi, il devient possible d'ajuster l'application d'un point de vue architectural mais aussi de configurer celle-ci en jouant sur les paramètres des composants sous-jacents. Les différentes configurations offertes par l'application sont le résultat du choix des composants, de leur nombre, de leur paramétrage (i.e. *offering*) ainsi que de leur agencement. L'application, selon son architecture et le paramétrage des composants qui la constituent (i.e. *configuration*), va offrir des résultats différents en termes de consommation de ressources, de QoS, de performance mais aussi d'expérience utilisateur.

Différentes offres

Pour ce qui est de la *paramétrisation* des applications SaaS et par analogie avec les ressources IaaS (i.e. VM) qui peuvent admettre différentes configurations (i.e. types d'instance), nous considérons que les composants de l'application peuvent eux aussi être fournis selon différents modes, que nous appelons offre ou *offering* (i.e. Off_{soft}). Ces différentes Off_{soft} offertes par l'application *SaaS* au travers de ses composants, tout comme les différents types d'instances (i.e. Off_{infra}) pour la couche *IaaS*, vont offrir différents niveaux de service. Il devient alors possible d'ajuster l'application en basculant certains de ces composants d'une offre à une autre.

Dans le cas de notre application de consultation de transport en commun (cf. sous-section 4.1.3), le composant de calcul d'itinéraires peut être paramétré selon trois Off_{soft} (i.e. $mode_1$, $mode_2$ et $mode_3$). De même, un composant de chiffrement *RSA* [rsa15] peut offrir différents modes qui reviennent à faire varier la taille de la clé *RSA* (e.g. 1024, 2048 ou 4096 bits). Plus la taille de la clé est importante, plus la sécurité est assurée, néanmoins, cela nécessite un traitement plus complexe qui va s'avérer davantage consommateur de ressources.

QoE, QoS et QoF

L'expérience utilisateur (i.e. *user experience*) est un terme apparu dans les années 2000 visant à qualifier le résultat et le ressenti de l'utilisateur lors d'une manipulation d'un objet fonctionnel ou d'une interface homme-machine [use15]. La qualité de l'expérience utilisateur ou *QoE* (i.e. *Quality of Experience*) est étroitement liée à la notion de QoS. En effet, des attributs de QoS tels que la disponibilité ou encore le temps de réponse d'une application (ou d'un service) vont influer grandement sur le ressenti de l'utilisateur. Ainsi, une meilleure QoS (offerte par le fournisseur de service) va entrainer une meilleure QoE (perceptible par l'utilisateur du service).

Dans le contexte de l'*élasticité logicielle*, les différentes *configurations* offertes par l'application peuvent mener à une expérience fonctionnelle distincte du point de vue de l'utilisateur final (e.g. service de calcul d'itinéraires), lui donnant l'impression d'un service de plus ou moins bonne qualité [FHTG10]. Nous introduisons le terme *qualité de fonctionnalité* ou *QoF* (i.e. *Quality of Functionality*) pour définir cette qualité fonctionnelle du service. Nous avons fait le choix d'introduire le nouveau terme *QoF* du fait que les termes *QoE* et *QoS* existants ne sont pas suffisants pour qualifier la variabilité fonctionnelle induite par l'*élasticité logicielle* ainsi que les nouveaux compromis qui en résultent. L'*élasticité logicielle* (i.e. reconfiguration architecturale et/ou paramétrisation) et les différentes *configurations* de l'application sous-jacentes peuvent fournir un contenu fonctionnel plus ou moins riche aux utilisateurs. On parle respectivement de *QoF* haute et basse. Bien que désignant des aspects distincts, les concepts de *QoE*, *QoS* et *QoF* sont corrélés.

Dans l'idéal, on souhaite bien entendu une *QoE*, une *QoS* et une *QoF* haute. On considère que la *QoE* inclut la *QoF* et la *QoS*. Ainsi, une *QoF* et une *QoS* hautes impliquent une *QoE* élevée. Une *QoF* élevée va certes améliorer la *QoE* ressentie par l'utilisateur final, cependant, cela va aussi influer sur la *QoS*, mais cette fois de manière négative. En effet, un contenu fonctionnel plus riche (e.g. fonctionnalités/composants supplémentaires, *offering* supérieure, etc.) va solliciter davantage les ressources IaaS (e.g. CPU, RAM, etc), ce qui risque, à ressource IaaS constante, de faire baisser la *QoS*. La *QoF* et la *QoS* admettent donc des tendances contraires à ressource IaaS constante (i.e. $QoF \propto \frac{1}{QoS}$). Ce nouveau compromis *QoF-QoS*, propre à l'*élasticité logicielle*, est illustré dans la Figure 5.1.

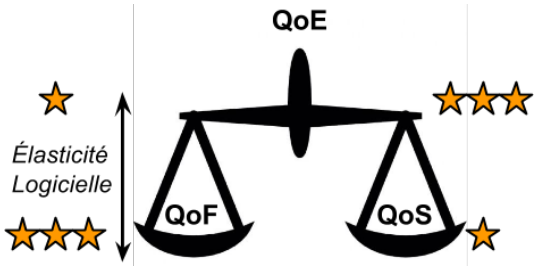


FIGURE 5.1 – Illustration du compromis *QoF-QoS* à ressource IaaS constante.

Il faut noter qu'il n'est pas possible d'agir de façon directe sur la *QoE* et la *QoS* (i.e. résultats des actions d'adaptation IaaS et SaaS), mais que l'*élasticité logicielle* offre les leviers nécessaires pour agir directement sur la *QoF* (et donc indirectement sur la *QoS* et la *QoE*). En résumé, l'*élasticité logicielle* fait apparaitre de nouveaux compromis qui viennent s'ajouter aux compromis classiques de l'élasticité de l'infrastructure (e.g. QoS fournie - coûts liés à quantité de ressources).

5.1.2 Nouvelles dimensions d'élasticité et actions associées

Dimensions d'élasticité

Par analogie avec l'élasticité des ressources IaaS, nous déclinons l'*élasticité logicielle* en deux types d'élasticité (i.e. *dimensions*) : verticale (VS_{soft}) et horizontale (HS_{soft}). L'*élasticité logicielle* verticale consiste à changer le niveau de service de l'application à travers la *paramétrisation* de ses composants (e.g. changement d'*offering*) alors que l'*élasticité logicielle* horizontale revient à apporter des changements architecturaux à celle-ci par l'ajout/retrait de composants logiciels (i.e. fonctionnels ou non). Dans notre travail, nous parlons de composants logiciels pour désigner les ressources PaaS/SaaS au sens large, pouvant admettre différents niveaux de granularité. Cependant, on retrouve de plus en plus le terme de *micro-service* pour désigner ces briques logicielles dans un contexte Cloud. L'*élasticité logicielle* horizontale est liée à l'élasticité horizontale de l'infrastructure du fait que l'ajout/retrait d'une VM revient généralement à connecter/déconnecter des composants (contenus dans la VM). Néanmoins, il est aussi possible de faire appel à l'*élasticité logicielle* horizontale de manière indépendante en ajoutant par exemple un composant logiciel sur une VM existante (e.g. téléchargement, installation et configuration du composant).

La Figure 5.2 illustre les deux types d'*élasticité logicielle*. Contrairement à la Figure 2.8, qui s'intéresse à l'élasticité de l'infrastructure et porte ainsi sur des ressources IaaS (i.e. VM, CPU, RAM, etc.), on peut voir ici que l'*élasticité logicielle* concerne des ressources propres à la couche SaaS, à savoir des composants logiciels. L'axe horizontal revient à ajouter (cf. *Scale Out* - SO_{soft}) ou retirer (cf. *Scale In* - SI_{soft}) un ou plusieurs composant(s) logiciel(s) (e.g. messagerie instantanée, chiffrage des données, recommandation pour un site E-commerce, etc.). L'axe vertical, quant à lui, correspond à la *paramétrisation* d'un composant. À l'instar de l'élasticité de l'infrastructure, nous reprenons le terme *offering* pour désigner le mode d'utilisation d'un composant. Ainsi, un composant fourni avec différentes Off_{soft} peut être reconfiguré dynamiquement en fonction du contexte d'exécution. Les termes *Scale Up* (SU_{soft}) et *Scale Down* (SD_{soft}) sont eux aussi conservés pour définir respectivement le passage d'un composant à une Off_{soft} supérieure et inférieure. Pour rappel, une Off_{soft} supérieure admet un meilleur niveau de fonctionnalité (i.e. *QoF*) au détriment de la sollicitation des ressources IaaS sous-jacentes (i.e. Off_{soft} plus ou moins gourmandes). Ce compromis est visible dans la Figure 5.2 au travers des étoiles, indiquant le niveau de *QoF*, et des labels énergétiques, représentant la consommation de ressources au sens large (CPU, RAM, réseau, énergie, etc.).

Actions d'élasticité multi-couche

Le fait de considérer l'élasticité de manière multi-couche revient à ajouter les 2 nouvelles dimensions de l'*élasticité logicielle* (i.e. VS_{soft} et HS_{soft}) et les 4 actions associées aux dimensions et actions classiques du modèle d'élasticité de l'infrastructure actuel. Le Tableau 5.1 reprend les différentes dimensions d'élasticité et actions associées considérées dans nos travaux. Il ne s'agit pas d'une liste exhaustive des dimensions/actions possibles dans un environnement Cloud. En effet, il serait possible de considérer d'autres actions telle que la migration de VM, de nouveaux états pour les ressources (e.g. VM en hibernation) ou encore de nouvelles dimensions et actions associées à de nouveaux types de ressources comme les *conteneurs logiciels* (e.g. *Docker* [DOC15]).

Les 8 actions présentées dans le Tableau 5.1, pouvant être qualifiées de *"atomiques"*, constituent l'*API* (*Application Programming Interface*) d'adaptation de base. En effet, il s'agit des actions basiques à partir desquelles il est possible de définir une *API* complète en faisant par exemple varier le nombre de paramètres des méthodes sous-jacentes. Une telle *API* peut être riche et donc permettre d'exprimer de manière concise une action évoluée sans nécessairement avoir recours à de multiples appels aux actions primitives. Par exemple, la méthode *deployVirtualMachine* de l'*API* de *Apache CloudStack* [dep16], qui crée et démarre une nouvelle VM (i.e. SO_{infra}), contient 3 paramètres obligatoires et 27 paramètres optionnels offrant une multitude d'appels possibles.

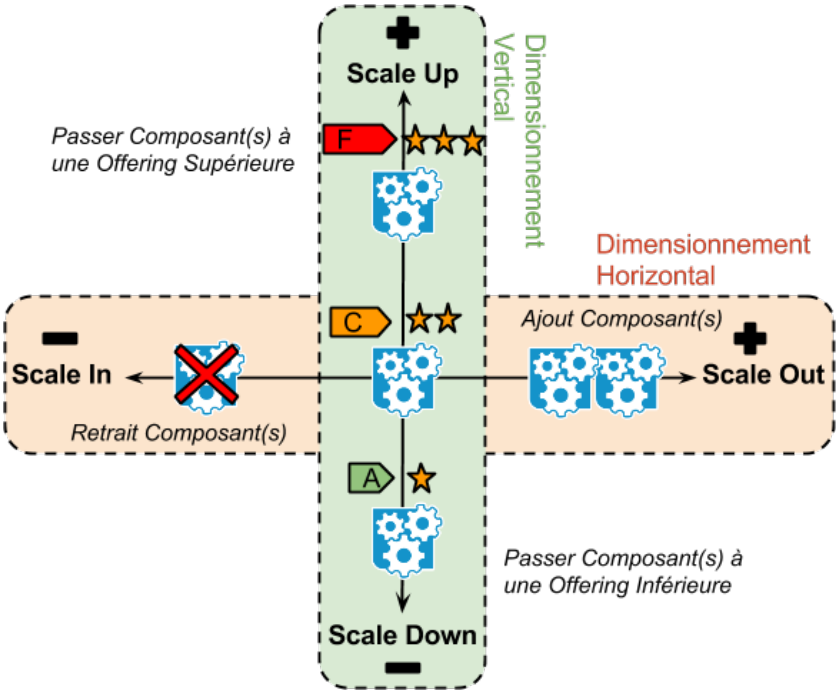


FIGURE 5.2 – Actions d'adaptation de l'*élasticité logicielle*.

Tableau 5.1 – Dimensions et actions de l'élasticité multi-couche.

Dimensions d'élasticité	**Actions associées**	**Descriptions**
Infrastructure Horizontal Scaling (HS_{infra})	Scale Out Infrastructure (SO_{infra})	Ajout de VM(s)
	Scale In Infrastructure (SI_{infra})	Retrait de VM(s)
Infrastructure Vertical Scaling (VS_{infra})	Scale Up Infrastructure (SU_{infra})	Augmenter Off_{infra} de VM(s)
	Scale Down Infrastructure (SD_{infra})	Diminuer Off_{infra} de VM(s)
Software Horizontal Scaling (HS_{soft})	Scale Out Software (SO_{soft})	Ajout de composant(s) logiciel(s)
	Scale In Software (SI_{soft})	Retrait de composant(s) logiciel(s)
Software Vertical Scaling (VS_{soft})	Scale Up Software (SU_{soft})	Augmenter Off_{soft} de composant(s)
	Scale Down Software (SD_{soft})	Diminuer Off_{soft} de composant(s)

5.1.3 Critères d'adaptation et compromis induits

Critères et actions d'adaptation

Les différentes actions d'élasticité considérées dans nos travaux impactent de manière distincte le système. On retrouve parfois le terme *elasticity dimension* dans la littérature [MCTD13] [CMTD13a] pour définir les différentes catégories de *critères* impactées par l'élasticité de l'infrastructure. Le fait de prendre en compte l'*élasticité logicielle* dans l'adaptation du système nécessite de considérer de nouveaux *critères*. Dans notre travail, nous considérons 6 *critères* :

- *Coût d'infrastructure* : l'impact budgétaire de l'adaptation (actions IaaS) ;
- *QoS* : l'impact de l'adaptation sur la *QoS* rendue (actions IaaS/SaaS) ;
- *QoF* : l'impact de l'adaptation sur la *QoF* fournie (actions SaaS) ;
- *Consommation énergétique* : l'impact énergétique de l'adaptation (actions IaaS et SaaS) ;
- *Temps de reconfiguration* : le temps de mise en place de l'adaptation (actions IaaS/SaaS) ;
- *Réactivité* : la rapidité de mise en place de l'adaptation (actions IaaS/SaaS), c'est-à-dire la capacité à réagir rapidement aux changements du système [HKR13].

Les *critères* évoqués ci-dessus ne constituent pas la liste exhaustive des dimensions impactées par l'élasticité multi-couche. Néanmoins, cette liste couvre selon nous les dimensions essentielles à prendre en compte dans le cadre de notre travail. En effet, les 8 actions d'adaptation considérées font apparaître des compromis entre ces 6 *critères*. Le Tableau 5.2 dresse les tendances de l'impact de chaque action d'adaptation sur les différents *critères*. Les symboles $-$, $+$ et $\emptyset$ signifient respectivement que l'action d'adaptation a un impact négatif, positif ou nul sur le *critère* concerné tandis que le symbole $*$ indique une tendance variable. Il est important de préciser qu'il s'agit des tendances globales des actions de base. En effet, en fonction de la méthode de l'*API* appelée, l'impact sur le *critère* sera plus ou moins important. Par exemple, ajouter une VM (i.e. SO_{infra}) de type $small$ ou $large$ va impacter plus ou moins fortement les *critères*. De même, les différentes dimensions d'élasticité vont parfois agir sur les *critères* à des échelles de grandeur différentes. En ce sens, le Tableau 5.2 ne représente pas l'impact d'un point de vue quantitatif.

Tableau 5.2 – Actions d'élasticité et critères d'adaptation : tendances.

Actions d'élasticité	**Critères d'adaptation**					
	coût	QoS	QoF	énergie	temps reconf.	réactivité
Scale Out Infrastructure (SO_{infra})	$-$	$+$	$\emptyset$	$-$	$-$	$-$
Scale In Infrastructure (SI_{infra})	$+$	$-$	$\emptyset$	$+$	$*$	$*$
Scale Up Infrastructure (SU_{infra})	$-$	$+$	$\emptyset$	$-$	$+$	$+$
Scale Down Infrastructure (SD_{infra})	$+$	$-$	$\emptyset$	$+$	$+$	$+$
Scale Out Software (SO_{soft})	$\emptyset$	$-$	$+$	$-$	$*$	$*$
Scale In Software (SI_{soft})	$\emptyset$	$+$	$-$	$+$	$*$	$*$
Scale Up Software (SU_{soft})	$\emptyset$	$-$	$+$	$-$	$+$	$+$
Scale Down Software (SD_{soft})	$\emptyset$	$+$	$-$	$+$	$+$	$+$

Nous considérons dans le Tableau 5.2 l'impact direct des actions sur les *critères*. On remarquera d'ailleurs que l'on retrouve le symbole $\emptyset$ concernant respectivement les *critères* de *QoF* et de *coût* pour les actions IaaS et SaaS. On notera aussi que les *critères coût* et *énergie* sont étroitement liés c'est-à-dire qu'ils admettent les mêmes tendances pour les actions IaaS (i.e. coût de l'infrastructure lié à la consommation énergétique sous-jacente). De même, on peut observer que les *critères temps de reconfiguration* et *réactivité* présentent les mêmes tendances pour toutes les actions de base. Néanmoins, nous verrons dans la suite de cette thèse que ces deux *critères* permettent de discriminer certaines adaptations lorsque l'on considère de multiples actions figurant dans un même plan de reconfiguration (cf. sections 6.2 et 7.4).

Pour rappel, nous considérons dans notre travail le dimensionnement vertical de l'infrastructure (i.e. VS_{infra}) à chaud (i.e. sans interruption de service) et sans duplication de VMs, qui admet une reconfiguration rapide (cf. $+$ pour les *critères temps de reconfiguration* et *réactivité* pour les actions SU_{infra} et SD_{infra}). En effet, en pratique, certains fournisseurs IaaS proposent un "faux" dimensionnement vertical qui s'appuie en réalité sur le dimensionnent horizontal (i.e. HS_{infra}). Cette approche consiste à démarrer une nouvelle instance (i.e. SO_{infra}) avec une Off_{infra} distincte, de migrer l'état de la VM source vers cette nouvelle VM cible puis d'arrêter la VM source (i.e. SI_{infra}). Dans ce cas de figure, la mise en place d'un SU_{infra} (à froid) admettrait un temps de reconfiguration (et de réactivité) de l'ordre du SO_{infra}.

On remarquera que le symbole $*$, indiquant une tendance variable sur un *critère*, n'est utilisé que dans le cas des *critères temps de reconfiguration* et *réactivité*. Cela s'explique du fait que le résultat de l'adaptation sur ces *critères* est fortement lié à l'application considérée, à l'architecture de celle-ci ou encore aux composants qui la constituent. Prenons l'exemple d'une application respectant une architecture 3-tiers (i.e. présentation, métier et accès aux données), la *réactivité* ou le *temps de reconfiguration* de l'action SI_{infra} vont être variables s'il s'agit d'arrêter une VM du *tier base de données*, qui va nécessiter des traitements importants et chronophages (e.g. réplication des données, etc.), que s'il s'agit d'une VM du *tier présentation*, qui est généralement sans état (i.e. *stateless*) et peut potentiellement être retirée instantanément.

De même, les actions SO_{soft} et SI_{soft}, en fonction du type de composant concerné ou des traitements effectués, peuvent admettre une *réactivité* et des *temps de reconfiguration* variables. Par exemple, le SO_{soft} peut simplement correspondre à l'activation d'un composant (e.g. démarrer un nouveau serveur d'application *Tomcat*) ou alors à son téléchargement, puis son installation, suivie de son paramétrage, son initialisation, etc. De la même façon, le SI_{soft}, suivant le type de composant, peut nécessiter des traitements variables (e.g. désactiver le composant, le désinstaller, attendre la fin de sessions d'utilisateurs, etc.). En revanche, tout comme la dimension d'élasticité VS_{infra} (i.e. actions SU_{infra} et SD_{infra} qui reviennent à changer l'Off_{infra} de VM), on remarquera les tendances positives pour ces deux *critères* en ce qui concerne la nouvelle dimension d'élasticité VS_{soft} (i.e. actions SU_{soft} et SD_{soft}) qui revient à changer l'Off_{soft} de composants existants.

Nous verrons plus tard dans ce manuscrit que notre travail offre les moyens à l'administrateur humain d'intégrer ces subtilités liées à l'architecture de l'application et de les considérer dans la définition des adaptations possibles. Ainsi, l'administrateur va par exemple pouvoir définir qu'il souhaite privilégier le HS_{infra} pour le *tier présentation*, tandis qu'il préfère avoir recours au VS_{infra} pour le *tier base de données*, qui nécessite un effort de reconfiguration moindre (e.g. pas de réplication de données).

Compromis de l'élasticité multi-couche

Le positionnement des actions d'élasticité selon les différents *critères d'adaptation* (cf. Tableau 5.2) laisse apparaître des compromis de différents types.

Compromis intra-action : il faut souligner le fait qu'aucune action d'adaptation n'est idéale selon tous les *critères*. En effet, une action va être bénéfique selon certains *critères* mais néfastes pour d'autres. Nous parlons ainsi de compromis *intra-action* pour désigner le fait que le gains sur certains *critères* se font au dépend des autres. En ce sens, suivant le point de vue considéré (i.e. *critère*), une même action peut être évaluée comme avantageuse ou dommageable. Par exemple, l'action SU_{infra} impacte favorablement les *critères* de *QoS*, de *temps de reconfiguration* et de *réactivité* (cf. $+$) bien qu'elle soit désavantageuse selon le *coût* et l'*énergie* (cf. $-$). Les compromis *intra-action* résultent de compromis entre *critères* qui évoluent dans des sens contraires. On voit notamment apparaître dans le Tableau 5.2 le compromis *QoS-QoF* évoqué précédemment (cf. Figure 5.1) concernant l'*élasticité logicielle* (cf. tendances toujours opposées). De même, le fait de réduire la quantité de ressources IaaS et donc les *coûts* (i.e. impact positif) va généralement impacter négativement la *QoS* et réciproquement.

Compromis inter-action : on peut globalement regrouper les actions d'élasticité en deux catégories. La première catégorie regroupe les actions répondant à un manque de ressources (i.e. besoin de ressources supplémentaires) tandis la seconde catégorie correspond aux actions visant à réduire la quantité de ressources (i.e. ressources sous-utilisées). Dans le cas de l'élasticité de l'infrastructure, la première catégorie va concerner les actions SO_{infra} et SU_{infra} tandis que la seconde catégorie va concerner le SI_{infra} et le SD_{infra}. Ainsi, face à un besoin de ressources supplémentaires (ou de ressources sous-utilisées), deux actions d'adaptation peuvent être appliquées (unitairement ou ensemble). Bien que répondant à un même besoin général, celles-ci ne sont pas pour autant équivalentes et admettent des tendances différentes selon les *critères*. En ce sens, nous parlons de compromis *inter-action* pour désigner les gains et les pertes résultant du choix d'une action d'élasticité par rapport à une autre.

Exemple : la Figure 5.3 donne une illustration de la notion de compromis associés aux actions d'élasticité. L'exemple donné ici correspond à une application web mettant en œuvre l'*élasticité logicielle verticale* (i.e. dimension VS_{soft} et les actions associées SU_{soft} et SD_{soft}) au travers d'un service fournissant de la publicité selon différents formats (cf. *Text, Image, Video* et *Video HD*). Les différentes Off_{soft} qui en résultent sont plus ou moins consommatrices de ressources IaaS. L'infrastructure sous-jacente, elle aussi élastique, repose sur l'élasticité horizontale (i.e. dimension HS_{infra} et les actions associées SO_{infra} et SI_{infra}). Pour des raisons de lisibilité, seul trois *critères* sont présentés sur cette figure (cf. *coût*, *QoS* et *QoF*). Les tendances (cf. + et -) sont présentées en rouge pour chacun des axes correspondants aux *critères*.

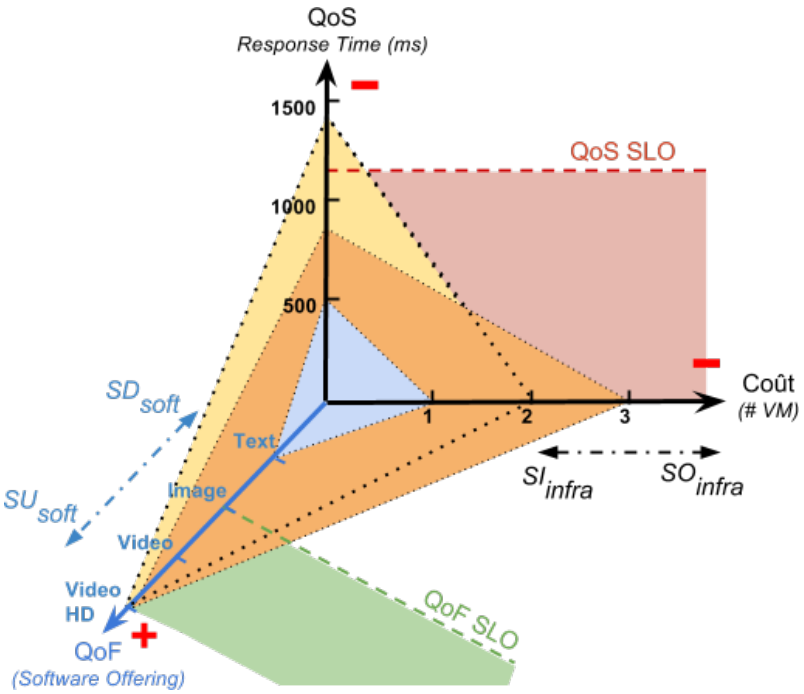


FIGURE 5.3 – Illustration des compromis : actions et critères.

On retrouve l'axe *QoF*, en bleu sur la figure, associé à l'*élasticité logicielle verticale* (i.e. proposition de la thèse). Plus l'Off_{soft} est importante, plus l'impact sur ce *critère* est positif (i.e. *QoF* supérieure). En ce sens, il est possible d'agir directement sur la *QoF* en faisant appel aux actions SU_{soft} et SD_{soft}. De même, les actions de dimensionnement horizontal de l'infrastructure permettent d'agir directement sur le *critère coût*. Plus la quantité de ressources IaaS est importante (i.e. nombre de VMs), plus le *coût* associé est élevé (i.e. impact négatif sur le *critère*). Enfin, le dernier axe *QoS* considère la métrique *response time* qui va être impactée par les 4 actions d'adaptation considérées.

La Figure 5.3 laisse apparaître 3 configurations (cf. zones bleue, orange et jaune). La première configuration, en bleue, correspond à une infrastructure et une *QoF* minimum (cf. 1 unique VM et $Off_{soft} = Text$). Le résultat de cette configuration offre de bonnes performances en termes de *QoS* (cf. $responseTime = 500ms$). Les configurations jaune et orange correspondent à la même Off_{soft}, à savoir la publicité fournie sous forme de *Video HD*. Néanmoins, les configurations IaaS diffèrent (i.e. nombre de VMs) ce qui entraine des *coûts* différents mais aussi des performances distinctes (cf. $1400ms$ avec 2 VMs contre $800ms$ avec 3 VMs). On voit alors apparaître le compromis "classique" de l'élasticité de l'infrastructure entre la quantité de ressources IaaS utilisée et les performances du système (i.e. compromis *intra-action*).

Considérons désormais un *SLO* portant sur les performances du système statuant que les temps de réponse des requêtes doivent être inférieurs à $1200ms$ (cf. *QoS SLO* en rouge). Dans le cas de la configuration jaune violant ce *SLO*, une adaptation possible est de rajouter une VM (i.e. SO_{infra}) et ainsi se retrouver avec la configuration orange qui admet certes des coûts supérieurs mais permet d'assumer le *SLA* signé. Cependant, en considérant l'*élasticité logicielle verticale*, une autre adaptation possible revient à diminuer la *QoF* de l'application (e.g. en passant de l'Off_{soft} *Video HD* à *Image*), ce qui va soulager les ressources IaaS et impacter positivement la *QoS*. La *QoF*, comme la *QoS*, peut être sujette à des limitations spécifiées dans des *SLOs* restreignant par exemple l'utilisation des différentes Off_{soft} (cf. *QoF SLO* en vert indiquant que l'on ne peut pas utiliser l'Off_{soft} minimum *Text*). En considérant les différentes dimensions d'élasticité, plusieurs adaptations sont possibles. Ainsi, dans le cas de la configuration jaune, si l'on souhaite respecter le *QoS SLO*, on peut faire appel au SO_{infra} ou au SD_{soft} (i.e. compromis *inter-action*) qui vont impacter distinctement les *critères* du système (i.e. *coût*, *QoF*, *QoS*, etc.). Néanmoins, il est aussi possible d'envisager de faire appel aux deux actions dans une même reconfiguration.

Bilan : La problématique de choisir la bonne action d'élasticité a été étudiée dans le cadre de l'élasticité de l'infrastructure. [AGN+13] se sont notamment intéressés à évaluer les avantages et les inconvénients des actions de passage à l'échelle vers le haut (i.e. SO_{infra} et SU_{infra}) tandis que [SHRE13] se sont penchés sur les compromis induits par les deux dimensions d'élasticité de l'infrastructure (i.e. HS_{infra} et VS_{infra}). Le fait de considérer l'*élasticité logicielle* et les nouvelles actions associées étend les possibilités d'adaptation du système. Prenons l'exemple d'un besoin de ressources supplémentaires pouvant être résolu au niveau IaaS en appliquant les actions SO_{infra} et SU_{infra}. L'*élasticité logicielle* va offrir la possibilité d'adapter les ressources SaaS/PaaS en vue de répondre à ce besoin au travers des actions SI_{soft} (i.e. retrait de composant(s) logiciel(s)) et SD_{soft} (i.e. réduire l'Off_{soft} de composant(s) et/ou de l'application) dans le but de soulager les ressources IaaS sous-jacentes. Ainsi, que ce soit face à un problème de sur-dimensionnement ou de sous-dimensionnement du système, 4 types d'actions sont envisageables à savoir une action possible par dimension d'élasticité (i.e. HS_{infra}, VS_{infra}, HS_{soft} et VS_{soft}). Le choix d'une action plutôt qu'une autre va toujours faire l'objet d'un compromis (*inter-action*) où l'on va certes gagner selon un (ou plusieurs) *critère(s)* mais au détriment d'un (ou plusieurs) autre(s).

Quelques rares travaux scientifiques se sont intéressés à la reconfiguration des applications à des fins de passage à l'échelle [KMÅHR14] [NKHR14] [GCH+04]. Bien qu'il ne s'agisse pas d'*élasticité logicielle* à proprement parlé, ces travaux ont parfois tenté de comparer les avantages et les inconvénients de ce type d'adaptation (majoritairement architecturale) face à l'élasticité de l'infrastructure [GSC09] [CGS09]. Néanmoins, comme pour les travaux liés à l'élasticité de l'infrastructure, il s'agit généralement de confronter deux actions entre elles dans le but d'identifier la meilleure action dans telle ou telle situation (i.e. en fonction du type de *workload*, du type d'application, etc.). C'est notamment le cas de [CGS06] qui compare deux adaptations s'apparentant aux actions SO_{infra} et SD_{soft}. En ce sens, les actions d'adaptation, quelle qu'elle soit, sont toujours considérées de manière unitaire dans les travaux. À notre connaissance, aucun travail scientifique et aucune solution industrielle considère la possibilité d'utiliser conjointement différentes actions d'élasticité (multi-couche ou non) dans une même adaptation du système. La décision d'adaptation se résume toujours à un choix parmi les actions d'élasticité considérées.

Synergie des actions d'élasticité

Les avantages et inconvénients de chaque dimension d'élasticité nous ont amenés à considérer la complémentarité des actions qui en découlent. En effet, une proposition de notre travail est de combiner les actions d'élasticité complémentaires en vue de lisser et éventuellement masquer leurs défauts respectifs. En ce sens, nous comptons tirer partie de la synergie possible entre les actions d'élasticité plutôt que de les considérer de manière unitaire. Cette proposition trouve tout son sens dans le contexte de l'élasticité multi-couche dans lequel on espère utiliser conjointement les deux types d'élasticité (IaaS et SaaS) pour bénéficier des avantages de chacun. Il s'agit alors de choisir judicieusement les actions d'élasticité à mettre en œuvre et de coordonner leur exécution, sous forme de plans de reconfiguration, pour en tirer les meilleurs bénéfices. En ce sens, le terme "synergie" définissant la "mise en commun de plusieurs actions concourant à un effet unique et aboutissant à une économie de moyens" [syn15] semble adapté.

L'analyse du Tableau 5.2 laisse déjà présager de certaines complémentarités entre actions. Il s'agit d'exploiter les différences entre les actions en termes d'impact sur les *critères*. Ces différences peuvent s'avérer être une richesse et non une contrainte comme c'est le cas lorsque l'on tend à limiter les possibilités d'adaptation à un choix exclusif entre des actions (i.e. compromis *interaction*). L'objectif de l'utilisation conjointe d'actions d'élasticité vise à améliorer la *réactivité* de l'adaptation (cf. sous-section 4.3.1) ainsi que sa *précision* (cf. sous-section 4.3.2). Par exemple, les actions SO_{infra} et SD_{soft} répondent toutes les deux à un problème de sous-dimensionnement des ressources mais admettent des tendances différentes pour certains *critères* (cf. *réactivité* et *énergie*). On peut ainsi envisager d'utiliser ces actions conjointement dans une même reconfiguration du système. Cela permettrait de soulager rapidement les ressources (cf. SD_{soft}) le temps du passage à l'échelle horizontal de l'infrastructure (cf. SO_{infra}) dont la mise en place est généralement longue (i.e. plusieurs minutes). Cet exemple de synergie permet notamment d'améliorer la *réactivité de l'adaptation*. Un autre exemple concerne les actions SI_{infra} et SD_{soft} qui admettent des tendances opposées concernant le *critère QoS*. Il peut être intéressant lors du retrait d'une VM qui va potentiellement impacter négativement la *QoS* (i.e. SI_{infra}), de réduire l'Off_{soft} et donc la *QoF* de l'application (i.e. SD_{soft}). Cela permet de contrebalancer le retrait de ressources IaaS et ainsi limiter l'impact du SI_{infra} sur la *QoS*. Ce second exemple de synergie permet d'ajuster plus finement le système et donc d'améliorer la *précision de l'adaptation*.

5.1.4 Modélisation

Informatique autonome : préférences de l'administrateur et aide à la décision

La complexité d'un système Cloud et l'environnement dynamique dans lequel il évolue rend la tâche d'administration ardue voir impossible pour un être humain. Pour cette raison, nous reposons sur l'informatique autonome [KC03] qui consiste à rendre le système capable de s'auto-administrer. Nous nous intéressons dans notre cas à automatiser le processus de dimensionnement automatique des ressources Cloud au sens large (i.e. IaaS, PaaS et SaaS). Bien que l'on souhaite rendre le système autonome et capable de s'auto-administrer, le rôle de l'administrateur humain reste essentiel. L'une des caractéristiques indispensables d'un système dit *autonomique* est de "comprendre les intentions des humains" [Hor01]. Cela implique de fournir à l'administrateur humain les moyens de transmettre ses intentions au système.

Dans notre contexte qu'est l'élasticité multi-couche du Cloud, les différentes ressources considérées peuvent faire l'objet d'adaptations au travers des actions d'élasticité présentées dans le Tableau 5.1. Dans le cas d'un système Cloud complet évoluant dans un environnement dynamique et comprenant de multiples ressources, le nombre de reconfigurations possibles peut s'avérer très important. Nous souhaitons ainsi offrir à l'administrateur humain les moyens de définir ses préférences d'adaptation, sous forme de politiques de haut niveau, qui vont lui permettre de spécifier de manière intelligible comment il souhaite résoudre les compromis induits par les différents *critères* évoqués précédemment. Ces politiques de haut niveau, que nous appellerons *stratégies d'élasticité* dans la suite de ce manuscrit, vont orienter le système dans sa prise de décision d'adaptation.

Décision d'adaptation de l'élasticité multi-couche : un problème multi-critère

Les multiples adaptations possibles résultant des actions d'élasticité considérées (cf. sous-section 5.1.2) et les *critères d'adaptation* impactés identifiés (cf. sous-section 5.1.3) nous ont amené à considérer les travaux et outils autour des problèmes d'optimisation combinatoires et plus précisément d'*aide multi-critère à la décision* [RV89] [Ehr13].

Une grande partie des problèmes d'optimisation combinatoire se résume à l'optimisation d'un unique *critère* (souvent économique et représenté par une fonction de coût que l'on souhaite minimiser) tout en satisfaisant un ensemble de contraintes diverses et variées (e.g. limites matérielles, contraintes de *QoS*, etc.). L'optimisation dite *mono-critère* est adaptée si le problème admet un point de vue unique (ou prédominant) ou si l'on a affaire à de multiples points de vue non conflictuels. Néanmoins, face à certains problèmes admettant plusieurs objectifs variés et contradictoires comme c'est le cas dans nos travaux, on souhaite parfois trouver une solution représentant un "bon compromis" du fait que la solution idéale (i.e. optimale pour tous les objectifs) n'existe pas.

En considérant les *critères* qui sont les nôtres, la solution idéale (i.e. optimale) reviendrait à trouver une adaptation pour le système aboutissant à une configuration admettant des *coûts* et une *consommation énergétique* minimum avec une *QoS* et une *QoF* maximum et dont la mise en œuvre serait instantanée. Une telle solution n'existe pas du fait des tendances variables des *critères* (i.e. *critères* conflictuels). Il va donc falloir faire des compromis ce qui va revenir à privilégier certains objectifs (e.g. minimiser les *coûts*) plutôt que d'autres (e.g. maximiser la *QoF*).

Modélisation multi-critère

Nous allons exploiter les résultats obtenus en *aide multi-critère à la décision* sur la modélisation des préférences d'une personne, appelée décideur. Dans notre cas, l'administrateur Cloud va jouer le rôle de décideur chargé de définir ses préférences en termes d'adaptation selon un ensemble de *critères*. Ses préférences vont être modélisées puis utilisées par un algorithme d'optimisation combinatoire chargé de rechercher une solution préférée selon ce modèle. On parle alors d'approche *a priori* [MA04]. Un modèle d'*aide à la décision* est composé de deux sous-modèles :

— *modèle des actions* : il s'agit de modéliser l'ensemble des actions potentielles A (i.e. adaptations). Nous sommes confrontés à une problématique de *choix* où l'on va chercher à identifier la meilleure solution parmi les actions envisageables selon les préférences établies au préalable. Le modèle d'actions peut être défini *explicitement* du fait que l'on connait la liste exhaustive des adaptations candidates (cf. *tactiques d'élasticité*, sous-section 5.2.2).

— *modèle des préférences* : un *critère* doit permettre de mesurer les préférences du décideur vis-à-vis de l'ensemble d'actions A, relativement à un point de vue. Un *critère* est représenté par une fonction $g : A \to X \subset \mathbb{R}$ qui permet de comparer 2 actions selon un certain point de vue. Ainsi, l'expression $g(a) \geq g(b) \implies aSb$ indique que *l'action a est au moins aussi bonne que (cf. S) l'action b* pour le *critère* représenté par la fonction g que l'on cherche ici à maximiser. De plus, un *critère* est souvent utilisé de la façon suivante (i.e. *vrai-critère* [Bou90]) où l'expression $g(a) > g(b) \iff aPb$ indique une *préférence stricte* (cf. P) de a sur b selon le *critère* porté par g.

Agrégation lexicographique

Dans le cas où plusieurs actions d'adaptation sont envisageables et afin de résoudre les compromis induits face aux *critères*, nous proposons la notion de *stratégie d'élasticité* permettant à l'administrateur d'exprimer des préférences de haut niveau venant orienter la prise de décision quant aux choix d'adaptation du système. Pour rappel, nous nous trouvons face à une problématique de *choix*, où nous souhaitons sélectionner la meilleure action à entreprendre. La sélection est basée sur une comparaison des *critères* suivant un ordre lexicographique des objectifs (i.e. méthode lexicographique [RTL96] [MA04]).

Prenons 2 actions a et b et 3 *critères* $c1$, $c2$ et $c3$ respectivement représentés par les fonctions f, g et h. Une *stratégie d'élasticité* va revenir à trier les *critères* par ordre décroissant d'importance selon les préférences de l'administrateur (i.e. du plus décisif au moins décisif). Dans notre cas, on se donne un ordre total sur les *critères* (i.e. classement sans ex-aequos). On peut ainsi représenter une *stratégie d'élasticité* sous la forme suivante : $S1 = c1 > c2 > c3$. En considérant la *stratégie* $S1$, si l'on peut identifier la meilleure action entre a et b seulement au regard du premier *critère* $c1$ (i.e. le plus important) c'est-à-dire $f(a) > f(b) \implies aPb$ ou $f(b) > f(a) \implies bPa$, alors on ne tient pas compte des autres *critères*. Sinon, c'est que ces actions sont indiscernables au regard de $c1$. On cherche alors à trancher entre celles-ci à l'aide du second *critère* $c2$, à savoir $g(a) > g(b) \implies aPb$ ou $g(b) > g(a) \implies bPa$, etc.

Nous verrons plus tard dans ce manuscrit que la décision d'adaptation ne revient pas uniquement à comparer les actions d'élasticité mais peut concerner des plans de reconfiguration complets (i.e. nommés *tactiques d'élasticité*) faisant parfois intervenir de multiples actions (cf. section 6.2). Nous montrerons alors le moment venu comment le concept de *stratégie d'élasticité* reste adapté à ce choix de la meilleure adaptation (et non de la meilleure action). De plus, nous verrons que la décision d'adaptation ne résulte pas uniquement des préférences de l'administrateur (cf. chapitre 7). En effet, avant de prendre en considération les préférences de l'administrateur (i.e. *stratégies*), la prise de décision d'adaptation doit considérer les contraintes imposées par le contexte d'exécution (e.g. état des ressources) réduisant le champ des possibles.

5.2 Gestion autonome de l'élasticité multi-couche

Dans cette section, nous allons tout d'abord définir notre modèle de ressources Cloud à savoir la représentation du système administré (i.e. *managed system*), prenant la forme d'un graphe de ressources de différents types. Puis, nous allons présenter les premières briques conceptuelles de notre *framework* de gestion de l'élasticité multi-couche *ElaStuff* qui repose sur un gestionnaire autonomique (i.e. *autonomic manager*) de type *MAPE-K* [KC03].

5.2.1 Système administré : modèle des ressources Cloud

Dans le cadre de nos travaux, nous nous intéressons à l'adaptation des ressources Cloud au sens large. Il s'agit de considérer les différents types de ressources propres aux couches du Cloud (i.e. IaaS, PaaS, SaaS). La Figure 5.4 donne un aperçu, sous forme de diagramme de classe *UML* [uml15], des ressources considérées dans nos travaux. On peut voir sur le diagramme de nombreuses relations de composition (e.g. une PM est constituée de VM) qui laisse présager une hiérarchie de ressources. De plus, on peut remarquer la présence d'une ressource *Tier*. En effet, nous nous sommes intéressés, dans nos travaux, à des applications respectant une architecture multi-tier. Contrairement aux relations "physiques" entre les autres ressources (e.g. une VM est contenue dans une PM), il s'agit ici d'une organisation logique (i.e. une application est constituée de tiers eux mêmes constitués de VMs). Notre modèle ne tend pas à répertorier de manière exhaustive toutes les ressources possibles d'un système Cloud. En effet, on pourrait par exemple ajouter les grappes de serveurs (i.e. *clusters*), le CPU, la RAM ou encore les conteneurs logiciels, etc.

La Figure 5.5 est une instance de notre modèle sous forme de graphe dans lequel les nœuds représentent les ressources Cloud et les arcs symbolisent les relations de composition entre celles-ci. On distingue deux types d'arcs, avec trait plein ou hachuré, qui permettent respectivement de distinguer l'architecture dite *physique* (e.g. relation entre PMs et VMs) de celle dite *logique* (i.e. relations entre l'application, les tiers et les VMs). Ainsi, une VM (e.g. *VM3*) est à la fois rattachée à une PM (e.g. *PM2*) et à un Tier (e.g. *Tier2*). Nous verrons par la suite lors de la description de notre *framework* qu'une telle instance du modèle correspond au système dont on attend un comportement autonomique (i.e. *managed system*).

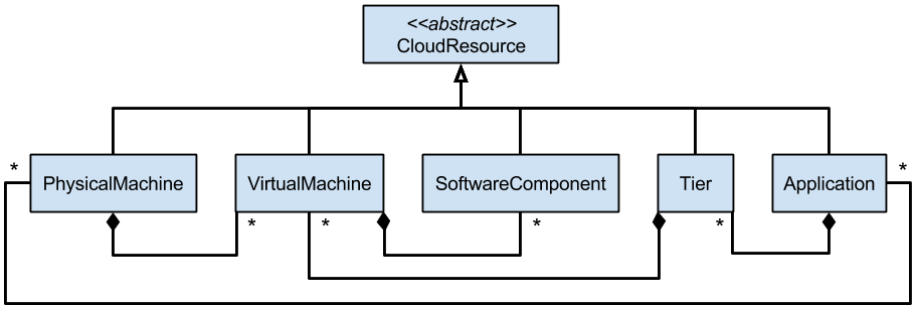


FIGURE 5.4 – Modèle des ressources Cloud.

Les adaptations apportées au système au travers des actions d'élasticité (cf. Tableau 5.1) vont revenir à modifier ce graphe de ressources. Ainsi, les dimensions d'élasticité HS_{infra} et HS_{soft} vont changer la structure de ce graphe en ajoutant (resp. SO_{infra} et SO_{soft}) ou en retirant (resp. SI_{infra} et SI_{soft}) respectivement des nœuds de type *VM* et *Composant* (i.e. modification de l'architecture). En revanche, les dimensions d'élasticité VS_{infra} (i.e. actions SU_{infra} et SD_{infra}) et VS_{soft} (i.e. actions SU_{soft} et SD_{soft}) ne vont pas modifier la structure du graphe mais simplement la valeur des attributs/paramètres des nœuds existants à savoir l'Off_{infra} de nœud *VM* et l'Off_{soft} de nœud *Composant* (i.e. paramétrisation).

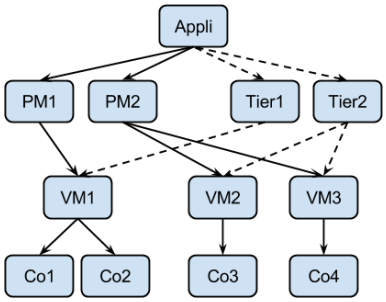


FIGURE 5.5 – Instance du modèle de ressources Cloud.

Le fait de représenter notre modèle de ressources à l'aide d'un graphe offre plusieurs avantages. En effet, comme nous le verrons plus tard, cette représentation permet de considérer les relations entre ressources (i.e. parenté entre nœuds) et ainsi profiter de la structure du graphe aussi bien pour la surveillance et la reconfiguration des ressources que pour la prise de décision d'adaptation.

En ce qui concerne notre modèle de ressources Cloud, nous suivons une approche de *modélisation à l'exécution* pour laquelle notre système administré (i.e. graphe de ressources) constitue un *model@runtime*, défini comme suit par *Blair et al.* [BBF09] :

> « *a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective [BBF09].* » (Gordon Blair et al. - 2009)

On retrouve le terme *lien causal bidirectionnel* dans la littérature [Bar14] pour désigner la relation entre le *model@runtime* (i.e. modèle réflexif [Mae87]) et le système concret. Cela induit que le modèle est à jour par rapport à l'état du système et réciproquement (i.e. représentation fidèle). Néanmoins, le *model@runtime* peut être modifié indépendamment du système administré (i.e. de manière *asynchrone*), puis synchronisé ultérieurement, ce qui permet notamment d'effectuer des étapes de vérification sur le modèle avant d'adapter "irréversiblement" le système réel ou encore de s'abstraire de certaines contraintes liées à celui-ci [Fou13].

Notre modèle constitue ainsi une représentation simplifiée (i.e. abstraction) d'un système Cloud réel, qui en contient les aspects fondamentaux (i.e. architecture, contexte d'exécution, *sensors*, *actuators*, etc.) tout en permettant d'interagir avec celui-ci (cf. Figure 5.6).

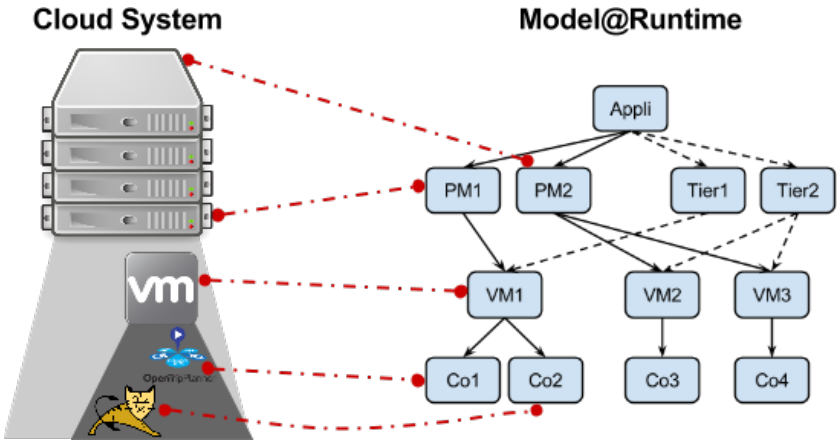


FIGURE 5.6 – Système Cloud et *model@runtime*.

5.2.2 Gestionnaire autonomique

Nous allons présenter l'implémentation de notre gestionnaire autonomique (*autonomic manager*) pour la gestion de l'élasticité multi-couche prenant la forme d'un *framework* autonomique de type *MAPE-K*. Il s'agit ici de présenter brièvement les différentes briques conceptuelles constituant notre *framework* et de rattacher celles-ci aux futurs chapitres de ce manuscrit.

Architecture générale du *framework* autonomique *ElaStuff*

La Figure 5.7 donne un aperçu de notre *framework* autonomique nommé *ElaStuff*. On y retrouve les différentes composantes du gestionnaire autonomique *MAPE-K* dans la partie haute tandis que la partie basse fait apparaître le système dont on attend un comportement autonomique, à savoir une instance de notre modèle de ressources Cloud (i.e. graphe). Les zones verte, rouge et bleue correspondent respectivement aux phases de *surveillance*, de *décision* et d'*adaptation* de notre *framework* autonomique. Bien que nous détaillerons les différents modèles sous-jacents plus tard dans ce manuscrit, nous allons donner ci-dessous quelques informations relatives à ces différentes phases.

Modèle de surveillance

La *surveillance* de notre *framework* autonomique correspond à la première phase de la boucle *MAPE-K*, appelée phase d'observation (*Monitoring - M*). En ce sens, la *surveillance* est en charge de la capture des informations exposées par les éléments gérés (i.e. nœuds du graphe de ressources) au travers des capteurs. Cette phase de collecte d'informations est essentielle car c'est par ce biais que le gestionnaire autonomique va avoir conscience de l'état de santé du système.

Notre modèle de *surveillance* repose sur un outil, que nous appelons *perCEPtion*. Il s'agit d'un *framework* basé sur le traitement des événements complexes (*Complex Event Processing - CEP*) permettant à l'administrateur Cloud de mettre en place une *surveillance* avancée du système en définissant en amont les *symptômes* (stockés dans le *K*), signes d'incohérence, pouvant nécessiter une adaptation à l'exécution. Nous reviendrons plus en détail sur notre modèle de *surveillance* ainsi que notre outil *perCEPtion* dans la section 6.1.

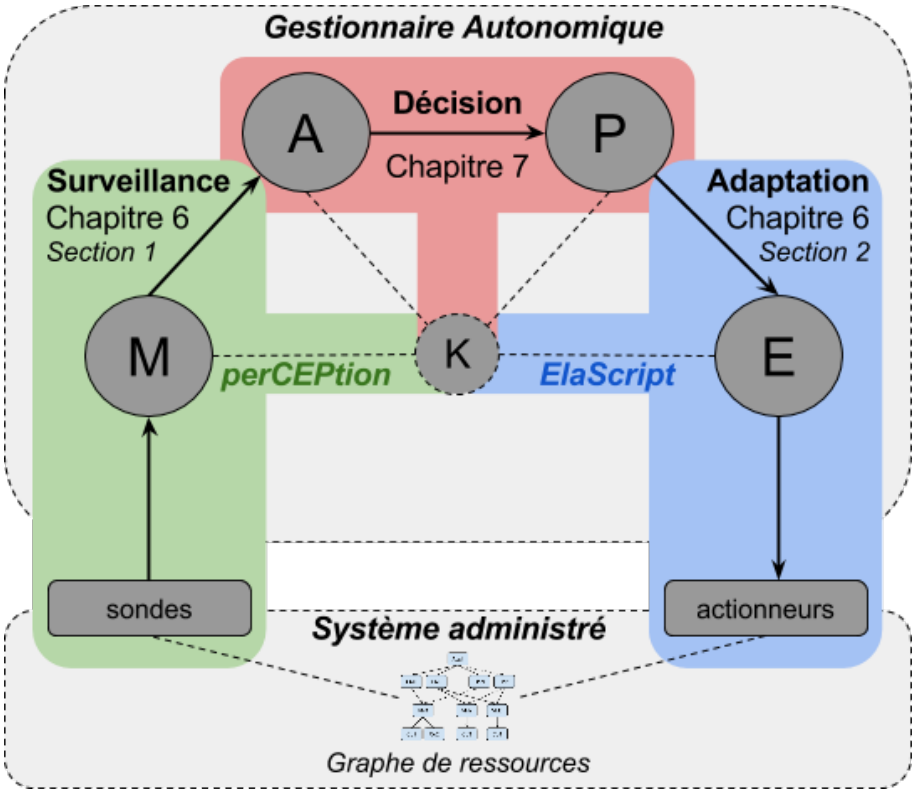


FIGURE 5.7 – *Framework* autonomique de gestion de l'élasticité : *ElaStuff*.

Modèle d'adaptation

L'adaptation du *framework* autonomique correspond aux dernières étapes de la boucle *MAPE-K*, à savoir la phase d'exécution (i.e. *Execute* - *E*) chargée d'appliquer le plan de reconfiguration issu de la phase de planification (i.e. *Plan* - *P*) sur le(s) élément(s) gérés (i.e. graphe de ressources).

Notre modèle d'adaptation repose sur la définition et l'exécution de *tactiques d'élasticité*. Nous définissons une *tactique d'élasticité* comme un plan de reconfiguration générique spécifiant l'adaptation à réaliser face à un type de *symptôme*. Il s'agit concrètement d'un *script* de reconfiguration portant sur une ou plusieurs ressources. Un tel *script* peut faire intervenir une ou plusieurs actions d'adaptation (de manière ordonnée) de différentes dimensions (i.e. HS_{infra}, VS_{infra}, HS_{soft} et VS_{soft}).

Une *tactique d'élasticité* va ainsi contenir un ensemble d'instructions ordonnées qui vont être traduites en appel(s) aux différents actionneurs (i.e. *actuators*) qui opèrent les modifications nécessaires sur le système. Afin de décrire efficacement de telles *tactiques d'élasticité*, potentiellement multi-couche, nous proposons un langage dédié (*DSL - Domain Specific Language* [Fow10] [MHS05]) nommé *ElaScript*.

ElaScript est un langage dédié à l'élasticité multi-couche permettant de définir simplement et de manière concise des plans de reconfiguration complexes sous forme de *scripts*. Ce langage offre la possibilité de composer et d'orchestrer les multiples actions d'élasticité via des opérateurs de séquentialité, parallélisme et synchronisation. L'ensemble des *tactiques d'élasticité* définies par l'administrateur va constituer le catalogue des adaptations possibles à entreprendre sur le système (stocké dans le *K*). Nous reviendrons plus tard dans ce manuscrit sur notre modèle d'*adaptation* ainsi que notre langage *ElaScript* (cf. section 6.2).

Modèle de décision

Le modèle de *décision* fait office de glu entre les activités d'*observation* (i.e. *perCEPtion*) et de *reconfiguration* (i.e. *ElaScript*). Ce dernier modèle est chargé de la prise de décision concernant les changements à apporter au système (i.e. adaptation).
Notre modèle de *décision* prend la forme d'un processus de décision constitué de trois étapes, représentées sous forme de *filtres* successifs et dont le point d'entrée est un *symptôme* remonté à l'exécution par notre modèle de *surveillance* (i.e. *framework perCEPtion*). Il s'agit, à partir de ce *symptôme* signe d'instabilité, de décider de l'adaptation à mettre en œuvre sur le graphe de ressources (i.e. système administré), en vue de retrouver un état stable.
La décision va revenir à choisir la meilleure *tactique* possible face à un *symptôme*, parmi les différentes *tactiques d'élasticité* définies au préalable par l'administrateur humain via le langage *ElaScript*. Nous reviendrons plus en détail sur les différentes phases du processus constituant notre modèle de décision d'adaptation dans le chapitre 7.

Base de connaissance

Les différentes phases de la boucle autonomique *MAPE-K* sont reliées à la base de connaissances (*Knowledge - K*). Pour rappel, cette base de connaissances peut contenir des informations diverses et variées (e.g. historiques de la phase d'observation, reconfigurations passées, politiques de gestion, etc.) et peut être peuplée automatiquement (e.g. par le gestionnaire autonomique) ou manuellement (e.g. administrateur humain).
Dans notre *framework* autonomique *ElaStuff*, le *K* va contenir des informations rajoutées par les administrateurs humains en vue d'aider le gestionnaire autonomique dans sa prise de décision. Cette configuration manuelle est transverse et concerne les modèles de *surveillance* (i.e. *symptômes*), de *décision* (i.e. *filtres*) et d'*adaptation* (i.e. *tactiques*). Ainsi, nous préciserons les interactions avec le *K* au fur et à mesure que nous balayerons les différentes briques de notre *framework* autonomique.

6

Outiller le processus de gestion de l'élasticité

Dans ce chapitre, nous allons nous intéresser au fait d'outiller le processus de gestion de l'élasticité en vue de simplifier la tâche d'administration de l'opérateur humain. Pour rappel, l'étude de l'état de l'art a permis d'identifier un manque d'outils concernant la gestion et le paramétrage de l'élasticité. Afin de jouir pleinement des bienfaits offerts par l'élasticité du Cloud, il est nécessaire d'industrialiser celle-ci au travers d'outils facilitant sa gestion. Pour ce faire, nous présentons dans ce chapitre deux outils permettant à l'administrateur Cloud de paramétrer aisément la surveillance et l'adaptation du système. Ces outils s'intègrent à notre *framework* autonomique global présenté dans la sous-section 5.2.2.

Le premier outil, que nous appelons *perCEPtion* (cf. section 6.1), consiste en un *framework* de surveillance (i.e. *monitoring*) basé sur le traitement des événements complexes (*Complex Event Processing - CEP*) permettant à l'administrateur Cloud de mettre en place une surveillance avancée du système en spécifiant les symptômes, signes d'incohérence, pouvant nécessiter une adaptation.

Le second outil, que nous présentons dans la section 6.2, est un *DSL* (*Domain Specific Language* [Fow10] [MHS05]) nommé *ElaScript*. Il s'agit d'un langage dédié à l'élasticité multi-couche permettant de définir simplement et de manière concise des plans de reconfigurations complexes. Ce langage de script offre la possibilité de composer et d'orchestrer les multiples actions d'élasticité via des opérateurs de séquentialité, parallélisme et synchronisation.

Contents

6.1 *perCEPtion* : un *framework CEP* pour la surveillance d'architecture Cloud

Dans cette section, nous présentons notre *framework perCEPtion*, une solution basée sur le traitement des événements complexes pour la surveillance avancée d'architectures Cloud. Nous motivons dans un premier temps l'intérêt de la solution, puis nous présentons le fonctionnement général de *perCEPtion* en détaillant les différents éléments conceptuels du *framework* tout en indiquant comment cet outil s'intègre dans la solution globale de gestion autonomique de l'élasticité *ElaStuff* abordée dans la sous-section 5.2.2. Enfin, nous abordons l'implémentation technique du *framework* et donnons quelques exemples des possibilités offertes par celui-ci.

6.1.1 Motivations

Les architectures Cloud deviennent de plus en plus complexes et font intervenir un grand nombre de ressources de différents types (i.e. PM, VM, composant logiciel, etc.), exposant elles-mêmes différentes métriques (i.e. CPU, bande passante, temps de réponse, etc.) qu'il est parfois nécessaire d'agréger et de recroiser afin d'identifier une dérive concernant la stabilité du système dans sa globalité.

L'objectif du *framework perCEPtion* est de permettre de définir des *symptômes* plus ou moins complexes, signes d'instabilité du système, nécessitant une éventuelle adaptation de celui-ci. Contrairement aux possibilités limitées offertes dans la définition des règles à base de seuils fournies par les services de dimensionnement automatique actuels tel que *Amazon Auto Scaling* [EC215a], le *framework* doit permettre de représenter et de définir des *symptômes* complexes, à granularité variable, en permettant notamment leur composition.

Les règles à base de seuils ne permettent pas de considérer l'architecture du système, c'est-à-dire les relations entre les ressources, qui constitue pourtant une source d'information importante. En effet, nous soutenons que la *spatialité* des ressources est à prendre en compte pour la définition de *symptômes* pertinents (i.e. corrélation entre ressources/métriques). Enfin, bien qu'il soit possible de considérer une fenêtre de temps dans la définition des règles (i.e. une métrique dépasse un seuil pendant une certaine durée), l'implémentation des règles à base de seuils peine à exprimer la notion de *temporalité* (e.g. spécifier qu'un événement s'est produit avant un autre).

Le *framework perCEPtion* vise à combler les limites évoquées ci-dessus en offrant des moyens de surveillance adaptés aux systèmes Cloud complexes qui évoluent dans des environnements dynamiques. Pour ce faire, *perCEPtion* tient compte de la *spatialité* des ressources ainsi que la *temporalité* des événements survenant à l'exécution. Le *framework* s'appuie sur le traitement des événements complexes (*Complex Event Processing - CEP*), qui permet d'effectuer le filtrage, la corrélation, l'agrégation, et le traitement d'événements (provenant de différentes sources et générés dans le temps) de manière *spatio-temporelle*.

6.1.2 Spécifications du *framework*

L'outil *perCEPtion* vise à outiller la phase de *surveillance* (i.e. *Monitoring* de *MAPE-K*) de notre *framework* autonomique de gestion de l'élasticité.

Fonctionnement général

Le but de *perCEPtion* est de permettre à l'administrateur humain de définir des *symptômes*, signes d'instabilité du système, puis de les identifier dynamiquement à l'exécution pour les remonter à la phase d'*Analyze* de la boucle *MAPE-K*, qui sera chargée de la décision d'adaptation à entreprendre face à ces *symptômes*. La Figure 6.1 présente l'architecture générale du *framework*.

Pour rappel (cf. sous-section 5.2.2), le système considéré est constitué de multiples ressources (i.e. graphe) qui exposent, au travers des sondes (*sensors*), un certain nombre de métriques représentatives de leurs états de santé. En ce sens, un *symptôme* peut concerner une (ou plusieurs) métrique(s) d'une (ou plusieurs) ressource(s) et ainsi admettre différents niveaux de granularité.

Nous avons évoqué que le *framework* reposait sur le traitement des événements complexes. En effet, pour répondre aux contraintes imposées par les systèmes évoluant dans des environnements hautement dynamiques dont les données nécessitent un traitement instantané, les informations récupérées à chaque relevé de sonde sont poussées sous forme d'événements (cf. *étape 1* Figure 6.1) dans un flux de données (i.e. *event streams* ou *data streams* [CM12] [SGB+13]).

Le point d'entrée du gestionnaire autonomique (i.e. *M* de *MAPE-K*) est un flux d'événements qui représente concrètement une séquence linéaire d'événements basiques (i.e. relevés de sondes), ordonnée par rapport à un axe temps, qu'il est possible d'interroger dynamiquement en vue d'identifier des *symptômes*, signes d'instabilité du système. Ainsi, définir un *symptôme* revient à définir une requête sur ce flux d'événements basiques dont le résultat sera un nouvel événement, pouvant lui même faire l'objet de requête (cf. *étape 2*). L'ensemble des définitions de requêtes CEP forme un catalogue de *symptômes* stocké dans la base de connaissance du gestionnaire autonomique (i.e.*Knowledge*).

Les événements générés, représentatifs de *symptômes* étant survenu à l'exécution, sont poussés dans une queue en attente de traitement (cf. *étape 3*). Comme nous le verrons par la suite, la queue de *symptômes* va faire l'objet d'un traitement (i.e. purge et tri dynamique), afin d'ordonner et prioriser les *symptômes* à traiter (cf. *étape 4*). Cette queue ainsi produite par la phase de *Monitoring* va alimenter la partie décision de notre gestionnaire autonomique à savoir la phase d'*Analyze* qui va défiler (cf. *étape 5*) de manière itérative les *symptômes* en tête de la queue pour les traiter (i.e. "soigner" le système en vue de retrouver un état "stable").

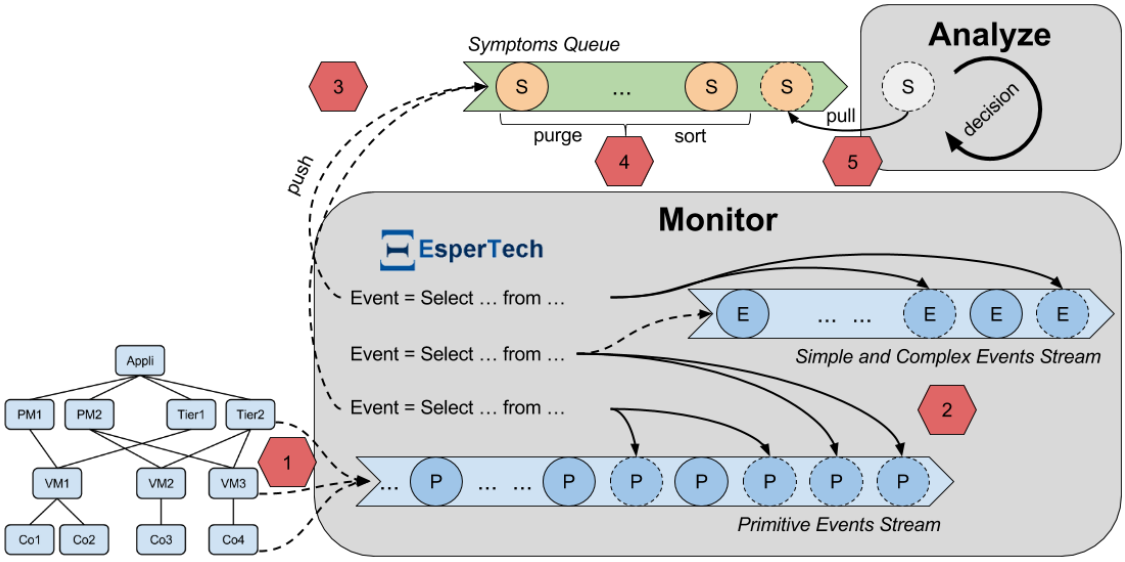


FIGURE 6.1 – *perCEPtion* : architecture générale du *framework*.

Types d'événements

Nous distinguons trois types d'événements admettant différents niveaux granularité :

— *événement primitif* : ce type d'événement correspond au relevé d'une sonde. Ainsi, un *événement primitif* porte une information correspondant à la valeur d'un seul attribut d'un unique nœud du graphe (i.e. valeur d'une métrique d'une ressource à un instant *t*). Ce type d'événement, ayant le plus fin niveau de granularité, ne constitue pas un *symptôme* mais une simple information que l'on peut éventuellement exploiter pour générer des événements porteurs de sens (i.e. *symptômes*) qui vont eux dénoter d'une "maladie" dans notre graphe de ressources. Un exemple d'*événement primitif* est le relevé de la sonde CPU d'une VM à un instant *t*. Les événements primitifs, notés *P* dans la Figure 6.1, sont poussés dans un flux qui va faire l'objet de requête (cf. *Primitive Events Stream*).

— *événement simple* : il s'agit d'un événement portant sur un unique nœud du graphe mais pouvant considérer plusieurs attributs (i.e. métriques). Par exemple, un *événement simple* peut correspondre au fait qu'une VM a vu sa consommation moyenne de CPU dépasser un certain seuil sur la dernière minute. Ce type d'événement, une fois généré, est poussé dans un flux d'événement (cf. *E* dans *Simple and Complex Events Stream*) pouvant lui-même faire l'objet de requêtes pour identifier des événements composites et davantage complexes. Notons qu'un *événement simple* peut aussi être considéré comme un *symptôme* à part entière nécessitant à lui seul une adaptation. Dans ce cas, l'*événement simple* généré est aussi poussé dans la queue de *symptômes* (cf. *étape 3*) en attente de traitement.

— *événement complexe* : ce type d'événement, contrairement à un *événement simple*, concerne plusieurs nœuds du graphe ressources. Un *événement complexe*, peut être défini par une requête portant sur le flux d'*événements primitifs* (cf. *Primitive Events Stream*) ou sur le flux d'événements déjà générés (cf. *Simple and Complex Events Stream*) en composant/agrégeant d'autres événements (i.e. simples ou complexes). Un exemple d'*événement complexe* est le fait que les temps de réponse moyens observés sur un tier dépassent un certain seuil et que plus de la moitié des VMs qui constituent ce tier sont surchargées en CPU (i.e. possiblement représenté par des *événements simples*) sur une certaine fenêtre de temps. Comme pour un *événement simple*, un *événement complexe*, une fois généré, est poussé dans le flux *Simple and Complex Events Stream*. De même, si celui-ci nécessite une adaptation, il est aussi poussé dans la queue de *symptômes* pour attente de traitement (cf. *étape 3*).

Des événements aux *symptômes*

Nous avons présenté les différents types d'événements considérés dans le *framework per-CEPtion*. Bien qu'ils apportent une information sur l'état du système, ils ne constituent pas nécessairement un besoin d'adaptation. Ainsi, nous introduisons le concept de *symptôme* comme un événement nécessitant une adaptation. Il s'agit concrètement d'un événement "auto-contenu" (*self contained*) indiquant une maladie que l'on souhaite analyser (i.e. transmettre au *A* de *MAPE-K*) pour la soigner. Le terme "auto-contenu" désigne ici le fait qu'un *symptôme* donne toutes les informations nécessaires pour permettre la mise en place d'un plan de reconfiguration.

Ressources à traiter : Un *symptôme* contient une collection de ressources malades qui désigne les ressources sur lesquelles agir. Cette collection peut contenir un unique élément si le *symptôme* correspond à un *événement simple*. Dans le cas d'un *événement complexe*, la collection de ressources va contenir plusieurs éléments. S'il s'agit d'un *événement complexe composite*, les ressources sont regroupées en sous-collections contenant les ressources retournées par les différents événements composés (i.e. regroupées par maladie).

Prenons par exemple 2 *événements simples ES1* et *ES2* qui représentent respectivement un tier admettant des temps de réponse trop élevés et une VM surchargée en CPU. Prenons maintenant un *événement complexe EC* nécessitant un besoin d'adaptation (i.e. considéré comme le *symptôme S1*). *EC* est défini comme la composition des événements *ES1* et *ES2* (avec lien de parenté), qui correspond au fait qu'un tier (e.g. *T1*) est atteint de *ES1* et que 2 de ses VMs (e.g. *VM1* et *VM2*) sont atteintes de *ES2*. Dans ce cas, la collection de ressources du *symptôme S1* sera *T1, VM1, VM2*.

Score d'un symptôme : Dans un environnement fortement dynamique, de nombreux *symptômes* peuvent survenir à l'exécution, empêchant le gestionnaire autonomique de tous les traiter (i.e. effectuer les phases *A*, *P* et *E* de *MAPE-K*). Il devient alors nécessaire de donner la priorité à certains *symptômes*. C'est pour cette raison que nous proposons de munir les *symptômes* d'un *score*, dont la valeur est calculée à l'exécution lorsque celui-ci est généré.

Le calcul du *score* d'un *symptôme* prend en compte deux aspects que sont le nombre de ressources impactées et leur type (e.g. Tier, VM, etc.). En effet, nous estimons que ces deux informations constituent un bon moyen de prioriser les *symptômes* les uns par rapport aux autres. Ainsi, nous donnons à chaque type de ressources un poids (pouvant être paramétré par l'administrateur Cloud) sous forme de valeur. Nous estimons que plus la ressource admet un niveau de granularité important, plus la valeur du poids doit être importante (i.e. intuitivement, une application "malade" est plus critique qu'un composant "malade").

Prenons par exemple les valeurs 1, 2, 3 et 4 correspondant respectivement aux poids des types de ressources *Composant*, *VM*, *Tier* et *Application*. Le *symptôme S1* décrit précédemment a alors un *score* = *7* (i.e. 2 VMs et 1 Tier impactés, $(2 \times 2) + (1 \times 3) = 7$). Il faut préciser que le *score* doit être calculé à la génération du *symptôme* (et non défini comme un *score* statique propre à chaque type de *symptôme*) du fait que le nombre de ressources concernées par un *symptôme* peut être variable (i.e. identifié *at runtime*).

Durée de vie d'un symptôme : Le fait qu'un *symptôme* ne puisse pas nécessairement être analysé dès qu'il survient soulève un autre problème qui concerne son obsolescence. En effet, un *symptôme* constitue une instabilité du système à un instant *t* et peut ainsi ne plus avoir de sens après un certain temps. Le temps de latence entre la génération d'un *symptôme* et son traitement par la phase d'*Analyse* est résolu par notre *framework* à travers les concepts de de *durée de vie* et de *péremption*. Chaque type de *symptôme* se voit attribuer une *durée de vie* pendant laquelle celui-ci a du sens. Cela permet notamment de distinguer des types de *symptômes* "fugaces" (e.g. temps de réponse d'un composant trop élevé qui n'a de sens que quelques secondes) ou au contraire "durables" (e.g. PM surchargée). Ainsi, lors de la génération d'un *symptôme*, sa date de *péremption* est calculée ce qui va permettre de maintenir un ensemble de *symptômes* d'"actualité" en purgeant de la *symptoms queue* ceux considérés comme obsolètes.

Afin d'illustrer le concept de *symptôme*, le Code 6.1 donne la définition d'une classe *Java MySymptom*. Pour des raisons de place et de lisibilité, certaines portions de code ont été omises et d'autres portions normalement présentes dans la classe abstraite mère (i.e. *Symptom*) ont été dupliquées. On retrouve les différents concepts abordés précédemment sous forme de variables d'instance (i.e. *resources*, *score*, *peremption*) ou de classe (i.e. *dureeDeVie*), l'initialisation de celles-ci au travers du constructeur ainsi que l'implémentation de la méthode *compareTo(Symptom s)* permettant de comparer deux *symptômes*.

```
// Classe MySymptom qui herite de la classe abstraite Symptom
// et qui implemente l'interface Comparable
public class MySymptom extends Symptom implements Comparable<Symptom> {
  // Collection de ressources impactees
  private Collection<Resource> resources;
  private int score = 0;
  private long naissance, peremption;
  // Duree de vie de 60sec pour tous les symptomes de type MySymptom
  static long dureeDeVie = 60000;

  // Constructeur prenant un evenement en parametre
  // et qui initialise les variables d'instance
  public MySymptom(Event event) {
    // Naissance = Timestamp en ms du moment courant
    naissance = System.currentTimeMillis();
    peremption = naissance + dureeDeVie;
    // Les ressources du symptome sont celles portees par l'evenement
    resources = event.getResources();
    // Score calcule en fonction du nb de ressources et de leur poids
    for (Resource r : resources) {
      score = score + r.getPoids();
    }
  }

  // Methode pour comparer deux symptomes (pour trier la queue)
  public int compareTo(Symptom s) {
    // On compare les scores
    int resCompScore = Integer.compare(this.getScore(), s.getScore());
    // En cas de scores egaux
    if (resCompScore == 0) {
      // On compare les naissances (moment de la generation)
      return Long.compare(this.getNaissance(), s.getNaissance());
    }
    return resCompScore;
  }
}
```

Code 6.1 – Portion de code *Java* illustrant la définition d'un *symptôme*.

Queue de symptômes : trier pour mieux traiter

L'interface entre les phases de *Monitoring* et d'*Analyze* de notre gestionnaire autonomique (i.e. boucle *MAPE-K*) se fait au travers de la queue de *symptômes* (i.e. *symptoms queue*) visible en vert sur la Figure 6.1. La phase de *Monitoring* est chargée de remonter tous les *symptômes* devant être analysés en les poussant (cf. *push*) dans la *symptoms queue*. La phase d'*Analyze* va traiter les *symptômes* un à un en dépilant (cf. *pull*) de manière itérative la tête de queue.

Une manière naïve de gérer cette queue (ou file) pourrait reposer uniquement sur le principe du premier entré, premier sorti *FIFO* (*First In, First Out*), pour lequel les premiers éléments (*symptômes*) ajoutés à la file seront les premiers à être récupérés et traités. Néanmoins, comme nous l'avons évoqué précédemment, la durée écoulée entre l'ajout d'un élément par la phase de *Monitoring* et son retrait par la phase d'*Analyze* peut être importante rendant ainsi le principe de *FIFO* peu pertinent du fait de l'obsolescence de certains *symptômes*. C'est pour cette raison que nous avons muni notre *framework* des concepts de *score* et de *péremption* afin de proposer une meilleure gestion de la queue de *symptômes*.

La queue de *symptômes* en attente de traitement est gérée grâce aux deux fonctions suivantes :

— **Purger** (cf. *purge*) les éléments (i.e. *symptômes*) de la queue en tenant compte de leur *durée de vie*. Chaque *symptôme*, une fois généré et ajouté à la queue, se voit attribuer une date de *péremption*. Il s'agit donc de purger de la queue tous les éléments ayant atteint leur date de *péremption* et ainsi éviter de traiter des *symptômes* obsolètes (cf. Algorithme 1). Cette étape de *purge* est effectuée avant chaque ajout (i.e. *push* du *Monitoring*) et retrait (i.e. *pull* de l'*Analyze*) dans la queue.

— **Trier** (cf. *sort*) les éléments de la queue en fonction de leur *score*. Pour ce faire, nous reposons sur un type de *queue à priorité* proposé par *Java* (i.e. *java.util.PriorityQueue<E>*) permettant de manipuler les éléments en fonction de leurs priorités de traitement. Chaque élément ajouté à ce type de queue est directement positionné à la bonne place en fonction de sa priorité. La détection des priorités au moment de l'ajout (et le tri qui en résulte) est réalisée en implémentant l'interface *java.lang.Comparable<T>* (cf. définition de la classe *MonSymptome* du Code 6.1). Ainsi, au moment d'enfiler un nouveau *symptôme* dans la queue, on va comparer son *score* à ceux des *symptômes* déjà présents dans celle-ci (cf. méthode *compareTo(Symptome s)*). En cas de *scores* identiques, la priorité est donnée au *symptôme* le plus récent (i.e. principe *LIFO*, *Last In, First Out*). L'ajout d'un nouvel élément dans la liste (et le tri qui en résulte) est précédé par la phase de *purge* décrite ci-dessus.

Algorithme 1 Purge de la *queue de symptômes*.

INPUT : Symptom Priority Queue to purge S and the current timestamp $current\_time$
OUTPUT : Symptom Priority Queue purged S

for each Symptom $symptom$ in S **do** :
  if $current\_time$ > $symptom$.peremption **then** :
    S.remove($symptom$) ;

6.1.3 Gestion des événements avec *Esper*

Notre *framework perCEPtion* repose sur le traitement des événements complexes (*Complex Event Processing - CEP*) et s'appuie sur un moteur de gestion de flux d'événement nommé *Esper* [esp15b] qui intègre un langage dédié : *EPL* (*Event Processing Language*). L'utilisation conjointe de notre modèle conceptuel de surveillance, de l'outil *Esper* et du langage *EPL* va permettre à l'administrateur Cloud de définir des *symptômes*, plus ou moins complexes, qui seront identifiés et remontés dynamiquement à l'exécution du système.

Esper

Esper [esp15b] est un moteur open-source (fourni par la société *EsperTech*) de gestion de flux d'événement (i.e. *Event Stream Processing*) ainsi qu'un moteur de corrélation d'événement offrant une implémentation *Java* et *.Net*. Le but d'*Esper* est de détecter des événements en temps réel et éventuellement de déclencher des actions lorsqu'un ou plusieurs événements se produisent. En ce sens, *Esper* permet la mise en place d'un moteur d'alertes (représentées sous forme d'événements) en temps réel qui peut servir un large panel de besoins.

L'idée derrière *Esper* n'est pas de stocker toutes les données surveillées (i.e. représentant une quantité d'informations trop volumineuse), et ensuite de les analyser (i.e. pas assez réactif) mais plutôt de stocker des filtres, des alertes, puis d'appliquer ces filtres sur les données surveillées en temps réel. En ce sens, *Esper* peut être défini de moteur de règles appliqué aux données. Afin de définir les traitements à appliquer aux données, *Esper* propose un langage de requêtage appelé *Event Processing Language (EPL)*.

Représentation des événements avec Esper

Nous nous appuyons sur l'implémentation *Java* de *Esper*. Ainsi un événement est représenté par un objet *Java* d'un certain type. Pour faire le parallèle avec le monde des bases de données relationnelles, un type d'événement (e.g. classe *VmHighCpu* avec ses attributs) peut s'apparenter à une table (avec ses colonnes), tandis qu'un objet (e.g. *myVmHighCpuEvent*) peut être associé à un tuple. Ainsi, un nouvel événement généré à l'exécution va correspondre à la création d'un nouvel objet pouvant être poussé dans un flux d'événement. La relation entre événement et traitement est mise en place au travers du patron de conception *Observateur* (*Design Pattern Observer* [GHJV94]). Il s'agit concrètement d'associer à un type d'événement, représenté par un *statement* (*observable*), un *écouteur* (*listener*) jouant le rôle d'observateur, qui va être notifié de la création d'un nouvel objet du type concerné pour effectuer un traitement quand celui-ci survient (e.g. *logger* une information, ajouter l'événement à la queue de *symptômes*, etc.). Nous montrerons dans la sous-section 6.1.4 comment s'articulent ces différentes notions de requêtes *EPL*, *statement* ou *listener* avec *Esper*, tout en rattachant celles-ci à notre modèle conceptuel, en déroulant un exemple complet du processus de paramétrage de la surveillance réalisé par l'administrateur Cloud.

EPL : un DSL pour CEP

Les événements, générés dynamiquement par le moteur *CEP*, peuvent faire l'objet de requêtes (i.e. *select* dynamique) définies avec le langage *EPL* et dont le résultat peut constituer un nouvel événement. Avec *Esper*, une requête *EPL* est encapsulée dans un *statement* (i.e. interface *EPStatement*) qui correspond à un *état* particulier "intéressant" du système auquel il est possible d'associer un traitement (i.e. un événement considéré comme un *symptôme*).

EPL [epl15] est un langage de requêtage, proche de *SQL* (*Structured Query Language*), qui permet de spécifier des filtres, des événements, de les agréger, de lire leur contenu, etc. Néanmoins, contrairement à *SQL* qui vise à requêter sur des données historiques contenues dans une base de données relationnelle, *EPL* va permettre de requêter sur un ensemble d'évènements se produisant sur une période de temps donnée (i.e. flux de données représentées sous forme d'événements).

Le moteur *CEP* (i.e. *Esper*) permet d'interroger dynamiquement ce flux d'événements (i.e. objets *Java*), contrairement aux bases de données relationnelles où l'on va écrire les données (i.e. persister) pour les lire ultérieurement (i.e. requêter). En ce sens, une requête *EPL* constitue un état (i.e. *statement*) que l'on souhaite repérer et qui est automatiquement identifié quand celui-ci survient. Un flux d'événements (i.e. *event stream*) correspond concrètement à une séquence linéaire d'événements ordonnée par rapport à un axe temps.

6.1.4 Prise en main du *framework perCEPtion* : marches à suivre

Dans cette section, nous allons illustrer les possibilités offertes par le *framework perCEPtion* à travers un exemple complet constituant le processus de définition et de paramétrage de la surveillance réalisé par l'administrateur Cloud.

Graphe de ressources à surveiller

Nous nous appuyons sur l'exemple de système Cloud représenté dans la Figure 6.2. Il s'agit d'un graphe de ressources, qui constitue une configuration du système que l'on souhaite surveiller et dans lequel les nœuds représentent les ressources Cloud et les arcs symbolisent les relations de composition entre celles-ci. Dans cet exemple, on considère un système avec une application hébergée sur deux PMs et respectant une architecture 2-tiers. Chaque tier est constitué de VMs (déployées sur des PMs) intégrant elles-mêmes des composants. Chaque type de ressource (i.e. Application, PM, Tier, VM, Composant) expose des métriques qui lui sont propres au travers de sondes. La représentation d'un tel système est réalisée à travers un modèle implémenté en *Java* et maintenu à jour à l'exécution (cf. sous-section 5.2.1).

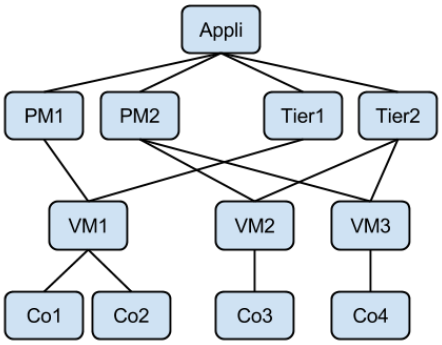


FIGURE 6.2 – Exemple d'un système Cloud : graphe de ressources.

Définition d'événements avec *Esper* : statement

Nous allons nous intéresser ici à l'utilisation d'*Esper* dans l'implémentation de notre *framework perCEPtion*. Pour ce faire, nous présentons quelques exemples de définition d'événements sous forme de requêtes *EPL*, encapsulées dans des *statements*. L'administrateur Cloud, à travers cette première étape, va spécifier les informations qu'il souhaite surveiller et qui seront remontées à l'exécution sous forme d'événements pouvant admettre différents niveaux de granularité.

Prenons un *événement simple*, nommé *VmHighCpu*, correspondant au fait qu'une VM a vu sa consommation moyenne de CPU dépasser le seuil de *70%* pendant les dernières 30sec. Cela revient à requêter notre flux d'*événements primitifs*, en s'intéressant particulièrement aux événements de type *VmCpu* (cf. Code 6.2), pour les agréger en vue d'obtenir un nouvel événement "intelligible" qui va porter l'information d'une VM surchargée en CPU.

```
// Type d'evenement primitif
public class VmCpu extends PrimitiveEvent {
  // Attributs presents dans la classe mere abstraite
  private long timestamp;
  private Resource source;
  private double value;
  // Getters and Setters
  // ...
}
```

Code 6.2 – Classe *Java* de l'*événement primitif VmCpu*.

La définition de la requête *EPL* (et le *statement Esper* associé) correspondant à l'*événement simple VmHighCpu* est donnée dans le Code 6.3. Un événement sera déclenché dans le système à chaque fois que la requête remontera des résultats (i.e. requête évaluée dynamiquement). Le langage *EPL* est à la fois expressif et complet ce qui permet d'écrire des requêtes succinctes. On remarquera la convention de nommage utilisée par nos soins pour nommer nos événements ainsi que les variables associées (i.e. requêtes *EPL* et *statements*).

```
// Definition de la requete EPL
String vmHighCpu_req =
"select source, avg(value)" +
  " from VmCpu.std:groupby(source).win:time(30 sec)" +
    " group by source having avg(value)>70.0";

// Enregistre la requete aupres de l'engine sous forme de statement
EPStatement vmHighCpu_stmt =
  myEpService.getEPAdministrator().createEPL(vmHighCpu_req);
```

Code 6.3 – Définition d'un *événement simple* : requête *EPL* et *statement* associé.

Le Code 6.4 donne un exemple de définition d'*événement complexe* qui spécifie qu'une VM est surchargée en CPU (e.g. *VmHighCpu* sur *VM2*) et que ses composants (i.e. *Co3*) admettent des temps de réponse élevés (e.g. représenté par un *événement simple* de type *CompHighRT*). Ainsi, l'événement généré va retourner la VM et le composant concernés pour un éventuel traitement (i.e. adaptation) sur ces deux ressources.

```
// Definition de la requete EPL
String vmHighCpu_compHighRT_req =
"select vmEvent.source, compEvent.source " +
  "from VmHighCpu.win:time(30 sec) as vmEvent, " +
    "CompHighRT.win:time(30 sec) as compEvent " +
      "where vmEvent.source = compEvent.parent";
// Enregistre la requete aupres de l'engine sous forme de statement
EPStatement vmHighCpu_compHighRT_stmt =
myEpService.getEPAdministrator().createEPL(vmHighCpu_compHighRT_req);
```

Code 6.4 – Définition d'un *événement complexe* : requête *EPL* et *statement* associé.

Traitement des événements générés : listener

Nous avons donné quelques exemples de définition d'événements (simples et complexes) avec notre *framework* sans présenter comment associer un traitement aux événements survenant à l'exécution avec *Esper*. Nous avons évoqué que la relation entre événement et traitement était mise en place au travers du patron de conception *Observateur*. Le principe revient à associer un objet *Observateur* (i.e. *Listener*) à un *statement*. Le Code 6.5 illustre cette association entre *statement* et *listener* en attachant deux *observateurs* à nos deux événements décrits précédemment (cf. Codes 6.3 et 6.4).

```
// Attache un listener a chaque statement
vmHighCpu_stmt.addListener(new VmHighCpuListener());
vmHighCpu_compHighRT_stmt.addListener(new VmHighCpuCompHighRTListener());
```

Code 6.5 – Association entre *statements* et *listeners*.

Un *Observateur*, dans notre *framework*, correspond à une classe *Java* implémentant l'interface *UpdateListener* ou *StatementAwareUpdateListener*. Ces interfaces contiennent toutes les deux une méthode *update()*, dont la signature diffère (cf. documentation *Esper* pour plus de détail [esp15a]), qui sera appelée lorsqu'un événement observé associé survient. Il s'agit alors d'implémenter la méthode *update()* dans notre classe *Observateur* pour définir le traitement attendu lors de la génération d'un événement. Les Codes 6.6 et 6.7 donnent respectivement le code des *listeners* associés aux *statements vmHighCpu_stmt* et *vmHighCpu_compHighRT_stmt*. Le premier *listener* est simplement chargé d'afficher sur la sortie standard (i.e. console *Java*) les VMs (i.e. méthode toString()) admettant un problème de surcharge CPU (i.e. *VmHighCpu*). Dans cet exemple, les événements de type *VmHighCpu* ne nécessitent pas d'adaptation et ne sont donc pas "convertis" en *symptômes*.

```
// Classe Observateur associee au statement vmHighCpu_stmt
public class VmHighCpuListener implements UpdateListener {
  // Traitement declenche en cas d'evenement
  public void update(EventBean[] newEvents, EventBean[] oldEvents) {
    System.out.println("VM " + newEvents[0].get("source") + " overloaded !");
  }
}
```

Code 6.6 – Implémentation d'un *listener* : logging.

Des événements aux *symptômes*

Le second *listener* (cf. Code 6.7) admet quant à lui un traitement plus avancé. Il s'agit ici de "convertir" les événements reçus (i.e. de type *VmHighCpuCompHighRT*) en *symptômes* et de les pousser dans la queue de *symptômes* en attente de traitement. Bien que ce *listener* soit spécifique aux événements de type *VmHighCpuCompHighRT*, il est possible d'envisager la création d'un *listener générique* chargé de créer un *symptôme* (à partir de n'importe quel type d'événement) puis de l'ajouter à la queue. Cela est rendu possible avec les mécanismes de réflexion (i.e. *reflection*) offerts par le langage *Java* ainsi qu'en exploitant nos conventions de nommage (i.e. un *symptôme* a le même nom que l'événement associé avec le suffixe *Symptom*).

```
// Classe Observateur associee au statement vmHighCpu_compHighRT_stmt
public class VmHighCpuCompHighRTListener implements
    StatementAwareUpdateListener {
  // Traitement declenche en cas d'evenement
  public void update(EventBean[] newEvents, EventBean[] oldEvents,
      EPStatement statement, EPServiceProvider epServiceProvider) {
    System.out.println("VmHighCpuCompHighRT event !");
    System.out.println("--> Involved resources: " +
      newEvents[0].get("source"));
    // Creation du symptome correspondant a l'evenement
    Symptom mySymptom = new VmHighCpuCompHighRTSymptom(newEvents[0]);
    // Ajout du symptome a la queue (triee) de symptomes
    // en attente de traitement (declenche la purge)
    SymptomsQueue.getInstance().add(mySymptom);
    System.out.println("--> Symptom VmHighCpuCompHighRTSymptom " +
      "created and added to the symptoms queue.");
  }
}
```

Code 6.7 – Implémentation d'un *listener* : création d'un *symptôme* pour analyse.

De la surveillance à l'analyse : queue de *symptômes*

Le Code 6.8 donne l'implémentation de la queue de *symptômes*. Il s'agit d'une classe *Java*, basée sur le patron de conception *Singleton* [GHJV94], dont le but est de restreindre l'instanciation d'une classe à un seul objet. Cette unique instance est alors accessible depuis tout le système (cf. *SymptomsQueue.getInstance()*).

Pour rappel, notre *SymptomsQueue* est implémentée sous forme de *queue à priorité* et étend donc la classe *java.util.PriorityQueue<Symptom>*. Cette queue de *symptômes* est triée (à chaque ajout d'élément) en fonction du score des *symptômes* (calculé à la création) et de leur date d'apparition (cf. méthode *compareTo(Symptom s)* Code 6.1). De plus, chaque ajout dans la queue est précédé d'une *purge*. Pour ce faire, nous avons surchargé la méthode *add()* afin d'effectuer une *purge* des *symptômes* obsolètes avant l'ajout d'un nouvel élément dans la queue.

La queue de *symptômes* joue le rôle d'interface entre les phases de *Monitoring* et d'*Analyze* du gestionnaire autonomique *MAPE-K*. La phase de *Monitoring*, au travers du *framework perCEPtion*, va pousser les *symptômes* nécessitants un traitement dans la queue de *symptômes* tandis que la phase d'*Analyze* va récupérer de manière itérative les éléments en tête de la queue pour les traiter. Afin de garantir que l'*Analyze* ne traite pas de *symptômes* obsolètes (i.e. ayant atteint leur date de *péremption*), le retrait d'un élément de la queue est précédé par une *purge*. Ainsi, comme pour la méthode *add()*, nous avons surchargé la méthode *poll* du type parent afin d'ajouter une *purge* avant de procéder à la récupération de l'élément en tête de la queue.

```
// Classe qui herite de java.util.PriorityQueue<E>
// Implementee a l'aide d'un singleton
// Instance creee a l'initialisation
public class SymptomsQueue extends PriorityQueue<Symptom> {
  // Constructeur prive
  private SymptomsQueue() {
  }
  // Instance unique pre-initialisee
  private static SymptomsQueue SYMPTOMSQUEUE = new SymptomsQueue();
  // Acces a l'instance du singleton
  public static SymptomsQueue getInstance() {
    return SYMPTOMSQUEUE;
  }
  // Surcharge de la methode add du type parent PriorityQueue
  // Preceder l'ajout d'un symptome d'une purge de la queue
  public boolean add(Symptom s) {
    this.purge();
    return super.add(s);
  }
  // Surcharge de la methode poll du type parent PriorityQueue
  // Preceder le retrait du symptome en tete de la queue d'une purge
  public Symptom poll() {
    this.purge();
    return super.poll();
  }
  // Methode chargee de purger les symptomes obsoletes
  public void purge() {
    Long currentTime = System.currentTimeMillis();
    for(Symptom s : this) {
      if(s.getPeremption() > currentTime) {
        this.remove(s);
      }
    }
  }
}
```

Code 6.8 – Implémentation *Java* de la *queue de symptômes*.

6.2 *ElaScript* : un langage dédié à l'élasticité multi-couche

Nous allons présenter dans cette section notre DSL *ElaScript*. L'objectif de ce langage dédié à l'élasticité est de permettre d'exprimer des plans de reconfiguration à mettre en œuvre sur un système Cloud de façon naturelle et concise.

6.2.1 Motivations

Un *DSL* (*Domain Specific Language*) est défini par *Martin Fowler* comme "un langage de programmation informatique dédié à un domaine particulier, ayant un champ d'application délimité, et une syntaxe naturelle" [Fow10]. Cette définition soulève trois caractéristiques essentielles :

— Un DSL est dédié à un seul domaine, ce qui (à l'extrême) le rend "inutilisable" pour un autre domaine. Le domaine peut être métier ou technique, et être lié à un système, ou un aspect d'un système. Dans notre cas, le langage proposé s'intéresse au domaine technique qu'est la reconfiguration d'architectures Cloud.

— Un DSL, contrairement aux langages de programmation généralistes (e.g. *Java*, *C*, etc.), est un langage ayant un but spécifique et un champ d'application délimité (ce qui limite sa réutilisabilité dans un autre contexte). On parle parfois d'*expressivité limitée* pour définir le fait qu'un DSL dispose uniquement du vocabulaire nécessaire pour le rendre approprié à son domaine.

— Enfin, un DSL est un langage de programmation, ce qui implique qu'un être humain doit être mesure de le lire et de l'écrire. En ce sens, un DSL doit être intelligible en fournissant une syntaxe naturelle et intuitive qui permette au développeur (i.e. expert du domaine) de spécifier ses intentions de manière simple et concise.

Le fait de proposer un langage dédié pour l'élaboration de plan de reconfiguration prend tout son sens dans le contexte industriel de cette thèse. Comme nous l'avons évoqué ci-dessus, l'une des caractéristique d'un DSL est qu'il doit se limiter à un domaine précis et ainsi être utilisé par des développeurs ou experts de ce domaine. En ce sens, les langages *EPL* (inclus dans le *framework perCEPtion*) et *ElaScript* ne vont pas être destinés aux mêmes personnes. En effet, une personne faisant partie d'une équipe de test va être en mesure d'identifier puis de définir les *symptômes* avec le langage *EPL* sans nécessairement être en mesure de définir un plan de reconfiguration (*ElaScript*) pertinent et efficace face à ce *symptôme*, qui relève davantage des compétences d'un architecte logiciel (i.e. pour l'adaptation PaaS/SaaS) ou d'un architecte système (i.e. pour l'adaptation IaaS). On peut néanmoins imaginer qu'une personne avec un profil *DevOps* [dev16] [Htt12], ayant des compétences de développement et d'exploitation, est en mesure d'assumer ces deux rôles.

6.2.2 Intégration au gestionnaire autonomique

ElaScript vient outiller les phases de *Plan* et d'*Execute* de notre gestionnaire autonomique (i.e. boucle *MAPE-K*) dans le but d'industrialiser le processus de gestion de l'élasticité.

Le langage doit permettre d'exprimer des plans de reconfiguration génériques (*P* de *MAPE-K*), sous forme de *scripts*, dans lesquels certaines ressources seront découvertes dynamiquement. Un *script* décrit une adaptation possible qui pourra être appliquée au système (*E* de *MAPE-K*). Les actions du plan de reconfiguration sont alors traduites en appels aux différents *actuators* mis à disposition par le système (e.g. *deployVirtualMachine* sur *Apache CloudStack* qui va créer et démarrer une nouvelle VM).

L'ensemble des scripts écrits avec le langage *ElaScript* forme le catalogue des plans de reconfiguration qu'il est possible d'appliquer au système dont on attend un comportement autonomique. Ce catalogue, tout comme le catalogue de *symptômes*, est stocké dans la base de connaissance du gestionnaire autonomique (i.e. *Knowledge*). La Figure 6.3 reprend le gestionnaire autonomique en y positionnant les contributions évoquées dans ce chapitre à savoir notre *framework perCEPtion* (cf. zone verte) et notre DSL *ElaScript* (cf. zone violette).

6.2.3 Spécifications du langage

Nous allons dresser ici les spécifications de notre langage dédié *ElaScript*.

Ressources et actions associées

ElaScript permet de définir des plans de reconfiguration portant sur une instance du modèle de ressources. Il s'agit concrètement de définir un ensemble d'actions d'adaptation (ordonnées) à entreprendre sur les différents nœuds du graphe de ressources face un *symptôme*. Les ressources accessibles dans le script sont contenues et remontées par le *symptôme*. Pour rappel, notre modèle de ressources considère des ressources de différents types (cf. sous-section 5.2.1) sur lesquelles il est possible d'effectuer des actions d'adaptation diverses (cf. sous-section 5.1.2).

Le langage offre la possibilité d'effectuer des actions sur les nœuds remontés par le *symptôme* et éventuellement sur les nœuds fils au travers de l'*API* d'adaptation. Néanmoins, un *script* de reconfiguration ne peut pas contenir d'action portant sur les nœuds parents des ressources remontées par le *symptôme*. Cette limitation se justifie du fait qu'un *symptôme* traduit une incohérence d'une partie du système (i.e. sous-graphe de nœuds). Si des ressources parentes peuvent nécessiter un traitement, celles-ci doivent être remontées par le *symptôme* ou faire l'objet d'un autre *symptôme*.

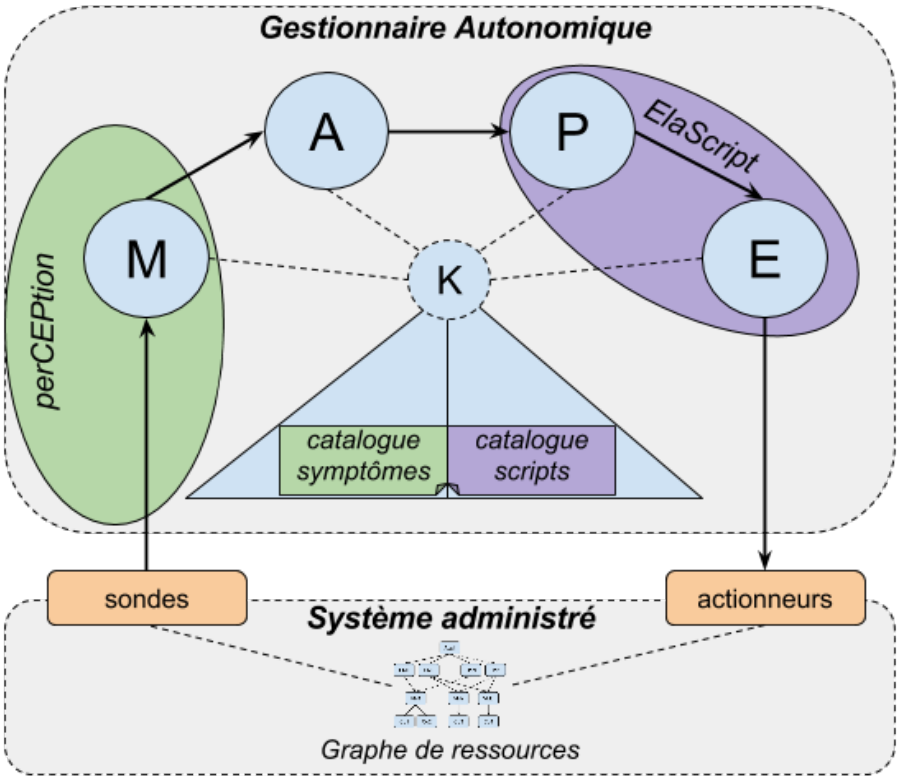


FIGURE 6.3 – *perCEPtion* et *ElaScript* dans le gestionnaire autonomique.

Les actions d'adaptation sont représentées sous forme de *fonctions*. Ainsi, une *invocation de fonction* va se résumer à une *instruction* classique de la forme suivante : $variable.fonction$ pour laquelle $variable$ et $fonction$ désignent respectivement une ressource et une action d'adaptation. Pour rappel, nous considérons dans ce travail 8 actions d'adaptation distinctes (i.e. *fonctions*) pouvant admettre différentes signatures. L'ensemble des signatures de *fonctions* possibles forme l'*API* de reconfiguration du système (i.e. ensemble des *actuators*).

ElaScript va permettre, au vu du nombre limité de ressources et d'actions, d'exprimer de façon naturelle des transformations (génériques) sur la structure du graphe de ressources (i.e. actions de type SO_{infra}, SI_{infra}, SO_{soft}et SI_{soft}) ainsi que sur les nœuds eux-mêmes (i.e. actions du type SU_{infra}, SD_{infra}, SU_{soft}et SD_{soft}), tout en offrant des garanties quant à la consistance de ces manipulations (i.e. au travers d'une analyse sémantique). Il s'agit par exemple de limiter certaines *fonctions* (i.e. actions d'adaptation) à certains types de ressources.

Passage du contexte et parcours des ressources

Le concept de *symptôme* a été présenté dans la sous-section 6.1.2 et défini comme un événement "auto-contenu" (*self contained*) donnant toutes les informations nécessaires pour mettre en œuvre un plan de reconfiguration.

Un *symptôme* contient une collection de ressources malades (i.e. *variables*) qui désigne les ressources sur lesquelles agir. Cette collection peut contenir un unique élément si le *symptôme* est issu d'un *événement simple* ou plusieurs s'il est issu d'un *événement complexe*. Dans le second cas, les ressources sont regroupées en sous-collections contenant les ressources retournées par les différents événements composés (i.e. regroupées par maladie).

Il est alors nécessaire de parcourir les différents éléments (i.e. ressources) d'une collection. C'est pour cette raison que nous munissons notre langage de la structure de boucle *foreach* qui offre une façon simple de parcourir des collections. Ce type de boucle va passer en revue chaque élément d'une collection de manière itérative et les stocker dans une *variable temporaire* en vue d'effectuer des traitements.

Portée des variables

Le langage intègre de manière implicite la notion de *variable* au travers des arguments du *symptôme* permettant de manipuler des ressources sous forme de *variables*. En ce sens, *ElaScript* ne permet pas de définir de nouvelles *variables* (i.e. déclarations) du fait que toutes les informations nécessaires à la définition d'un plan de reconfiguration sont contenues dans les arguments du *symptôme* (i.e. *auto-contenu*). Pour rappel, les *variables* du *symptôme* peuvent désigner une ressource ou une collection de ressources.

La portée des *variables* d'un *symptôme* encapsulant les ressources malades n'est pas limitée, c'est-à-dire que celles-ci sont accessibles dans toutes les parties du *script* (i.e. du point de vue de l'analyse syntaxique). Néanmoins, nous verrons que l'analyse sémantique peut limiter l'accès aux variables en fonction de certaines *invocations de fonctions* (i.e. actions appliquées aux ressources).

On distingue cependant un autre type de *variable* lié à la structure *foreach* définie précédemment. Il s'agit d'une *variable temporaire* qui joue le rôle d'objet *itérateur* désignant l'élément courant de la collection parcourue et dont la portée est limitée au bloc du *foreach* (i.e. indiqué par des accolades).

Ordonnancement des instructions

Nous avons évoqué qu'un *script* était constitué d'un ensemble d'*instructions* (i.e. *invocations de fonctions*). Néanmoins, nous n'avons pas abordé comment celles-ci pouvaient être ordonnancées.

Le langage *ElaScript* permet d'orchestrer les multiples actions d'élasticité via des opérateurs de *séquentialité*, de *parallélisme* et de *synchronisation*. En effet, l'adaptation dynamique d'architectures logicielles (i.e. graphe de ressources), en fonction des ressources et actions concernées, peut permettre, nécessiter ou encore bénéficier d'une exécution parallèle des tâches de reconfiguration.

Le langage *ElaScript*, en plus de la séquentialité des opérations, offre la possibilité de définir des blocs d'instructions qui vont s'exécuter en parallèle (i.e. *threads* différents). L'objectif est d'optimiser les *scripts* en jouant sur l'orchestration des actions d'élasticité en vue de réduire les *temps de reconfiguration* tout en améliorant la *réactivité* de l'adaptation (cf. critères, sous-section 5.1.3).

La *parallélisation* des instructions offerte par le langage *ElaScript* est corrélée à la notion de *synchronisation*. Il s'agit par exemple d'attendre l'exécution d'un bloc d'instruction (parallèle) avant de poursuivre l'exécution du script (i.e. point de *synchronisation* dans l'exécution du programme).

Les notions de *parallélisation* et de *synchronisation* sont illustrées dans la Figure 6.9 au travers d'un exemple commenté faisant intervenir 2 blocs d'instructions parallèles. Comme nous le verrons par la suite, ce type d'exécution parallèle nécessite de définir certaines propriétés sémantiques pour offrir des garanties quant à son utilisation (e.g. empêcher des *invocations de fonctions "concurrentes"* dans 2 blocs parallèles).

```
var1.foo();
var2.bar();
[ // Split operator
  // Bloc1
  var1.baz();
|| // Parallel operator
  // Bloc2
  var2.baz();
] // Join operator: synchronization point
// The executions of Bloc1 AND Bloc2 must be done
// Before the execution of the next instruction
var1.qux();
```

Code 6.9 – Exemple de blocs parallèles.

Le langage procure aussi une instruction *wait* permettant de définir une durée entre deux *instructions séquentielles*. Cela permet par exemple d'indiquer que la reconfiguration admet une attente d'une certaine durée entre 2 *invocations de fonctions*. Le Code 6.10 illustre cette notion de *wait* au travers d'un exemple visant à réduire l'Off_{soft} d'un composant pendant 2min (cf. action SD_{soft}) puis de re-basculer celle-ci à son mode initial (cf. action SU_{soft}).

```
// decrease comp's software offering
comp.sds();
// wait for 2 minutes
wait(120);
// increase comp's software offering
comp.sus();
```

Code 6.10 – Exemple d'instructions séquentielles avec l'instruction *wait*.

Grammaire du langage : analyse syntaxique

La grammaire formelle du langage *ElaScript* est donnée dans la Figure 6.4 en suivant le formalisme *EBNF* (i.e. *Extended Backus-Naur Form* [Gar03]).

Le respect de la grammaire du langage est assuré par un programme informatique que l'on appel analyseur syntaxique (i.e. *parser*) qui procède à la phase d'analyse syntaxique. Dans notre cas, l'analyseur syntaxique est chargé de s'assurer que les *scripts* écrits par le développeur *ElaScript* sont syntaxiquement corrects. Nous reviendrons sur l'analyse syntaxique ainsi que notre grammaire dans la sous-section 6.2.4 dans laquelle nous détaillerons l'implémentation de notre langage à travers l'utilisation de l'outil *Xtext* [xte15b].

On remarquera dans la grammaire de la Figure 6.4 que le langage offre la possibilité de définir des commentaires afin de documenter les plans de reconfiguration développés. Les commentaires, à l'instar de langage *Java*, peuvent être présentés sur une ligne unique (cf. *symbole* "//") ou sur plusieurs lignes (cf. entre les *symboles* "/*" et "*/" avec une "*" par ligne). Enfin, comme nous le verrons par la suite, les *scripts* de reconfiguration implémentés avec le langage *ElaScript* sont décrits dans des fichiers ayant l'extension *.elas*.

Propriétés du langage : analyse sémantique

La grammaire donnée dans la Figure 6.4 offre la spécification de la syntaxe du langage *ElaScript*, dont le respect est assuré par un analyseur syntaxique (i.e. *parser*). Néanmoins, bien que syntaxiquement corrects, les *scripts* définis peuvent ne pas avoir de sens par rapport à la sémantique du langage (ou du domaine). Ainsi, afin de rendre le langage opérationnel pour écrire des plans de reconfiguration "sûrs" (i.e. *safe*), il est nécessaire de doter celui-ci de certaines propriétés qui vont concerner la sémantique des programmes produits. Le respect des règles sémantiques est assuré par la phase d'analyse sémantique.

L'analyse sémantique est statique, c'est-à-dire qu'elle se fait au moment de la compilation. Ainsi, celle-ci ne peut pas exclure les programmes dont on ne peut savoir s'ils sont sémantiquement corrects qu'à l'exécution (i.e. *at runtime*). Dans notre cas, les ressources sont découvertes à l'exécution du système, nécessitant de contrôler le contexte de manière dynamique. Nous verrons plus tard dans ce manuscrit (cf. section 7.3) que le contrôle sémantique à l'exécution incombe à la partie *décision* de notre *framework* autonomique (i.e. *A* de *MAPE-K*).

```
script = BEGIN WHEN symptom DO {action} END ;
symptom = identifier param_symptom_list ;
param_symptom_list = LP identifier {COMMA identifier} RP ;
identifier = ( LETTER ) { LETTER | DIGIT | "_" } ;
body = body_part {PARALLELSEPARATOR body_part} ;
body_part = {comment | parallelized | for_each | action} ;
comment = COMMENT {LETTER | DIGIT} ;
parallelized = SPLIT body JOIN ;
for_each = FOREACH identifier IN identifier LB {action} RB ;
action = identifier CALL command ;
command = (function | function param_action_list) SEQUENCIALSEPARATOR ;
function = (scale_function | wait_function) ;
param_action_list = LP param_action {COMMA param_action} RP ;
param_action = {DIGIT} | {LETTER} ;
scale_function =
SCALE_IN_INFRA
| SCALE_OUT_INFRA
| SCALE_UP_INFRA
| SCALE_DOWN_INFRA
| SCALE_IN_SOFT
| SCALE_OUT_SOFT
| SCALE_UP_SOFT
| SCALE_DOWN_SOFT ;
wait_function = WAIT LP {DIGIT} RP ;
```

(a) Symboles non-terminaux et règles de production.

```
SCALE_IN_INFRA = "sii" ;
SCALE_OUT_INFRA = "soi" ;
SCALE_UP_INFRA = "sui" ;
SCALE_DOWN_INFRA = "sdi" ;
SCALE_IN_SOFT = "sis" ;
SCALE_OUT_SOFT = "sos" ;
SCALE_UP_SOFT = "sus" ;
SCALE_DOWN_SOFT = "sds" ;
BEGIN = "begin" ;
END = "end" ;
WHEN = "when" ;
DO = "do" ;
LETTER = "a".."z" | "A".."Z" ;
DIGIT = "0".."9" ;
LP = "(" ;
RP = ")" ;
LB = "{" ;
RB = "}" ;
SEQUENCIALSEPARATOR = ";" ;
PARALLELSEPARATOR = "||" ;
SPLIT = "[" ;
JOIN = "]" ;
COMMA = "," ;
CALL = "." ;
COMMENT = "#" ;
FOREACH = "foreach" ;
IN = "in" ;
WAIT = "wait" ;
```

(b) Symboles terminaux et règles de production.

FIGURE 6.4 – Grammaire formelle *EBNF* du langage *ElaScript*.

Les règles sémantiques définies au préalable et évaluées au moment de la compilation, peuvent être diverses et variées. Nous allons donner ci-dessous quelques exemples de règles sémantiques que nous souhaitons assurer lors de l'écriture de *scripts* avec notre langage.

— *Règle de nommage :* un premier type de propriété peut être l'impossibilité d'avoir 2 ressources avec le même nom dans un même *script*. Il s'agit ici d'empêcher l'écriture de plans de reconfiguration faisant intervenir 2 ressources distinctes (i.e. *variables*) ayant le même identifiant (i.e. nom) ;

— *Règle de portée :* ce type de règle, comme son nom l'indique, correspond à la portée (i.e. *scoping*) des éléments manipulés dans le langage. Un exemple peut être de limiter la visibilité des *variables* à certaines portions du *script* (e.g. l'accès d'une variable à un bloc d'instructions dans le cas d'une boucle) ;

— *Règle de typage :* une action du type SI_{infra} ne peut pas être appelée sur une ressource de type *Component* (i.e. pas de sens). On peut aussi étendre cette propriété à plusieurs actions ;

— *Règle métier :* une action d'adaptation du type SI_{infra}, visant à éteindre une VM, ne peut être suivie d'une autre action sur cette VM (du fait qu'elle va être éteinte). Cette propriété porte véritablement sur le sens que l'on souhaite apporter au langage (i.e. sémantique de l'action SI_{infra}). Il est possible d'étendre cette règle métier aux ressources filles de la VM concernée (i.e. en bénéficiant de la structure de graphe), en bloquant les actions d'adaptation sur les nœuds "fils" de celle-ci (i.e. composants de la VM).

Le fait de fournir des propriétés sémantiques au langage est une phase essentielle dans la définition d'un *DSL*. Dans le cas du langage *ElaScript*, cela nous permet non seulement d'enrichir celui-ci en apportant du sens et de la pertinence aux *scripts* produits mais aussi de contrôler et de garantir à l'utilisateur du langage (i.e. développeur *ElaScript*) l'écriture de programmes "sûrs". De plus, comme nous allons le voir par la suite, différents niveaux de règles sémantiques peuvent être définis (i.e. implémentés) permettant, en cas de règle violée, de lever au choix une erreur, une alerte (i.e. *warning*) ou encore une information. Tout cela, intégré dans un environnement de développement (e.g. éditeur adapté), offre l'outillage nécessaire pour utiliser efficacement le langage et ainsi accroître la productivité des administrateurs.

6.2.4 Implémentation et illustration

Nous allons présenter ici certaines briques logicielles que nous avons utilisées pour l'implémentation du langage *ElaScript*. Puis, nous verrons l'outillage mis en place autour de celui-ci afin de simplifier son utilisation. Enfin, nous donnerons quelques exemples concrets de définition de plans de reconfiguration avec *ElaScript*.

Briques logicielles

L'implémentation de notre langage s'appuie sur deux outils : *Xtext* [xte15b] et *Xtend* [xte15a]. Le premier outil est un *framework Java* pour le développement de langages de programmation et de *DSL*. Il repose sur une grammaire générée *ANTLR* (i.e. *ANother Tool for Language Recognition* [ant15]) ainsi que sur le *framework* de modélisation *EMF* (i.e. *Eclipse Modeling Framework* [emf15]). *Xtext* s'intègre parfaitement à l'environnement de développement (i.e. *IDE*) *Eclipse* [ecl15a] et offre donc les fonctionnalités suivantes : coloration syntaxique, auto-complétion, proposition de correction (i.e. *quick fix*), etc. En ce sens, *Xtext* permet de développer (en partie) un *DSL*, à partir d'une grammaire, et de le munir d'un environnement convivial et adapté à destination des utilisateurs de celui-ci.

Le second outil, *Xtend*, est quant à lui un langage compilé en *Java* (et développé à partir d'*XText*) s'exécutant sur la *JVM* (i.e. *Java Virtual Machine*) et s'intégrant ainsi avec toutes les bibliothèques *Java*. En ce sens, sa syntaxe et sa sémantique sont proches du langage *Java*. Concernant le développement de notre *DSL*, *Xtend* est utilisé pour éditer certaines classes générées par *Xtext* permettant ainsi d'implémenter un ensemble de fonctionnalités propres au fonctionnement et à l'outillage de *ElaScript* (i.e. règles de validation, règles pour limiter la portée des éléments du langage, messages produits par l'éditeur, etc.). Il s'agit concrètement d'implémenter nos règles sémantiques à l'aide de ce langage. Nous donnerons par la suite quelques exemples de définition de règles sémantiques avec *Xtend*.

Le *framework Xtext* et le langage *Xtend*, utilisés conjointement [Bet13] [EB10], offrent une solution complète pour l'implémentation d'un *DSL*. Pour rappel, ces deux outils s'intègrent parfaitement à l'environnement de développement *Eclipse* [ecl15a]. En ce sens, l'éditeur *Eclipse* peut être utilisé pour développer avec le langage *ElaScript* ce qui permet notamment de bénéficier de toutes les fonctionnalités de celui-ci. Néanmoins, il est aussi possible d'utiliser l'éditeur d'un autre *IDE* (e.g. *IntelliJ IDEA* [int16]) ou encore un éditeur contenu dans un navigateur web (i.e. accessible au travers d'un serveur web local lancé via *Gradle* [gra16]) comme présenté dans la Figure 6.5.

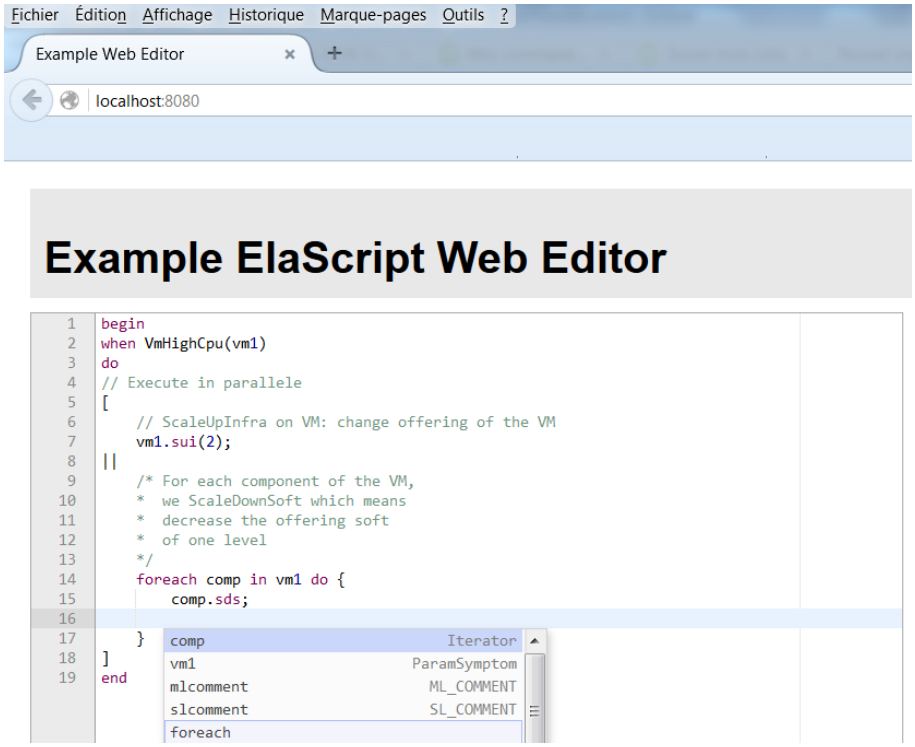


FIGURE 6.5 – Exemple d'éditeur *ElaScript* intégré au navigateur web.

Description de l'implémentation

La première étape fut d'écrire la grammaire *Xtext* de notre langage (i.e. correspondant à la grammaire *EBNF* donnée dans la Figure 6.4). La grammaire *Xtext* d'*ElaScript* est donnée dans le Code 6.11. Il est possible de constater qu'*Xtext* permet d'écrire une grammaire de manière concise.

À partir de la description de notre grammaire, *Xtext* va pouvoir générer différents objets (*lexer*, *parser*, etc.), des classes Java (i.e. interfaces et implémentations) correspondant à chaque symbole non terminal ayant fait intervenir une variable (cf. *script*, *symptom*, *param*, etc.). Ces classes, encapsulées comme objets dans l'arbre syntaxique généré par *Xtext*, vont être utilisées pour écrire le compilateur ou l'interpréteur du langage.

```
ElaScript:
  'begin'
  script=Script
  'end';
Script:
  'when' symptom=Symptom '(' param+=ParamSymptom (',' param+=ParamSymptom)*
     ')' 'do' body=Body;
Symptom:
  name=ID;
ParamSymptom:
  name=ID;
Body:
  body1+=BodyPart+ ('||' body2+=BodyPart+)?;
BodyPart:
  (Comment | Action | Foreach | Parallelized);
Comment:
  slcomment=SL_COMMENT | mlcomment=ML_COMMENT;
Action:
  ((res=([Variable]) '.' scalefunction=ScaleFunction) |
     waitfunction=WaitFunction) ';';
Variable:
  ParamSymptom | Iterator;
ParamScaleFunctionList:
  '(' param+=ParamScaleFunction (',' param+=ParamScaleFunction)* ')';
ParamScaleFunction:
  (stringvalue=STRING | intvalue=INT);
ScaleFunction:
  name=('sii' | 'soi' | 'sui' | 'sdi' | 'sis' | 'sos' | 'sus' | 'sds')
     params=(ParamScaleFunctionList)?;
WaitFunction:
  {WaitFunction} 'wait(' (time=INT)? ')';
Foreach:
  'foreach' iterator=Iterator 'in' collection=[ParamSymptom] 'do' '{'
  (body=Body)
  '}';
Iterator:
  name=ID;
Parallelized:
  '[' Body ']';
```

Code 6.11 – Grammaire *ElaScript* décrite avec *Xtext*.

Chaque classe encapsulante met à disposition les *getters* et *setters* correspondant à la structure des règles de production de la grammaire. Cela va par exemple permettre depuis la classe *Script*, d'accéder au contenu de la variable *symptom* via le getter *getSymptom()* ou encore aux ressources contenues dans le *symptom* via le getter *getParam()* qui va retourner la liste d'objets *ParamSymptom* correspondants (i.e. ressources "malades" remontées par le *symptôme*), etc. Nous avons présenté comment générer les classes *Java* liées aux symboles non terminaux de notre grammaire ainsi que le parseur (i.e. analyseur syntaxique) et autres objets *Java* de notre *DSL*. Il s'agit désormais de définir les règles sémantiques de *ElaScript* via le langage *Xtend* en éditant certaines classes générées par *Xtext*. Nous allons donner dans la suite quelques exemples de règles sémantiques de différents types.

Règle de portée : Le Code 6.12 donne un premier exemple de règle sémantique particulière visant à spécifier la portée (i.e. *scoping*) des *variables* dans le langage *ElaScript* (i.e. *ParamSymptom* et *Iterator*). Il s'agit notamment de limiter la visibilité de l'objet *itérateur* (i.e. variable *iterator*) au bloc d'instructions du *foreach*. On remarquera la syntaxe du langage *Xtend* qui diffère quelque peu de celle de *Java*. La Figure 6.6 illustre le résultat de la *Règle 1* sur l'éditeur *Eclipse* qui génère une erreur lors de l'accès à la variable *vm* en dehors du bloc *foreach*. On remarquera au passage la fonctionnalité *quick fix* (i.e. proposition de correction en un *clic*) suggérant au développeur de remplacer la *variable* inaccessible *vm* par la *variable tier*.

```
class ElaScriptScopeProvider extends AbstractElaScriptScopeProvider {
override getScope(EObject context, EReference reference) {
    if(context instanceof Action && reference ==
        ElaScriptPackage.Literals.ACTION__RES) {
      var EObject e = context;
      var EList<Variable> candidates = new BasicEList();
      while(e != null ) {
        if(e instanceof Script) {
          candidates.addAll((e as Script).param)
        }
        if(e instanceof Foreach) {
          candidates.add((e as Foreach).iterator)
        }
        e = e.eContainer
      }
      var IScope scope = Scopes.scopeFor(candidates)
      return scope
    }
    super.getScope(context, reference)
  }
}
```

Code 6.12 – Règle 1 - limiter la portée des variables.

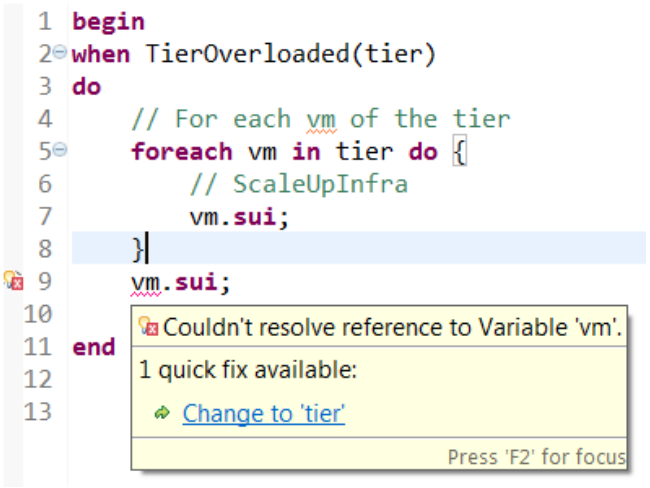


FIGURE 6.6 – Résultat de la Règle 1 sur l'éditeur *Eclipse*.

Règles de nommage : Le Code 6.13 contient 2 règles sémantiques portant sur les conventions de nommage de *ElaScript*. On remarquera l'annotation *@Check* indiquant au *framework Xtext* que la méthode qui suit est une règle de validation à vérifier.

La première règle indique que le nom d'un *symptôme* (i.e. son identifiant *name*) doit commencer par une lettre majuscule tandis que la seconde règle suggère le besoin de déclarer des noms de variable (i.e. pour les identifiants des *ParamSymptom* et *Iterator*) commençant par une lettre minuscule.

La première règle va déclencher une erreur (cf. *error(...)*) du fait que le nom d'un *symptôme* correspond à un type de *Symptom* (i.e. classe *Java*), tandis que la seconde règle va lever une alerte (cf. *warning(...)*). La Figure 6.7 donne un aperçu du résultat de ces deux règles obtenu sur l'éditeur.

```
class ElaScriptValidator extends AbstractElaScriptValidator {
  public static val INVALID_NAME_SYMPTOM = 'invalidSymptomName'
  public static val INVALID_NAME_RESOURCE = 'invalidVariableName'
  @Check
  def checkSymptomStartsWithCapital(Symptom symptom) {
    if (!Character.isUpperCase(symptom.name.charAt(0))) {
      error('Symptom Name must start with a capital',
          ElaScriptPackage.Literals.SYMPTOM__NAME,
        INVALID_NAME_SYMPTOM)}}
  @Check
  def checkResourceStartsWithLowerCase(Variable resource) {
    if (!Character.isLowerCase(resource.name.charAt(0))) {
      warning('Resource name should start with a lowercase letter',
          ElaScriptPackage.Literals.VARIABLE__NAME,
        INVALID_NAME_RESOURCE)}}
```

Code 6.13 – Règles 2 et 3 - respecter les conventions de nommage.

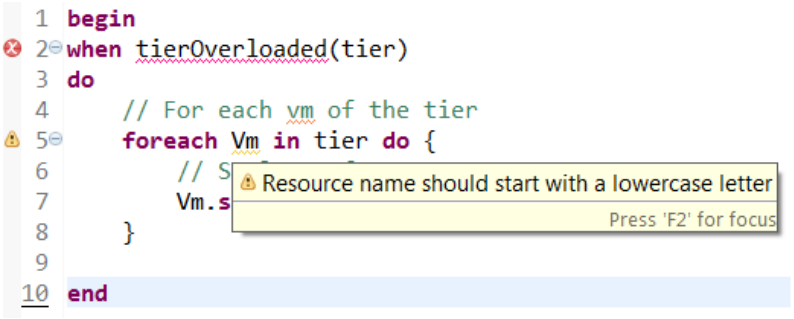


FIGURE 6.7 – Résultat des Règle 2 et 3 sur l'éditeur *Eclipse*.

Le Code 6.14 et la Figure 6.8 présentent une règle de nommage avancée générant une erreur lorsque deux paramètres d'un *symptôme* (i.e. ressources représentées sous forme de *ParamSymptom*) portent le même nom (i.e. empêchant de les distinguer dans la suite du *script*).

```
public static val DUPLICATE_NAME_RESOURCE = 'duplicateVariableName'
@Check
def checkDuplicateNameInScript(Script script) {
  for (ParamSymptom par1 : script.param) {
    var int cpt = 0;
    for (ParamSymptom par2 : script.param) {
      if (par1.name.equals(par2.name))
        cpt++;
    }
    if (cpt > 1)
      error('Duplicate name for ParamSymptom: ' + par1.name,
          ElaScriptPackage.Literals.SCRIPT__SYMPTOM,
        DUPLICATE_NAME_RESOURCE)}}
```

Code 6.14 – Règle 4 - assurer l'unicité des noms de variables.

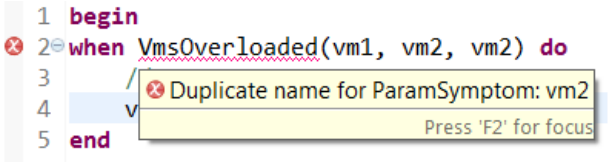


FIGURE 6.8 – Résultat de la Règle 4 sur l'éditeur *Eclipse*.

Suggestion métier : La validation avec *Xtend* offre aussi la possibilité de suggérer certaines informations au développeur via l'éditeur (i.e. sous forme de conseil) permettant une meilleure prise en main du langage et ainsi appréhender les spécificités de celui-ci. Le Code 6.15 donne 2 exemples de suggestion à propos de l'utilisation de certaines actions d'adaptation *ElaScript* (i.e. *ScaleFunction*).
La Figure 6.9 illustre le résultat de la seconde règle sur l'éditeur *Eclipse*. Il s'agit ici d'avertir le développeur *ElaScript* des répercussions de l'action d'adaptation SO_{infra} (i.e. *soi*) en termes de temps de reconfiguration et de lui suggérer textuellement de profiter de cette durée incompressible pour paralléliser d'autres instructions.

```
public static val OPERATION_INFO = 'infoOperation'

@Check
def giveInfoSII(Action action) {
  if (action.scalefunction.name.equals("sii")) {
    info("You will not be able to execute another action on this VM later
        because of the SII action.",
      ElaScriptPackage.Literals.ACTION__SCALEFUNCTION, OPERATION_INFO)
  }
}

@Check
def giveInfoSOI(Action action) {
  if (action.scalefunction.name.equals("soi")) {
    info(
      "The SOI action takes time, perhaps you can take the opportunity to
          parallelize other actions during that time...",
      ElaScriptPackage.Literals.ACTION__SCALEFUNCTION, OPERATION_INFO)
  }
}
```

Code 6.15 – Règles 5 et 6 - suggestions concernant l'utilisation d'actions.

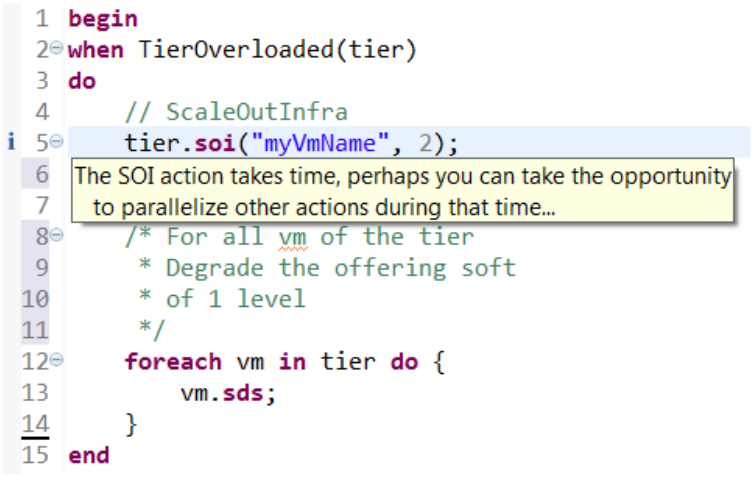


FIGURE 6.9 – Résultat de la Règle 6 sur l'éditeur *Eclipse*.

Règles métier : Le Code 6.16 donne un exemple de règle sémantique visant à interdire l'utilisation d'une action d'adaptation (i.e. *ScaleFunction*) à certains types de ressources. On peut voir qu'il s'agit ici d'interdire (cf. *error(...)*) l'action d'adaptation SO_{soft} (i.e. *sos*) aux ressources de type *Pm* ou *Component*. Le résultat de cette règle au niveau de l'éditeur est donné dans la Figure 6.10.

```
public static val OPERATION_FORBIDDEN = 'errorOperation'

@Check
def checkSOSAction(Action action) {
  val String resourceType = checkResourceType(action.res);
  if (action.scalefunction.name.equals("sos")) {
    switch (resourceType) {
      case "Pm": {
        error("SOS action can't be executed on a Pm resource",
          ElaScriptPackage.Literals.ACTION__SCALEFUNCTION,
              OPERATION_FORBIDDEN)
      }
      case "Component": {
        error("SOS action can't be executed on a Component resource",
          ElaScriptPackage.Literals.ACTION__SCALEFUNCTION,
              OPERATION_FORBIDDEN)
      }
      default: {}}}}
```

Code 6.16 – Règle 7 - limiter l'utilisation d'une action aux types de ressources concernés.

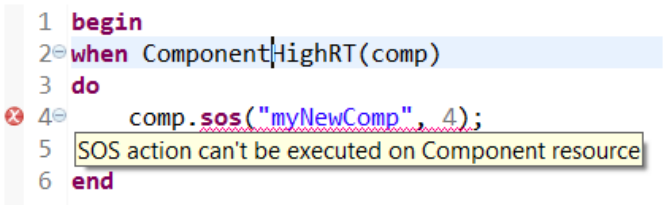


FIGURE 6.10 – Résultat de la Règle 7 sur l'éditeur *Eclipse*.

Enfin, le Code 6.17 et la Figure 6.11 présentent la règle métier évoquée précédemment concernant l'utilisation de l'action SI_{infra} (i.e. *sii*). Pour rappel, il s'agit de générer une erreur lorsqu'une VM exécutant une action *sii* est utilisée ultérieurement pour une autre action (i.e. ce qui n'a pas de sens du fait que l'action *sii* va arrêter la VM). Le but de cette règle est donc de rendre l'action *sii* bloquante pour la ressource concernée (i.e. empêcher un appel ultérieur ou parallèle à une autre action depuis cette ressource).

```
public static val RESOURCE_LOCKED_AFTER_SII = 'resourceLocked'

@Check
def checkResourceActionAfterSIIAction(Action action) {
  var EObject myContainer = action.eContainer;
  if (myContainer instanceof Body) {
    for (BodyPart part : (myContainer as Body).body1) {
      if (part instanceof Action) {
        if (part.equals(action)) {
          return
        }
        if ((part as Action).scalefunction.name.equals("sii"))
          if ((part as Action).res.name.equals(action.res.name))
            error(action.res.name + " is locked: can't execute another
                action on this resource after SII action.",
              ElaScriptPackage.Literals.ACTION__RES,
                 RESOURCE_LOCKED_AFTER_SII);}}}}
```

Code 6.17 – Règle 8 - bloquer une VM suite à l'action $ScaleIn_{infra}$.

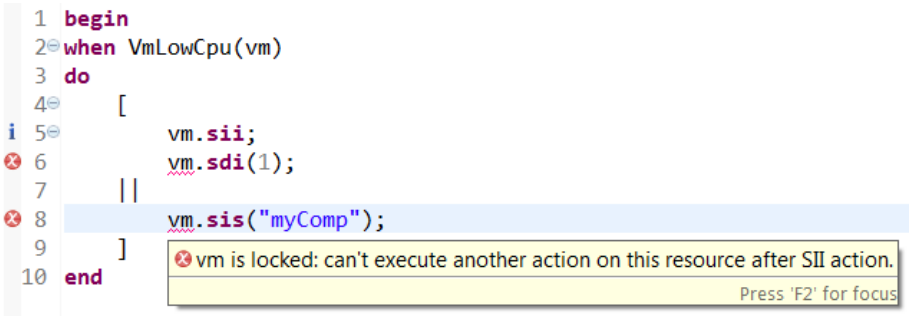


FIGURE 6.11 – Résultat de la Règle 8 sur l'éditeur *Eclipse*.

Exemples de définition de *tactiques* d'élasticité avec *ElaScript*

Nous avons présenté dans la sous-section 5.2.2 que l'*adaptation* de notre *framework* autonomique reposait sur la définition de *tactiques d'élasticité*. Il s'agit concrètement de plans de reconfiguration définis sous forme de *script* avec notre *DSL ElaScript*. Nous allons donner ci-dessous trois exemples de *tactiques* d'élasticité définies avec le langage *ElaScript*. L'*Exemple 1* et l'*Exemple 2* donnés dans les Figures 6.12 et 6.13, correspondent à deux *tactiques* face aux *symptômes* de type *VmHighCpu(vm)*. Ce type de *symptôme* a été défini par le biais de notre *framework perCEPtion* (issu d'un type d'événement simple retournant une unique ressource). Il s'agit ici de remonter une ressource de type *Vm* admettant une consommation CPU trop élevée. Face à un *symptôme* de type *VmHighCpu*, le script de l'*Exemple 1* va simplement faire appel à l'*élasticité logicielle verticale* en diminuant l'Off_{soft} (i.e. *sds*) des composants de la *vm* (cf. ligne 6) de manière permanente (i.e. jusqu'à une future reconfiguration). L'*Exemple 2* va quant à lui avoir recours à l'élasticité verticale de l'infrastructure. Il s'agit ici d'augmenter l'Off_{infra} de la *vm*, dont la valeur courante est découverte à l'exécution, de 2 niveaux (cf. ligne 5) en passant par exemple d'une *small instance* à une *large instance*. Ce changement d'Off_{infra} est réalisé pour une durée de *10 min* (cf. ligne 7), ce après quoi l'Off_{infra} est re-basculée dans son mode nominal (cf. ligne 9) en repassant par exemple en *small instance*. Les deux *tactiques d'élasticité* décrites ici répondent toutes les deux au même type de *symptômes* (i.e. *VmHighCpu*). Nous verrons dans le chapitre 7 comment le gestionnaire autonomique va choisir le "meilleur" *script* à exécuter en fonction du contexte d'exécution et de préférences émises par l'administrateur humain concernant l'adaptation du système.

```
1 begin
2 when VmHighCpu(vm)
3 do
4     // Degrade the offering soft of one level
5     // for all vms's components
6     vm.sds;
7 end
```

FIGURE 6.12 – Exemple 1 : *tactique ElaScript*.

```
1 begin
2 when VmHighCpu(vm)
3 do
4     // Increase the vm's offering infra of 2 levels
5     vm.sui(2);
6     // During 10 minutes (600 seconds)
7     wait(600);
8     // Then switch back to the nominal offering infra
9     vm.sdi(2);
10 end
```

FIGURE 6.13 – Exemple 2 : *tactique ElaScript*.

L'*Exemple 3*, donné dans la Figure 6.14, correspond à une *tactique* répondant aux *symptômes* du type *TierHighRT_VmsOverloaded(tier, vms)*. Ce type de *symptôme* est issu d'un type d'événement complexe composant/agrégeant d'autres événements (définis avec *perCEPtion*). Comme son nom et sa signature l'indique, celui-ci va remonter un *tier* admettant des temps de réponse trop élevé (i.e. *response time RT*) ainsi que la liste des *vms* surchargées de ce tier. Face à ce type de *symptôme*, notre *script* indique que l'on va ajouter au *tier* 2 instances de VM avec une Off_{infra} *small* (cf. ligne 7) au travers de l'action *soi*. On remarquera au passage l'éditeur qui suggère de paralléliser cette instruction avec d'autres opérations du fait que l'action *soi* admet un temps de reconfiguration considérable. On peut imaginer que le développeur a tenu compte de cette suggestion en appliquant la création de deux blocs parallèles (cf. lignes 5, 11 et 16).

Le *script* indique alors que toutes les *VMs* surchargées vont voir leur Off_{soft} baisser de 1 niveau (cf. lignes 13-15) pendant une durée indéterminée, qui va correspondre à la durée de mise en place du *soi* évoqué précédemment. En effet, la synchronisation des deux blocs d'instructions parallèles (cf. ligne 16) correspond au point de rencontre des deux blocs, signifiant que les deux processus (i.e. *threads Java*) doivent avoir terminés leur exécution avant de poursuivre celle du *script*. Il est fort à parier que l'exécution du premier bloc va prendre davantage de temps que celle du second bloc (i.e. à cause du *soi*). En ce sens, le second bloc va attendre que le premier bloc ait terminé son exécution (i.e. que les deux instances soient effectivement démarrées et prêtes à recevoir des requêtes).

Une fois le point de rencontre atteint (cf. ligne 16), le *script* indique que l'on va re-basculer l'Off_{soft} des VMs dans leur mode nominal (cf. lignes 18-20). Le *script* mis en place dans cet exemple permet de soulager les *vms* surchargées, en faisant appel à l'*élasticité logicielle verticale*, le temps de passer à l'échelle l'infrastructure de manière horizontale (i.e. *soi*). En ce sens, ce *script* bénéficie de la synergie des différents types d'élasticité. Nous reviendrons plus tard dans ce manuscrit (cf. chapitre 8) sur les gains de l'utilisation conjointe de ces deux dimensions d'élasticité.

```
1  begin
2  when TierHighRT_VmsOverloaded(tier, vms)
3  do
4      // Execute in parallel
5      [
6          // Add 2 small instances to the tier
7          tier.soi(2, "small");
8          The SOI action takes time, perhaps you can take the opportunity
9           to parallelize other actions during that time...
10
11     ||
12         // Degrade the offering soft of one level for all overloaded vms's components
13          foreach vm in vms do {
14             vm.sds;
15          }
16     ]
17      // Switch back (upgrade) the offering soft to nominal state
18     foreach vm in vms do {
19         vm.sus;
20     }
21 end
```

FIGURE 6.14 – Exemple 3 : *tactique ElaScript*.

7

Décision de reconfiguration

Dans le chapitre 5, nous avons présenté de manière générale notre gestionnaire autonomique *MAPE-K* ainsi que les différents éléments constituants le système dont on attend un comportement autonome. Nous avons développé davantage les phases *Monitor*, *Plan* et *Execute* dans le chapitre 6 en fournissant notamment l'outillage nécessaire à leur gestion et leur paramétrage. Ce chapitre vise donc à présenter la phase d'*Analyze*, qui à partir des données issues de l'observation (i.e. remontées par le *M*) et des politiques globales de gestion définies au préalable (i.e. stockées dans le *Knowledge*), est chargée de la prise de décision sur les changements à apporter au système.

Dans ce chapitre, nous présentons dans un premier temps notre modèle de décision dans la section 7.1 en rappelant et précisant comment s'articule la phase d'*Analyze* avec les autres phases du gestionnaire autonomique. Puis, nous exposons dans les sections 7.2, 7.3 et 7.4 les différentes étapes (i.e. filtres successifs) constituant le processus de décision de notre gestionnaire autonomique. Enfin, nous abordons comment mettre en place une gestion parallèle des décisions de reconfiguration dans la section 7.5 sous forme de pistes exploratoires.

Contents

7.1 Modèle de décision : la phase d'analyse

Dans cette section, nous allons détailler comment s'intègre la phase d'*Analyze* avec les différentes composantes de notre gestionnaire autonomique *MAPE-K*. De plus, nous présentons le fonctionnement général du processus de décision de notre *gestionnaire* autonomique qui est constitué de 3 étapes prenant la forme de *filtres* successifs. Il s'agit, à partir d'un *symptôme* survenant à l'exécution, d'identifier la *tactique* à appliquer parmi le catalogue de *tactiques d'élasticité* proposé en amont par l'administrateur humain et définissant les adaptations possibles et souhaitées du système.

7.1.1 Entrées : interface avec la phase de surveillance

Notre modèle de décision s'interface avec le modèle de surveillance présenté dans la section 6.1 au travers de la queue de *symptômes* (*symptoms queue*). Pour rappel, la phase de *Monitoring* du gestionnaire autonomique est implémentée par notre *framework perCEPtion*, qui est chargé de remonter tous les *symptômes* devant être analysés en les poussant (*push*) dans la *symptoms queue* (cf. sous-section 6.1.2).

La phase d'*Analyze* va avoir accès à une liste triée et purgée de *symptômes* nécessitants une adaptation du système. Il s'agit alors de traiter les *symptômes* un à un en dépilant (*pull*) de manière itérative la tête de queue. La phase d'*Analyze* a la garantie de traiter des *symptômes* d'"actualité" du fait que la *symptoms queue* est maintenue à jour avant chaque interaction (en écriture, *push* par le *M* et en lecture, *pull* par le *A*) avec celle-ci (cf. Code 6.8). La récupération d'un *symptôme* (i.e. tête de la *symptoms queue*) déclenche le processus de décision que nous détaillerons dans la section 7.2.

7.1.2 Sorties : interface avec la phase de planification

Dans le cas de notre *framework* autonomique, les données d'observation sont remontées sous forme de *symptômes*, signes d'instabilité du système, tandis que les actions possibles à entreprendre face au contexte d'exécution (i.e. porté par le *symptôme* généré) sont représentées par le concept de *tactiques d'élasticité*.

Nous définissons une *tactique d'élasticité* comme un plan de reconfiguration générique spécifiant une adaptation possible face à un type de *symptôme*. En ce sens, plusieurs adaptations peuvent être définies face à un même type *symptôme*. Il s'agit concrètement d'un *script* de reconfiguration "générique" portant sur une ou plusieurs ressources (remontées par le *symptôme*) pouvant faire intervenir une ou plusieurs actions d'adaptation. Les actions d'adaptation sont ordonnées et peuvent concerner différentes dimensions d'élasticité (i.e. HS_{infra}, VS_{infra}, HS_{soft} et VS_{soft}).

Les *tactiques d'élasticité* sont définies au préalable par l'administrateur à l'aide de notre langage dédié *ElaScript* (cf. section 6.2). L'ensemble des *tactiques* forme le catalogue des adaptations possibles à mettre en place sur le système. Ainsi, notre phase d'*Analyze* revient à sélectionner la *tactique* à appliquer au système en vue de le ramener dans un état "stable" (i.e. cohérent).

Le choix de la *tactique* à exécuter est réalisé par le processus de décision à partir du contexte d'exécution et de préférences émises par l'administrateur humain concernant l'adaptation du système. La *tactique* va ainsi être "instanciée" (i.e. avec les ressources réelles) après avoir été confrontée aux contraintes du système (e.g. état des ressources, *SLO*, limite budgétaire, etc.) ainsi qu'aux préférences d'adaptation de l'administrateur.

Le résultat du processus de décision d'adaptation va être un plan de reconfiguration prêt à être exécuté sur le système. En ce sens, notre modèle de décision intègre à la fois les phases d'*Analyze* et de *Plan* du gestionnaire autonomique *MAPE-K*. Il s'agit alors d'appliquer au système la *tactique* "instanciée" avec les informations *runtime*. Cette tâche incombe à la phase *Execute* de la boucle autonomique.

7.1.3 Base de connaissance

Les différentes phases de la boucle autonomique *MAPE-K* sont reliées à la base de connaissances (*Knowledge - K*). Pour rappel, le *K* peut contenir des informations diverses et variées (e.g. historiques de la phase d'observation, reconfigurations passées, politiques de gestion, etc.) et être peuplé automatiquement (e.g. par les 4 phases *MAPE* du gestionnaire autonomique) ou manuellement (i.e. administrateur humain).

Dans le cas de notre *framework* autonomique, la base de connaissances va contenir des informations rajoutées par l'administrateur humain en vue d'aider le gestionnaire autonomique dans sa prise de décision. En effet, bien que l'on souhaite rendre le système autonome et capable de s'*auto-gérer* [KC03], le rôle de l'administrateur humain reste essentiel.

Nous avons déjà évoqué les interactions du *K* avec le *M* au travers de la description des *symptômes* définis puis évalués par le biais du *framework perCEPtion* (cf. sous-section 6.1.2) ainsi que ses interactions avec le *P* et le *E* à savoir la définition de l'ensemble des *tactiques d'élasticité* associées à ces *symptômes* (cf. sous-section 6.2.2).

Les *tactiques d'élasticité* et le *mapping symptôme-adaptation* qui en résulte va permettre d'identifier l'ensemble des adaptations possibles (i.e. *éligibles*) pour un *symptôme* survenant à l'exécution (cf. section 7.2). De plus, le *K* va contenir certaines contraintes *runtime* du système (e.g. état courant des ressources, *SLO*, limite budgétaire, etc.), qui vont empêcher d'appliquer certaines *tactiques éligibles* et ainsi permettre d'identifier les *tactiques applicables* (cf. section 7.3). Enfin, le *K* va contenir des informations relatives aux préférences d'adaptation de l'administrateur humain, sous forme de *stratégies d'élasticité*, qui vont orienter la prise de décision et permettre d'identifier la "meilleure" *tactique* à appliquer au système (cf. section 7.4).

7.1.4 Gestionnaire autonomique : cycles de vie

La Figure 7.1 donne une illustration de notre gestionnaire autonomique. On retrouve les différentes tâches de la boucle *MAPE-K* ainsi que la *symptoms queue*, jouant le rôle d'interface entre le *M* et le *A*. De plus, on remarquera les catalogues de *symptômes* et de *tactiques* venant peupler le *K*.

La Figure 7.1 met en lumière les *cycles de vie* du gestionnaire autonomique à savoir le cycle de la phase de surveillance *M-cycle* (cf. zone verte) et le cycle des phases de décision et d'adaptation *APE-cycle* (cf. zone rouge). Le fonctionnement est le suivant : le *M-cycle* peut être vu comme un *thread* s'exécutant en permanence (i.e. cycle court) et dont le but est d'identifier les *symptômes* survenant à l'exécution et de les remonter dans la *symptoms queue* pour traitement (cf. *push*). L'*APE-cycle*, quant à lui, ne s'exécute pas en continu mais uniquement lorsque le *A* récupère l'événement en tête de la *symptoms queue* pour traitement (cf. *pull*). Cela déclenche alors un cycle complet de reconfiguration (i.e. *APE*) aboutissant à l'adaptation du système. En ce sens, la communication entre les deux cycles peut être qualifiée de *asynchrone*.

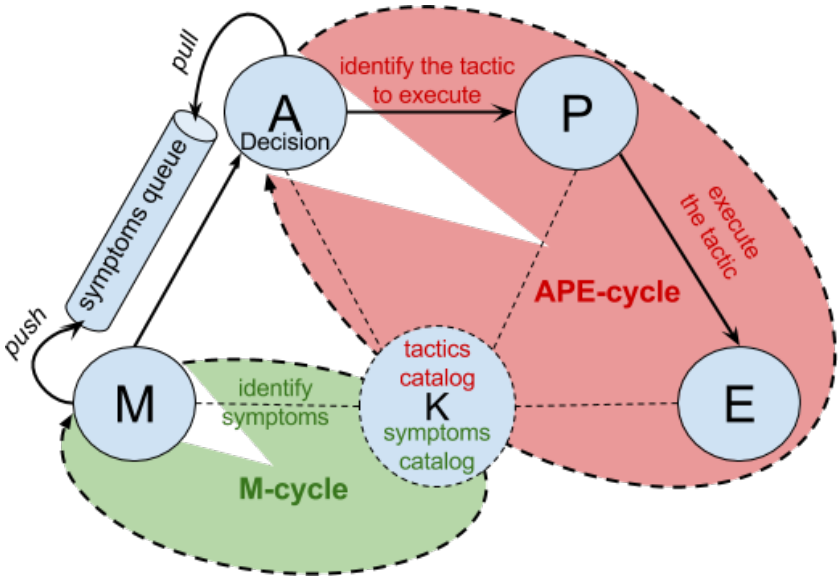


FIGURE 7.1 – Cycles du gestionnaire autonomique.

La Figure 7.2 illustre les interactions entre les différentes tâches du gestionnaire autonomique au travers d'un diagramme de séquence *UML*. On y retrouve aussi la *symptoms queue* permettant la communication asynchrone entre les deux cycles (cf. *loop*). Dans la version évoquée ici, les *symptômes* sont traités un par un, ce qui limite l'adaptation du système à un seul *APE-cycle* à la fois (i.e. *thread* unique). Nous verrons plus tard dans la section 7.5 qu'il est possible d'envisager plusieurs *APE-cycle* s'exécutant en même temps (i.e. *threads* parallèles) bien que cela nécessite de contrôler les ressources impactées par les différents cycles (i.e. parties du graphe).

7.1.5 Processus de décision : fonctionnement général

La Figure 7.3 donne une vision globale du processus de décision de notre *gestionnaire* autonomique. On retrouve le *symptôme* comme point d'entrée du processus (cf. *Symptom*). La décision va revenir à choisir la meilleure *tactique* (cf. *Executed Tactic*) face à un tel *symptôme*, parmi les différentes *tactiques d'élasticité* définies au préalable par l'administrateur humain (avec le langage *ElaScript*) et stockées dans le *Knowledge*.

Le processus de décision est constitué de 3 étapes distinctes jouant le rôle de filtres successifs s'appliquant au catalogue de *tactiques* (cf. *Tactics Filter*, *Context Filter* et *Strategy Filter*). Nous allons présenter brièvement le but de ces trois filtres ci-dessous :

— *Tactics Filter :* Il s'agit ici de filtrer, parmi le catalogue de *tactiques d'élasticité*, les *tactiques* répondant au *symptôme* poussé par la phase de *surveillance*. Cette première étape va s'appuyer sur le *mapping symptôme-adaptation* (cf. *(1) Tactics catalog*) et retourner l'ensemble des *tactiques éligibles* (i.e. indiquées comme pouvant répondre au problème porté par le *symptôme*) ;

— *Context Filter :* Parmi les *tactiques éligibles*, certaines peuvent ne pas être applicables du fait de contraintes liées au contexte courant du système (e.g. état des ressources). Pour rappel, nous avons évoqué qu'une *tactique d'élasticité* constituait un plan de reconfiguration générique non instancié. Une fois le *symptôme* généré et les informations contextuelles récupérées (e.g. ressources concernées), il s'agit de savoir s'il est possible d'"instancier" ou non l'adaptation.

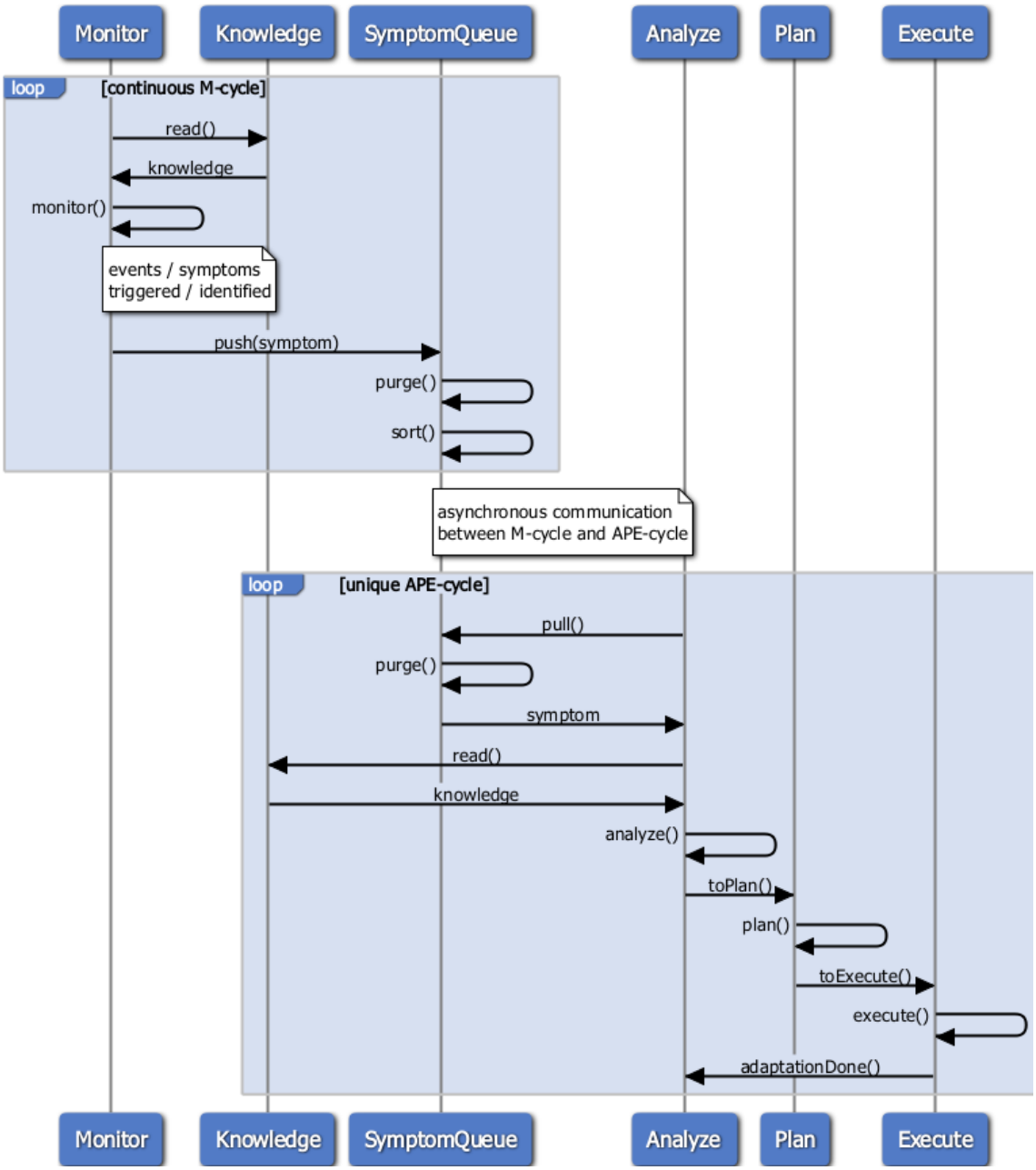


FIGURE 7.2 – Interactions entre les phases du gestionnaire autonomique.

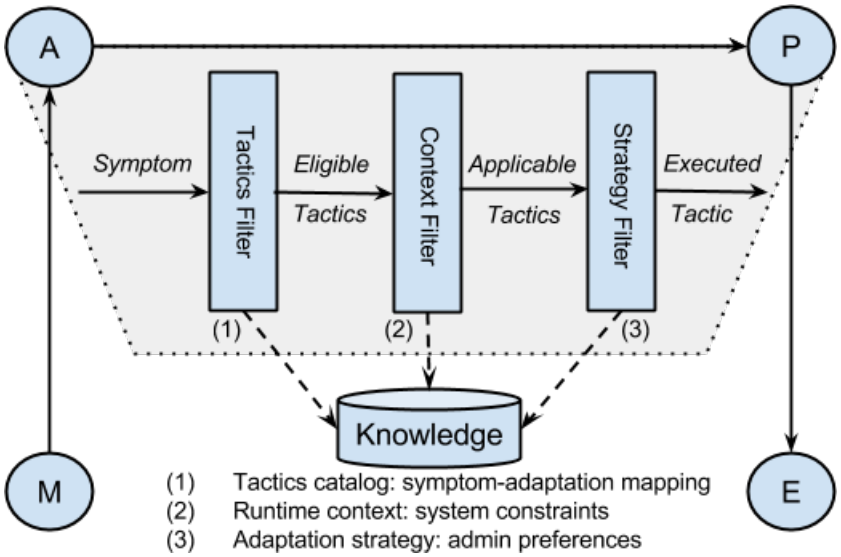


FIGURE 7.3 – Processus de décision : filtres.

Par exemple, si une tactique contient l'instruction $vm.SU_{infra}$ mais que la vm est déjà paramétrée avec l'Off_{infra} maximum (empêchant d'appliquer le SU_{infra}), cette tactique n'est pas instanciable. Nous verrons plus tard que les contraintes, stockées dans le *K* (cf. *(2) Runtime context*) peuvent être diverses et variées (e.g. limite budgétaire, *SLO*, etc.). Le *Context Filter* va filtrer parmi les *tactiques éligibles* les tactiques qui ne peuvent être exécutées en fonction des contraintes *runtime* (i.e. contexte d'exécution) et ainsi retourner l'ensemble des *tactiques applicables* (i.e. satisfaisants les contraintes du système) ;

— *Strategy Filter :* Cette dernière étape peut être qualifiée de optionnelle du fait qu'elle est nécessaire uniquement si plusieurs *tactiques applicables* sont retournées par l'étape précédente. En effet, le *Context Filter* va retourner la ou les *tactiques* (ou aucune si le système est fortement contraint) dont on sait qu'elle(s) offre(nt) une réponse au *symptôme* étant survenu. Néanmoins, dans le cas où plusieurs *tactiques* sont *applicables*, laquelle choisir ? Laquelle est la "meilleure" ? Selon quels *critères* ? Ce dernier filtre vise à répondre à ces questions.
Le *Strategy Filter* va intégrer les préférences de l'administrateur humain dans le but d'orienter la prise de décision du gestionnaire autonomique (cf. *(3) Adaptation strategy*). Le résultat de cette dernière étape va retourner l'unique *tactique* qui sera exécutée.
Le fonctionnement est le suivant : chaque *tactique* est qualifiée en fonction de différents *critères* définis au préalable (cf. sous-section 5.1.3). Les préférences sont représentées sous forme de *stratégie d'élasticité* dans lesquelles l'administrateur va préférer certains *critères* d'adaptation plutôt que d'autres (e.g. *coût* > *QoS* > *QoF*). Au travers de ces *stratégies*, l'administrateur va indiquer comment résoudre les compromis induits par l'élasticité multi-couche (cf. sous-section 5.1.4) et ainsi orienter la prise de décision du gestionnaire autonomique.

La Figure 7.4 offre une vision globale des étapes évoquées ci-dessus ainsi que les interactions induites par le processus de décision sous forme de diagramme de séquence *UML*. Nous allons détailler par la suite les trois étapes de notre processus du décision à savoir le *Tactics Filter* (cf. section 7.2), le *Context Filter* (cf. section 7.3) et le *Strategy Filter* (cf. section 7.4).

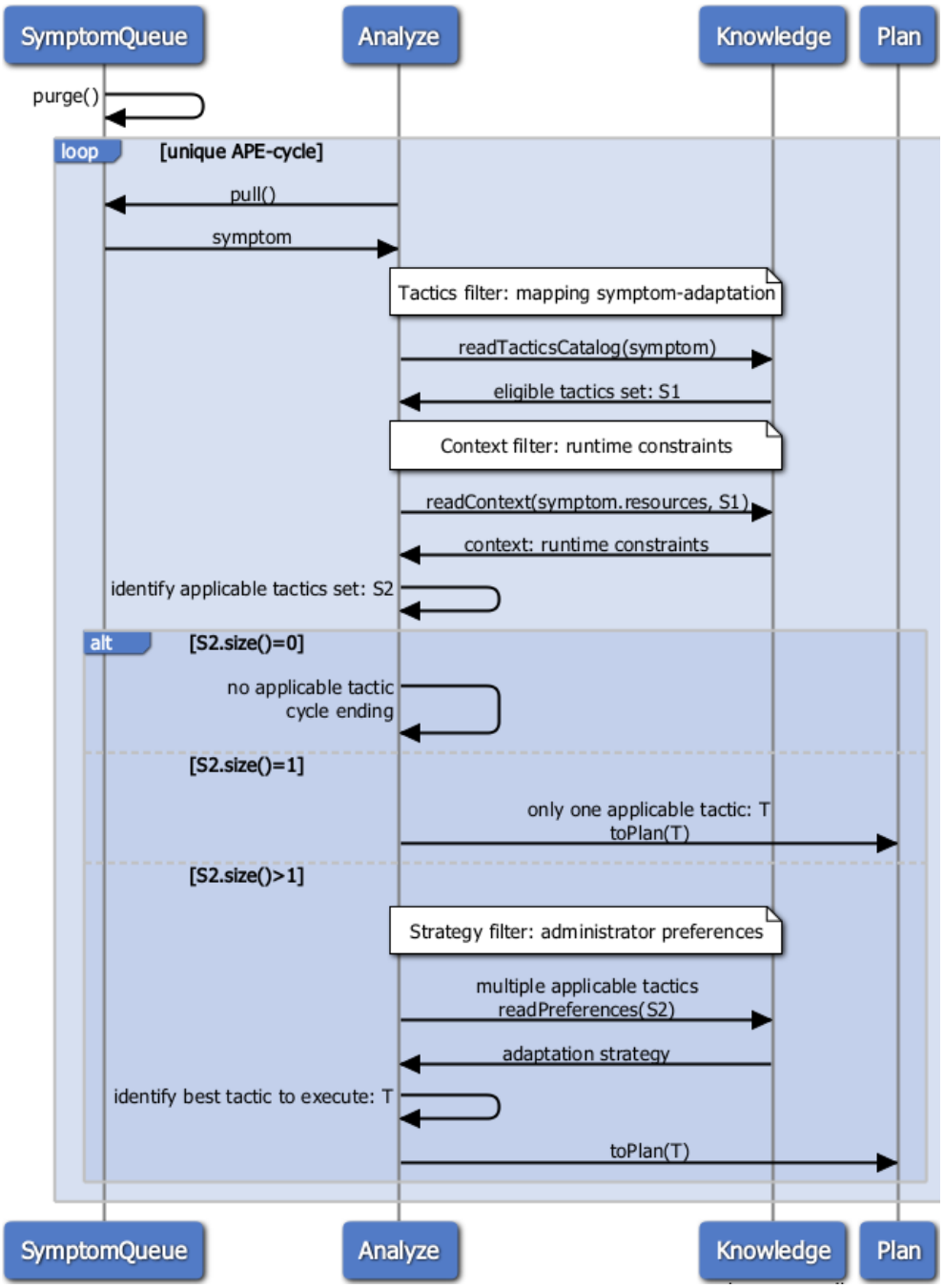


FIGURE 7.4 – Processus de décision : interactions.

7.2 Mapping symptôme-adaptations : *tactiques d'élasticité*

7.2.1 Motivations

L'administrateur humain, en se basant sur son expertise du domaine et certaines logiques métiers ou architecturales, est en mesure de définir les adaptations qu'il souhaite entreprendre sur le système face à des situations récurrentes (i.e. *symptômes*) qui ont elles-mêmes été identifiées et définies en amont (par lui-même ou par un autre expert).

Cette phase de *design* de *symptômes* et de *tactiques* permet de réduire considérablement le champ des possibles simplifiant ainsi la décision du gestionnaire autonomique. Bien que nous ayons admis que les *symptômes* et les *tactiques* étaient définis en amont, l'administrateur a la possibilité d'ajouter de nouveaux *symptômes* et de nouvelles *tactiques*, de les supprimer si besoin ou encore de les modifier (e.g. changer le seuil d'un *événement* ou rajouter une instruction à une *tactique* existante) rendant ainsi la solution évolutive. De plus, le modèle d'adaptation de notre *framework* autonomique permet d'écrire des plans de reconfiguration génériques sous forme de *tactiques* offrant ainsi la possibilité de *réutiliser* ce code (i.e. *scripts*) pour différentes applications.

7.2.2 Spécifications

Le principe du premier filtre du processus de décision est très simple. En effet, le traitement réalisé à l'exécution est minimal du fait que le *mapping symptôme-adaptation* a été effectué en amont lors de l'écriture des *tactiques* par l'administrateur. Il s'agit simplement de récupérer, parmi le catalogue de *tactiques* stocké dans le *K*, celles correspondant au *symptôme* en entrée du processus. Nous désignons les *tactiques* ainsi retournées de *tactiques éligibles*, c'est-à-dire ayant été désignées par l'administrateur humain comme répondant à la "maladie" portée par le *symptôme*.

7.2.3 Implémentation

Notre implémentation du *mapping* repose sur un simple modèle *XML*, maintenu à jour par l'administrateur en s'appuyant sur les *tactiques* définies au préalable avec le langage *ElaScript* et stockées dans des fichiers ayant l'extension *.elas*. La *DTD* (i.e. *Document Type Definition* [dtd16]) et un extrait du modèle *XML* sont donnés dans les Codes 7.1 et 7.2.

```
<!ELEMENT mapping (#PCDATA | symptom)*>
<!ELEMENT symptom (#PCDATA | sdescription | tactics)*>
<!ATTLIST symptom sname CDATA #IMPLIED>
<!ELEMENT sdescription (#PCDATA)>
<!ELEMENT tactics (#PCDATA | tactic)*>
<!ELEMENT tactic (#PCDATA | path | tdescription)*>
<!ATTLIST tactic tname CDATA #IMPLIED>
<!ELEMENT path (#PCDATA)>
<!ELEMENT tdescription (#PCDATA)>
```

Code 7.1 – *DTD* du modèle *XML*.

Le modèle *XML* est interrogé à l'exécution (i.e. après récupération de l'objet *Symptom* dans la *symptoms queue*), sous forme de requête *XPath* (i.e. *XML Path Language* [xpa16]) comme indiqué dans le Code 7.3. Il s'agit ici de rechercher le type du *symptôme* dans le modèle *XML* (cf. *symptom[@sname="VmHighCpu(Vm vm)"]*) et de retourner les chemins vers les fichiers *.elas* des *tactiques éligibles* correspondantes (cf. */tactics/tactic/path/text()*). Le résultat de la requête est donné dans le Code 7.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapping SYSTEM "file:../STmapping/mapping.dtd">
<mapping>
   <symptom sname="VmHighCpu(Vm vm)">
      <sdescription>
         vm overloaded: CPU consumption of the vm is to high
      </sdescription>
      <tactics>
         <tactic tname = "suiOnly">
            <path>
               //path/to/my/tactic/file/suiOnly.elas
            </path>
            <tdescription>
               increase the vm's offeringInfra
            </tdescription>
         </tactic>
         <tactic tname = "sdsOnly">
            <path>
               //path/to/my/tactic/file/sdsOnly.elas
            </path>
            <tdescription>
               decrease the vm's offeringSoft
            </tdescription>
         </tactic>
      </tactics>
   </symptom>
   <!-- ... -->
   <symptom sname="TierHighRT(Tier tier)">
   <!-- ... -->
   </symptom>
</mapping>
```

Code 7.2 – Extrait du modèle *XML* : *mapping*.

```
$doc/mapping/symptom[@sname="VmHighCpu(Vm vm)"]/tactics/tactic/path
```

Code 7.3 – Exemple de requête *XPath* sur le modèle *XML*.

```
//path/to/my/tactic/file/suiOnly.elas
//path/to/my/tactic/file/sdsOnly.elas
```

Code 7.4 – Résultat de la requête *XPath*.

Nous avons évoqué que le *mapping* (i.e. modèle *XML*) était réalisé et maintenu par l'administrateur humain. Cette tâche de paramétrage du gestionnaire autonomique peut sembler lourde, cependant, il est possible de l'automatiser. Une évolution possible serait donc de générer automatiquement le *mapping* et de le maintenir à jour par le biais d'un programme chargé de parcourir tous les fichiers *.elas* (e.g. stockés et regroupés dans un répertoire) et de les *parser* afin d'identifier les *symptômes déclencheurs* (i.e. signature du *symptôme* après le mot clé *when*).

7.2.4 Exemple

Nous allons présenter ici un simple scénario dont le but est d'illustrer les différentes étapes de notre processus de décision. Nous conserverons cet exemple pour les 3 filtres constituant le processus de décision en vue de voir la chronologie des filtres et leurs impacts respectifs.

La Figure 7.5 illustre schématiquement le *mapping* réalisé par l'administrateur humain, à savoir le modèle *XML* présenté précédemment. On retrouve un ensemble de *symptômes* définis avec le *framework perCEPtion*, un ensemble de *tactiques* définies avec le langage *ElaScript* et le *mapping* sous forme de relations entre les éléments de ces deux ensembles.

Pour des raisons de place, les *tactiques* sont représentées par des noms succincts laissant présager les adaptations correspondantes (i.e. actions de dimensionnement). De même, toutes les relations entre les deux ensembles ne sont pas décrites ici pour des raisons de lisibilité. À ce stade, le processus de décision n'est pas encore amorcé, il s'agit de la définition du catalogue de *tactiques* qui fait partie du paramétrage du gestionnaire autonomique réalisé en amont par l'administrateur humain (i.e. *design*).

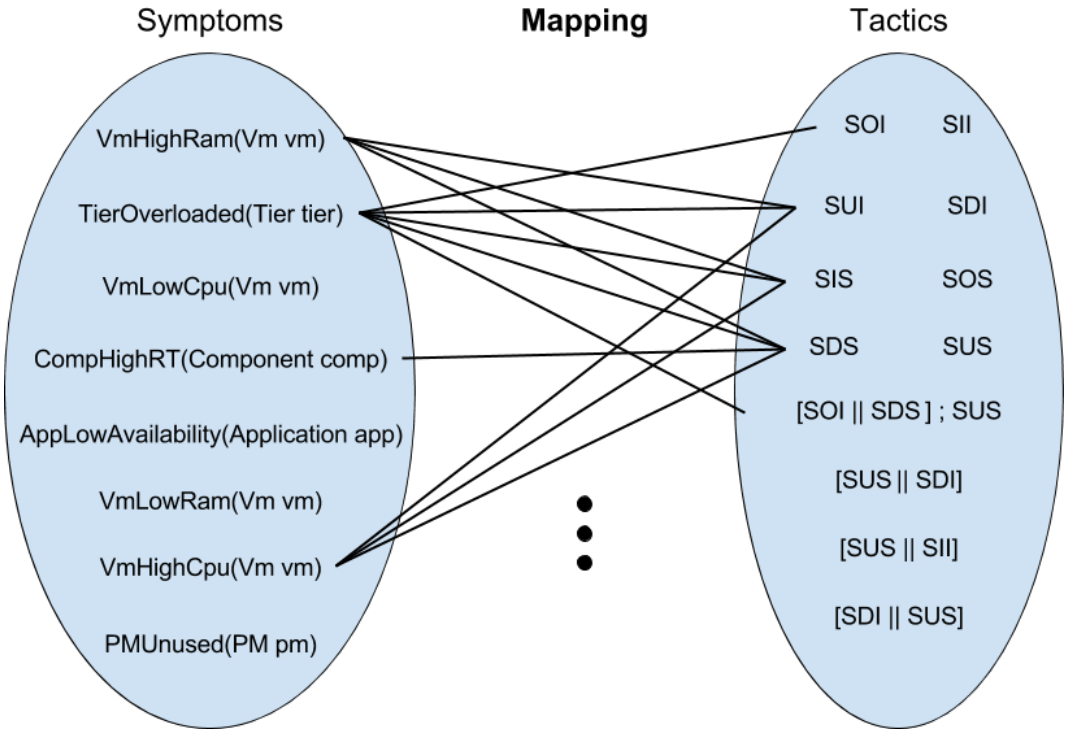


FIGURE 7.5 – Scénario d'exemple : définition du catalogue de *tactiques*.

La Figure 7.6 offre une vision schématique du *Tactics Filter*, première étape du processus de décision, déclenché par la récupération d'un *symptôme* dans la *symptoms queue*. Dans cet exemple, le *symptôme* récupéré est de type *TierOverloaded(Tier tier)*. Ce type de *symptôme* indique un tier surchargé (e.g. consommation moyenne de CPU des VMs sous-jacentes trop importante).

Il s'agit alors de requêter le modèle *XML* (i.e. *mapping*) afin d'obtenir les *tactiques éligibles* face à ce *symptôme* (cf. en noir sur la figure). En procédant de la sorte, les 5 *tactiques éligibles* suivantes sont retournées (i.e. correspondants à 5 fichiers ayant l'extension *.elas*) :

— *SOI* : ajouter une VM au tier ;
— *SUI* : augmenter l'Off_{infra} des VMs du tier ;
— *SIS* : retirer un composant au tier ;
— *SDS* : diminuer l'Off_{soft} (i.e. *QoF*) des VMs du tier ;
— *[SOI || SDS] ; SUS* : adaptation multi-couche visant à diminuer l'Off_{soft} des VMs du tier durant l'ajout d'une nouvelle instance (i.e. en parallèle) puis de rebasculer l'Off_{soft} des VMs à leur mode initial (i.e. après *synchronisation* des blocs d'instructions parallèles).

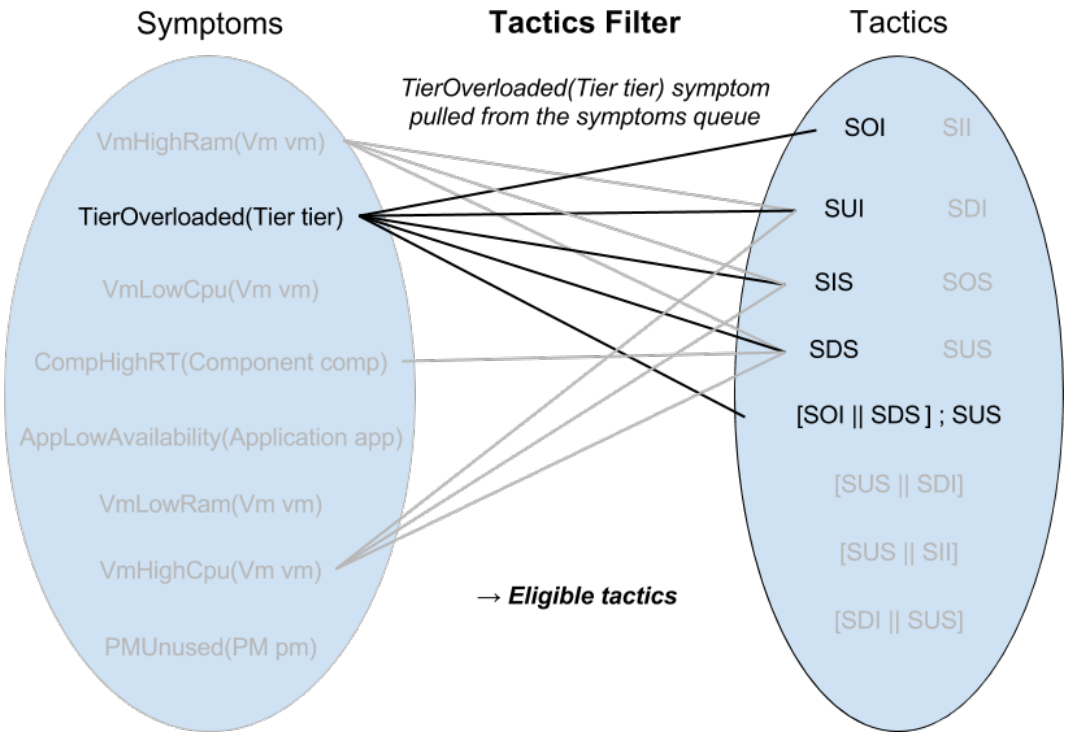


FIGURE 7.6 – Scénario d'exemple : *Tactics Filter*.

7.3 Prise en compte du contexte : contraintes à l'exécution

7.3.1 Motivations

Nous avons évoqué que le langage *ElaScript* permettait d'assurer l'écriture de plans de reconfiguration sémantiquement corrects (cf. sous-section 6.2.3). Pour rappel, l'analyse sémantique étant statique (i.e. réalisée au moment de la compilation), elle ne peut pas exclure les programmes dont on ne peut savoir s'ils sont sémantiquement corrects qu'à l'exécution. Dans notre cas, les ressources intervenants dans un *script* sont découvertes à l'exécution du système (i.e. au moment de la récupération du *symptôme*), nécessitant ainsi de contrôler le contexte de manière dynamique.

L'objectif du second filtre est de prendre en considération le contexte d'exécution du système (i.e. contraintes *runtime*) en vue d'identifier les *tactiques applicables* parmi les *tactiques éligibles*. Le système, qui évolue dans un environnement dynamique, est sujet à certaines contraintes qui vont induire des limitations en termes d'adaptation.

Les contraintes sont diverses et variées et peuvent concerner l'état courant des ressources, leurs relations dans le graphe, la capacité maximum de l'infrastructure, une limite budgétaire fixée par le fournisseur ou encore des contrats de niveaux de services à respecter, etc. Les contraintes peuvent être définies au préalable par l'administrateur humain (e.g. *SLO* portant sur la *QoF* stocké dans le *K*) ou tout simplement être le fruit de l'évolution du système dans le temps (e.g. état *runtime* du système). Le but du second filtre est d'évaluer ces contraintes à l'exécution et d'identifier si celles-ci empêchent l'exécution de certaines *tactiques éligibles* (i.e. filtrer les *tactiques non applicables*).

7.3.2 Spécifications

Nous appelons notre second filtre *Context Filter*, du fait qu'il s'agit ici de considérer le contexte d'exécution dans la prise de décision d'adaptation. Dans notre cas, le contexte est constitué d'informations, portées par les ressources elles-mêmes (i.e. ressources remontées par le *symptôme*) ou stockées dans le *K*. Nous distinguons deux types de contraintes :

— *les contraintes imposées par le système* : il s'agit des limites d'adaptation résultants de la configuration *runtime* du système. En ce sens, ce type de contrainte est étroitement lié à l'état des ressources à l'exécution. La configuration du système (i.e. ressources sous-jacentes) va ainsi contraindre les possibilités d'adaptation. On peut par exemple citer une capacité maximum de l'infrastructure (e.g. nombre de PMs, de cœurs CPU, etc.), la configuration courante d'une VM paramétrée avec l'Off_{infra} maximum (e.g. *large instance*) limitant le passage à l'échelle verticale de l'infrastructure (i.e. action SU_{infra}) ou encore un composant paramétré avec l'Off_{soft} minimum qui va condamner l'exécution d'un SD_{soft} (i.e. diminuer la *QoF*) ;

— *les contraintes définies par l'administrateur humain* : contrairement aux contraintes imposées par le système, il s'agit ici de contraintes spécifiées par l'administrateur sous forme d'intentions. Ces contraintes peuvent concerner des aspects architecturaux (e.g. limiter le nombre de VMs pour un tier, c'est-à-dire le nombre de nœuds du sous-graphe), métiers (e.g. limiter l'utilisation d'un composant spécifique à un profil de client), contractuels (e.g. SLO portant sur l'utilisation des différentes Off_{soft} d'un composant) ou encore financiers (e.g. limite budgétaire fixant le coût d'infrastructure maximum à ne pas dépasser pour un client, une application, etc.).

Les contraintes vont être évaluées en confrontant l'état des ressources remontées par le *symptôme* aux actions d'adaptation contenues dans les *scripts* de reconfiguration *ElaScript* (i.e. *tactiques*). Il s'agit concrètement de parcourir les différentes *tactiques éligibles* (i.e. *parser* les fichiers *.elas* correspondants) en vue de les confronter au contexte d'exécution et ainsi identifier les *tactiques applicables*. Une *tactique applicable* est définie comme une *tactique éligible* dont on sait que son exécution est rendue possible par l'état courant du système (i.e. satisfaisant toutes les contraintes).

L'évaluation des contraintes peut être *locale* ou *globale*. L'évaluation *locale* revient à considérer une unique instruction contenue dans un *script*. On parle alors de *micro-contrôle*. L'évaluation *globale*, quant à elle, correspond au fait de contrôler une *tactique* dans son ensemble (i.e. toutes les instructions qui la constituent). Nous parlons alors de *macro-contrôle*.

Évaluation locale : Pour rappel, les actions d'adaptation avec *ElaScript* sont représentées sous forme de *fonctions*. Une *invocation de fonction* se résume à une *instruction* classique de la forme $variable.fonction$ pour laquelle $variable$ et $fonction$ désignent respectivement une ressource et une action d'adaptation. Une première étape dans l'évaluation des contraintes revient à contrôler, pour chaque *invocation de fonction* contenue dans un *script* (i.e. évaluation *locale*), si l'état de la ressource (i.e. $variable$) permet d'appliquer l'action associée (i.e. $fonction$).

Imaginons une *tactique* contenant l'unique instruction $vm.sui$, indiquant que l'on souhaite augmenter l'Off_{infra} de la vm. Il s'agit ici de contrôler l'état courant de la vm afin de savoir s'il est possible ou non d'appliquer l'action sui (i.e. augmenter l'Off_{infra}). Si l'état de la vm ne permet pas l'exécution du sui (e.g. déjà paramétrée avec l'Off_{infra} maximum), alors la *tactique* est retirée des adaptations possibles (i.e. "désactivée"). Dans le cas contraire, celle-ci est considérée comme une *tactique applicable*. Dans le cas de *tactiques* plus évoluées (i.e. contenant plusieurs instructions), il s'agit de contrôler chaque instruction comme évoqué ci-dessus (i.e. chaque couple $variable.fonction$). Si au moins l'une de ses instructions est inapplicable, alors la *tactique* est considérée comme telle.

Évaluation globale : Prenons une seconde contrainte, *définie par l'administrateur humain*, statuant un budget maximum (i.e. *budgetMax*) à ne pas dépasser concernant le coût de l'infrastructure de l'application (i.e. *invariant*). Il va s'agir ici de contrôler si le contenu des *scripts* fait intervenir les actions SO_{infra} et SU_{infra} (mais aussi les actions SI_{infra} et SD_{infra}) et de calculer si la somme de celles-ci engendre le dépassement de la limite budgétaire fixée (i.e. *(coût infra actuel + coût supplémentaire induit par la tactique) > budgetMax*).

De même que pour l'exemple précédente, le résultat de cette évaluation de contrainte va permettre d'identifier si oui ou non, une *tactique* est *applicable*. Néanmoins, contrairement à l'exemple précédent faisant intervenir un *micro-contrôle* (i.e. au niveau de l'instruction), il s'agit ici d'une évaluation *globale* où la *tactique* est considérée dans son ensemble (i.e. *macro-contrôle*). Ainsi, on va évaluer l'impact de l'exécution de la *tactique* d'un point de vue budgétaire en calculant, à partir des instructions qui la constituent, le *coût* total qu'elle représente.

7.3.3 Implémentation

L'implémentation de notre *Context Filter* revient à *parser* (i.e. analyser) les fichiers *.elas* correspondant aux *tactiques éligibles* et de confronter leurs contenus aux contraintes *runtime* du système. Nous allons détailler ci-dessous les différentes étapes et éléments constitutifs de notre implémentation du *Context Filter* :

— **Génération d'un modèle *XML*** : Lors de la création d'un fichier *.elas* par l'administrateur humain, une représentation *XML* (*Extensible Markup Language*) du *script* correspondant est générée. De même, lorsqu'un *script* est modifié par l'administrateur, le modèle *XML* est mis à jour. Ces opérations de génération et de mise à jour du modèle *XML* sont réalisées au moment de la sauvegarde du fichier *.elas*.
Le modèle au format *XML* est contenu dans un fichier *XMI* (*XML Metadata Interchange*). Il s'agit d'un standard pour l'échange d'informations de métadonnées *UML* basé sur *XML*. Le fichier *.xmi* généré contient des données qui correspondent à des instances du métamodèle de notre langage *ElaScript*.
À titre d'exemple, le Code 7.5 donne le contenu du fichier *.xmi* généré lors de la sauvegarde du fichier *.elas* présenté dans la Figure 7.7. Il s'agit ici d'une simple *tactique* face aux symptômes du type *VmHighCpu(Vm vm)* contenant une seule action d'adaptation portant sur l'unique ressource remontée par le symptôme (cf. $vm.sui$). On retrouve dans le fichier *.xmi* généré, les éléments de notre langage *ElaScript* (e.g. balises *<symptom name="VmHighCpu"/>* ou *<scalefunction name="sui"/>*).

```
<?xml version="1.0" encoding="UTF-8"?>
<elaScript:ElaScript xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:elaScript="http://www.sigma.fr/elascript/ElaScript">
  <script>
    <symptom name="VmHighCpu"/>
    <param name="vm"/>
    <body>
      <body1 xsi:type="elaScript:Action" res="//@script/@param.0">
        <scalefunction name="sui"/>
      </body1>
    </body>
  </script>
</elaScript:ElaScript>
```

Code 7.5 – Fichier *.xmi* généré correspondant au fichier *.elas* de la Figure 7.7.

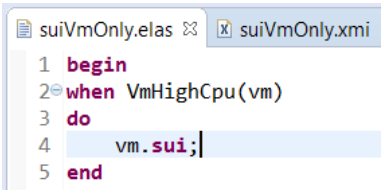


FIGURE 7.7 – Définition d'une *tactique* et génération du fichier *.xmi* correspondant.

— **Parcours du modèle *XML*** : Il s'agit d'analyser le contenu des différents fichiers *.xmi* correspondant au *tactiques éligibles* en vue de les confronter au contexte d'exécution. Cette étape est réalisée à l'aide d'un analyseur (i.e. *parser*) chargé de parcourir l'arborescence *XML*. Nous avons développé un *parser Java* reposant sur *SAX* (*Simple API for XML* [sax16]) et *JDOM* (*Java Document Object Model* [jdo16]). *SAX* est une interface de programmation pour de nombreux langages permettant de lire et de traiter des documents *XML*. *JDOM* est une *API* propre au langage *Java* qui utilise des collections *SAX* pour *parser* les fichiers *XML* et manipuler les éléments d'un *DOM* (*Document Object Model* [dom16]).
Notre *parser*, implémenté par la classe *TacticParser*, fonctionne de la manière suivante. Chaque *invocation de fonction* (i.e. couple $variable.fonction$) va faire l'objet d'un contrôle spécifique relatif au type de la ressource (i.e. portée par le *symptôme*) et de l'action d'adaptation concernée (i.e. la valeur de l'attribut $name$ de l'élément $scalefunction$). En ce sens, chaque couple $resourceType.actionType$ possible va se voir attribuer une méthode chargée de son contrôle (i.e. *micro-contrôle* par instruction). Ces méthodes sont implémentées dans la classe *Java* $Checker$ qui va être chargée d'assurer les différents contrôles de notre *Context Filter*.

Micro-contrôle (par instruction) : prenons l'exemple de l'unique *invocation de fonction* du *script* défini dans la Figure 7.7. Celui-ci concerne une ressource de type Vm et une action d'adaptation de type sui. On va ainsi contrôler la faisabilité de cette instruction en appelant la méthode *checkSuiForVm(Vm vm)* définie dans la classe $Checker$. Cette méthode, retournant un *booléen*, va être chargée de contrôler si l'action sui peut être exécutée ou non sur l'objet de type Vm remonté par le *symptôme* $VmHighCpu$. Dans ce cas, on va par exemple vérifier si la *vm* n'est pas déjà paramétrée avec l'Off_{infra} maximum.

Contrôle étendu (parcours du graphe) : le contrôle réalisé par les méthodes de la classe $Checker$ ne s'arrête pas nécessairement aux contraintes des ressources portées par les *symptômes*. En effet, il est possible de considérer des contraintes liées à la hiérarchie de ressources en bénéficiant notamment de la structure de graphe (e.g. navigation vers les nœuds parents, fils, etc.). Ainsi, dans le cas de la méthode *checkSuiForVm(Vm vm)*, on va par exemple pouvoir contrôler si la *PM* hébergeant la *VM* (i.e. nœud parent) a suffisamment de ressources disponibles pour appliquer l'action sui (i.e. contrôle en cascade).

Macro-contrôle (par *tactique*) : la classe $Checker$ va contenir des méthodes chargées du contrôle des contraintes globales (i.e. *macro-contrôle*), pouvant s'appliquer à toute la *tactique*, comme par exemple une limite budgétaire à ne pas dépasser. Ce type de contrôle peut être associé à une *post-condition*. Ainsi, après avoir contrôler unitairement la faisabilité des différentes instructions durant l'analyse de la *tactique* (i.e. *parsing* réalisé par le *TacticParser*), on s'assure que l'exécution de la *tactique* elle-même ne rentre pas en conflit avec certaines contraintes globales.
En résumé, chaque *tactique éligible* est analysée par notre *TacticParser* qui procède au contrôle de chaque instruction qui la constitue ainsi qu'à la *tactique* dans sa globalité (i.e. en faisant appel aux méthodes concernées dans la classe $Checker$). Si tous les contrôles effectués sont positifs (i.e. toutes les méthodes ont retournées $true$), alors la *tactique* est considérée comme *applicable*.

7.3.4 Exemple

La Figure 7.8 illustre le *Context Filter* en considérant 2 contraintes évaluées à l'exécution. Ces deux contraintes portent sur l'état courant des ressources. La première indique que toutes les VMs du tier (remonté par le *symptôme* de type *TierOverloaded(Tier tier)*) sont déjà paramétrées avec l'Off_{infra} maximum (e.g. *Large instance*) tandis que la seconde précise qu'aucun composant supplémentaire ne peut être débranché c'est-à-dire que les VMs du *tier* admettent déjà la configuration architecturale minimale. La première contrainte va avoir comme impact de rendre la *tactique* SUI (cf. Figure 7.9) "non applicable" du fait qu'il n'est pas possible d'augmenter davantage l'Off_{infra} des *vms* sous-jacentes. La seconde contrainte va quant à elle rendre la *tactique* SIS "non applicable" du fait qu'aucun composant supplémentaire ne peut être retiré.

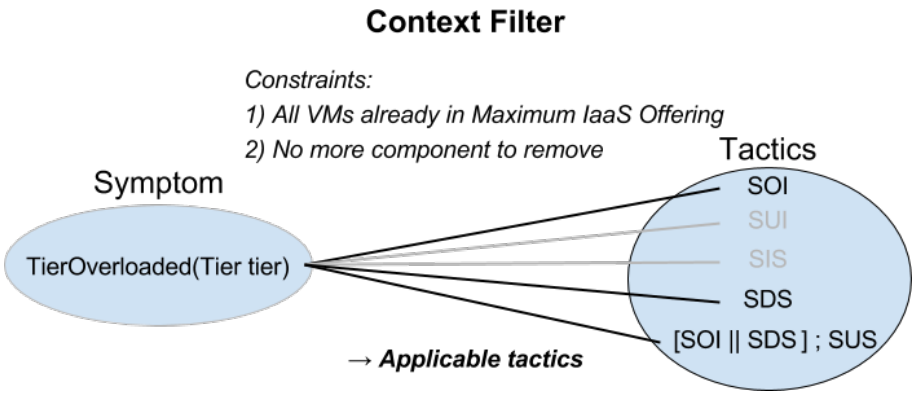


FIGURE 7.8 – Scénario d'exemple : *Context Filter*.

```
1 begin
2 when TierOverloaded(tier)
3 do
4     // sui on tier's vms
5     tier.sui;
6 end
```

FIGURE 7.9 – Tactique *SUI* : fichier *.elas*.

Le Code 7.6 donne un aperçu du contrôle effectué dans la méthode *checkSuiForTier(Tier tier)* de la classe $Checker$. Cette méthode va être appelée lors de l'analyse (*parsing*) du fichier *.xmi* de la *tactique SUI* au moment de l'évaluation de l'instruction *tier.sui* par le *TacticParser*. Il s'agit ici d'un simple contrôle qui vise à identifier si toutes les *vms* du *tier* (i.e. nœuds fils) sont déjà paramétrées avec l'Off_{infra} maximum. On considère que si au moins une *vm* peut passer à l'échelle verticalement, alors le *tier* aussi. Le résultat de ce contrôle va permettre d'identifier la première contrainte évoquée précédemment. La méthode *checkSuiForTier(Tier tier)* va donc retourner *false* à l'objet *TacticParser*, qui va ainsi considérer la *tactique SUI* comme *non-applicable*.

```
// Check whether we can apply the sui action on the tier
public boolean checkSuiForTier(Tier tier) {
  // Foreach vm of the tier
  for (Vm vm : tier.getVms()) {
    // If at least one vm can be scaled up
    if (!((vm.getOffering() + 1) > Vm.MAXOFF))
      // We consider that the tier also
      return true;
  }
  // No vm to scale up
  return false; // Tactic non applicable
}
```

Code 7.6 – Exemple de contrôle effectué sur les nœuds fils d'une ressource.

7.3.5 Bilan

En résumé, le *Context Filter* va évaluer chacune des *tactiques éligibles*, au travers du *Tactic-Parser*. Ce dernier est chargé de parcourir et contrôler le contenu de chaque *script* en rattachant celui-ci au contexte d'exécution. Le résultat de l'analyse des *tactiques* en fonction des contraintes *runtime* va être un ensemble de *tactiques applicables*.

Le résultat du *Context Filter* peut potentiellement retourner aucune *tactique applicable* ce qui laisse présager un système trop contraint ou un panel d'adaptation trop restreint (i.e. *mapping* du *Tactics Filter* limitant). Bien que cela n'ait pas été traité dans ce travail, il est possible d'imaginer de relâcher certaines contraintes (e.g. en leur associant des priorités) jusqu'à obtenir au moins une *tactique applicable*.

Nous avons présenté brièvement notre implémentation du *Context Filter* chargé de contrôler les *contraintes runtime* afin d'identifier les *tactiques applicables*. Une piste d'amélioration concernant l'implémentation du *Context Filter* serait de s'appuyer sur les concepts de la *programmation par contrat* (i.e. *contract programming* [Mey92]). On peut ainsi imaginer une implémentation reposant sur *OCL* (i.e. *Object Constraint Language* [WK98]) pour assurer le respect des contraintes en définissant celles-ci sous-forme d'*invariant*, de *pré-condition* ou encore de *post-condition*.

7.4 Préférences de l'administrateur : *stratégies d'élasticité*

7.4.1 Motivations

Dans le cas où plusieurs *tactiques applicables* sont retournées par le *Context Filter*, il devient nécessaire de départager celles-ci afin d'identifier la "meilleure" *tactique* possible selon des préférences définies par l'administrateur humain. Il s'agit alors de munir l'administrateur de moyens simples et efficaces, lui permettant de transmettre ses intentions au gestionnaire autonomique de manière intelligible sous forme de *stratégies d'élasticité* (cf. sous-section 5.1.4), en vue d'orienter la prise de décision d'adaptation.

7.4.2 Spécifications

Stratégies d'élasticité

Les *stratégies d'élasticité* (ou *stratégies d'adaptation*) peuvent être vues comme des politiques de haut niveau, spécifiées en amont par l'administrateur humain et stockées dans le K, qui vont être évaluées à l'exécution par le *Strategy Filter* et dont le résultat va être de classer les *tactiques applicables* les unes par rapport aux autres en vue d'identifier la *tactique* répondant le mieux aux intentions de l'administrateur.

Les *tactiques* ne sont pas équivalentes et admettent des avantages et des inconvénients selon certains *critères d'adaptation*. Pour rappel, nous considérons 6 critères d'adaptation dans notre travail (cf. sous-section 5.1.4) en fonction desquels les actions d'élasticité (cf. sous-section 5.1.2) peuvent être classées (cf. Tableau 5.2). Les *tactiques* vont ainsi faire l'objet d'une analyse afin de calculer leurs impacts respectifs sur les *critères d'adaptation* en fonction des actions qui les constituent. Une *stratégie d'élasticité* revient à donner un ordre total sur les *critères* (i.e. classement sans ex-aequos) en triant ceux-ci du plus important selon l'administrateur (et donc le plus décisif) au moins important. La sélection est basée sur une comparaison des *critères* suivant un ordre lexicographique des objectifs (i.e. méthode lexicographique [RTL96] [MA04]). Dans le cas de nos *critères*, les objectifs vont être de maximiser la *QoS*, la *QoF*, le *temps de reconfiguration* et la *réactivité* et/ou de minimiser les *coûts* et la *consommation énergétique*.

Score des tactiques pour les critères

Dans le cadre du *framework perCEPtion*, nous avons introduit la notion de *score* pour les *symptômes* afin de prioriser et trier ceux-ci (i.e. *symptoms queue*) en fonction du nombre et du type de ressources impactées (cf. sous-section 6.1.2). De la même manière, nous introduisons la notion de *score* pour les *tactiques*. Néanmoins, contrairement au *score* pour les *symptômes*, il s'agit ici de calculer l'impact d'une *tactique* selon un certain point de vue (i.e. un *critère d'adaptation*).

Chaque *tactique* va faire l'objet d'une analyse à l'exécution afin de calculer son *score* pour chaque *critère* en fonction des actions d'élasticité qu'elle contient (i.e. leur type et leur signature). Le *Strategy Filter* va s'appuyer sur ces *scores* ainsi que la *stratégie d'élasticité* pour comparer les différentes *tactiques* (i.e. méthode lexicographique). Pour rappel, une phase d'analyse (i.e. *parsing* des fichiers *.elas*) est déjà réalisée dans le cadre du *Context Filter* afin de contrôler si une *tactique* est applicable ou non en fonction du contexte d'exécution (i.e. contraintes *runtime*). Comme nous allons le voir, nous allons profiter de ce *parsing* pour calculer les *scores* des *tactiques*.

Le *Strategy Filter* est ainsi chargé de sélectionner la meilleure *tactique*, selon les préférences de l'administrateur (i.e. *stratégie d'élasticité*), parmi les *tactiques applicables* retournées par le *Context Filter*. Cette sélection de la meilleure *tactique* va s'appuyer sur un tri (i.e. *sort*) des *tactiques applicables*. Le tri des *tactiques*, tout comme le tri des *symptômes*, va être réalisé au travers d'une *queue à priorité* (i.e. *tactics queue*). La *tactics queue* va ainsi trier les *tactiques applicables* en s'appuyant sur la *stratégie d'élasticité* définie en amont par l'administrateur (stockée dans le *K*) ainsi que les *scores* de celles-ci pour chaque *critère d'adaptation*.

Fonctionnement général du *Strategy Filter*

La Figure 7.10 donne une illustration conceptuelle du *Strategy Filter* ainsi que son intégration avec le gestionnaire autonomique. On y retrouve les différents concepts évoqués ci-dessus (cf. *scores*, *elasticity strategy*, *tactics queue*, etc.) ainsi que les liens avec l'étape précédente du processus de décision, à savoir le *Context Filter*. Nous allons nous intéresser dans ce qui suit à l'implémentation du *Strategy Filter* correspondant au schéma de la Figure 7.10.

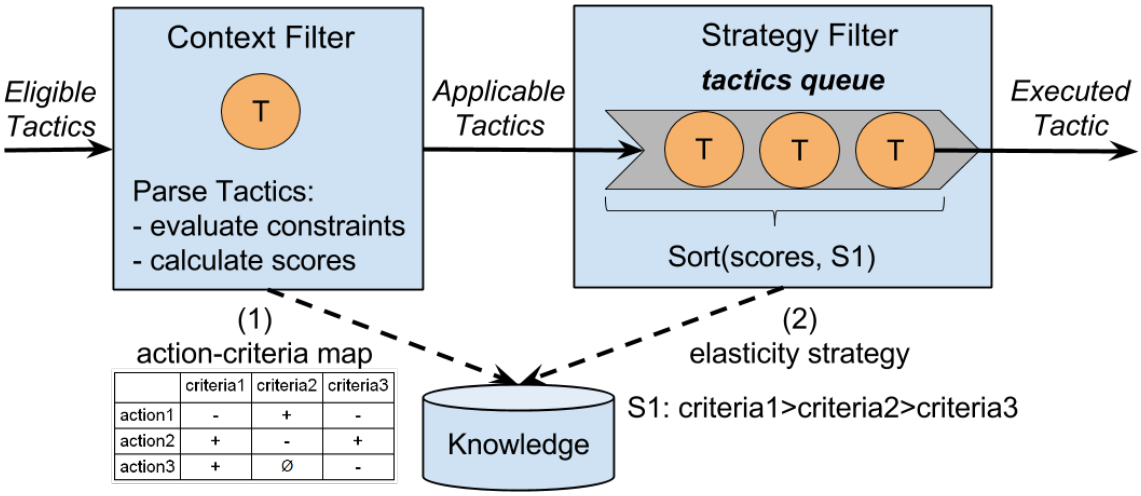


FIGURE 7.10 – *Strategy Filter* : *scores*, *queue de tactiques* et *stratégie*.

7.4.3 Implémentation

Le principe du *Strategy Filter* repose sur les 4 étapes suivantes dont les 2 premières sont à la responsabilité de l'administrateur (i.e. paramétrisation *at design-time*) tandis que les 2 dernières sont à la charge du gestionnaire autonomique (i.e. *runtime*) :

1. *définir le mapping action-critère* nécessaire au calcul des *scores* des *tactiques*. Il s'agit de définir l'impact unitaire des actions de base sur les *critères* concernés (cf. Tableau 5.2) ;
2. *définir une stratégie d'élasticité* qui revient à donner un ordre total sur les *critères d'adaptation*. Les *critères* sont ainsi ordonnés par ordre décroissant d'importance traduisant les préférences d'adaptation de l'administrateur ;
3. *calculer les scores des tactiques applicables* en fonction du contexte d'exécution. Il s'agit ici d'identifier l'impact de chaque *tactique* sur les différents *critères* considérés. Cette étape repose sur le *parsing* des *tactiques* (i.e. fichier *.elas*). Le calcul des *scores* s'appuie sur les actions qui constituent la *tactique* (et leur signature, c'est-à-dire les paramètres), le *mapping action-critère* défini au préalable ainsi que l'état des ressources à l'exécution. De cette phase résulte un *score* pour chaque *critère* de chacune des *tactiques* ;
4. *comparer les tactiques applicables* en fonction de la *stratégie d'élasticité* définie. Il s'agit de comparer les *scores* des *tactiques*, pour chaque *critère*, dans l'ordre d'évaluation imposé par la *stratégie d'élasticité* (i.e. méthode lexicographique). Nous allons détailler dans ce qui suit l'implémentation des différentes étapes évoquées ci-dessus.

Mapping action-critère : tableau des tendances

Le *mapping action-critère* consiste à "quantifier" l'impact des actions de dimensionnement de base (cf. sous-section 5.1.2) sur les différents *critères d'adaptation* (cf. sous-section 5.1.3). Nous avons déjà présenté un tel *mapping* dans le Tableau 5.2. Ce *mapping* offrait une vision des tendances globales des différentes actions selon les *critères* (i.e. en considérant les 4 valeurs suivantes : $+$, $-$, $*$ et $\emptyset$). Idéalement, nous souhaiterions peupler un tel *mapping* de valeurs numériques issues de calculs préalables. Néanmoins, cela n'est que rarement réalisable du fait de la nature même des *critères*, des nombreux paramètres pouvant impacter les résultats ainsi que du très grand nombre de tests de calibrage à réaliser si l'on souhaite être exhaustif en couvrant tous les cas possibles (i.e. problème d'explosion combinatoire).

Un *critère* doit permettre de mesurer les préférences du décideur (i.e. administrateur) vis-à-vis de chaque action, selon un certain point de vue. Bien que l'impact sur certains *critères* peut être valué aisément, par calcul (e.g. le *coût* calculé à partir d'une fonction) ou suite à une phase de tests de calibrage, il est parfois plus difficile de se prononcer numériquement sur d'autres *critères* (e.g. *QoS* ou *énergie* qui font intervenir d'autres paramètres comme la charge de travail).

Dans le cadre de notre travail, nous reposons sur un simple *mapping* comme celui présenté dans le Tableau 5.2. Il s'agit ainsi de considérer les tendances globales des actions d'élasticité sans rentrer dans le détail de leurs impacts précis sur les *critères*. Ainsi, nous proposons une implémentation du *mapping action-critère* sous forme d'une collection *Java Map<Action, Map<Criteria, Integer>* pour laquelle *Action* désigne le type d'action d'élasticité (e.g. SO_{infra} ou SU_{soft}) et *Map<Criteria, Integer>* représente les couples *critère-valeur* pour cette action. Nous nous appuyons ensuite sur ce *mapping* pour le calcul des *scores* des *tactiques*.

La version actuelle du *mapping action-critère* peut être qualifiée de naïve du fait que l'on associe à chaque type d'action (e.g. SO_{infra} ou SU_{soft}) une valeur unique pour un *critère*. En effet, nous donnerons par la suite (cf. sous-section 7.4.5) quelques pistes exploratoires pour définir un *mapping* davantage précis pour lequel les valeurs des différents couples *action-critère* peuvent être représentées sous forme de fonctions (e.g. la fonction qui permet de calculer le *coût* associé à une action SO_{infra} spécifique en fonction des paramètres de l'instruction, de l'état des ressources du système découvert à l'exécution, etc.).

Stratégie d'élasticité

La définition d'une *stratégie d'élasticité* peut être implémentée au travers d'une liste contenant les *critères d'adaptation* considérés. Les éléments de cette liste sont ordonnés en fonction des préférences de l'administrateur. Par la suite, l'algorithme chargé de comparer les *tactiques applicables* va comparer celles-ci en suivant l'ordre des *critères* défini dans cette liste. Le Code 7.7 donne un aperçu de l'implémentation d'une *stratégie d'élasticité* à savoir une classe *Java* héritant du type *java.util.ArrayList<E>*. La collection (cf. *ArrayList<Criteria>*) va correspondre à la liste ordonnée des *critères* traduisant les préférences de l'administrateur. Les *critères* considérés sont représentés au travers d'une énumération *Java* (i.e. *enum*) comme présenté dans le Code 7.8. Enfin, le Code 7.9 donne un exemple de déclaration de *stratégie d'élasticité*.

```
public class Strategy extends ArrayList<Criteria> {
  // ...
  public Strategy(Criteria... criteria) {
    for(Criteria cri:criteria) {
      this.add(cri);
    }
  }
  //...
}
```

Code 7.7 – Portion de code *Java* de la classe *Strategy*.

```
public enum Criteria {
  CRI_QOF("QoF"), CRI_QOS("QoS"), CRI_COST("cost"), CRI_ENERGY("energy"),
      CRI_RECONFTIME("reconftime"), CRI_REACTIVITY("reactivity");
  private final String name;
  private Criteria(final String name) {
    this.name = name;
  }
  public String toString() {
    return name;
  }
}
```

Code 7.8 – *Critères d'adaptation* : énumération *Java*.

```
strategy = new Strategy(Criteria.CRI_COST, Criteria.CRI_QOS, Criteria.CRI_QOF,
    Criteria.CRI_REACTIVITY, Criteria.CRI_ENERGY, Criteria.CRI_RECONFTIME);
```

Code 7.9 – Déclaration d'une *stratégie d'élasticité* avec nos *critères*.

Scores des tactiques

Le calcul des *scores* revient à *parser* les *tactiques applicables*, à savoir les fichiers *.elas*, en tenant compte du contexte d'exécution. Il s'agit de prendre en considération l'état des ressources, et éventuellement leurs liens de parenté, afin de savoir quel sera l'impact de l'application d'une *tactique* (e.g. nombre de VMs impactées dans le cas d'une action SU_{infra} appelé sur un tier). En ce sens, cette phase est proche de ce qui est réalisé dans le *Context Filter*. Pour cette raison, afin de ne pas effectuer deux *parsing* des fichiers *.elas* (ou plus exactement des fichiers *.xmi* générés, cf. Code 7.5), nous avons fait le choix de procéder au calcul des *scores* au même moment que nous contrôlons les contraintes *runtime* lors du *parsing* réalisé dans le *Context Filter*.

Contrairement au calcul du *score* des *symptômes* pour lequel on tient compte du nombre de ressource et de leur type, il s'agit ici de tenir compte des actions qui constituent les *tactiques* en considérant leur type (e.g. SO_{infra} ou SU_{infra}) mais aussi leurs signatures (i.e. méthodes de l'API et les paramètres). En effet, deux appels différents d'une même action peuvent impacter le système de manière distincte (i.e. en termes de *QoS*, de *coût*, de *QoF*, etc.).

Prenons les deux instructions *ElaScript* suivantes : $mytier.soi(1,"medium")$ et $mytier.soi(3,"medium")$ qui correspondent respectivement à ajouter 1 et 3 *Medium instances* au tier $mytier$. Dans ce cas, le *coût* de l'adaptation induit par la seconde instruction va être 3 fois supérieure à celui de la première, d'où la nécessité de prendre ces informations en considération dans le calcul du *score*. De la même manière, un changement d'Off_{infra} appliqué à un tier, à savoir l'exécution d'un SU_{infra} (ou d'un SD_{infra}) sur toutes les VMs de celui-ci, va avoir un impact différent en fonction du nombre de VMs du tier (i.e. nœuds fils) ou encore s'il s'agit d'augmenter l'Off_{infra} de *Small instance* à *Medium instance* ou de *Small instance* à *ExtraLarge instance* (i.e. impact distinct en termes de *coûts*, QoS, etc.).

Dans la même veine que pour le contrôle des contraintes *runtime* effectué par la classe *Checker* du *Context Filter* (cf. Figure 7.6), nous proposons ici une classe *ScoreCalculator* qui va être chargée de calculer, pour chaque $scalefunction$ de la *tactique*, l'impact sur les *scores* des différents *critères*. Ce calcul prend en considération le type d'action (i.e. $scalefunction$), l'état des ressources appelantes mais aussi les différents paramètres de l'action (cf. $ParamScaleFunctionList$ de la Figure 6.11). Le *score* total de la *tactique* va correspondre à la somme des *scores* des $scalefunction$ qui la constituent.

Nous profitons du *parsing* réalisé par le *TacticParser* pour appeler l'objet *ScoreCalculator* (e.g. méthode *Map<Criteria, Integer> calculateSuiForVm(Vm vm, List<Object> scaleParams)*) sur l'action courante après avoir contrôlé la validité de celle-ci via l'appel à l'objet *Checker* (e.g. méthode *boolean checkSuiForVm(Vm vm)*). Un objet de la classe *Tactic* (correspondant à un fichier *.xmi*) a un attribut *scores* de type *Map<Criteria, Integer>*. Les valeurs de cette *Map* sont mises à jour après chaque appel aux méthodes de l'objet *ScoreCalculator* qui retournent elles-aussi des *Map<Criteria, Integer>* (i.e. ajouter de manière itérative les valeurs des *scores* des actions à l'attribut *scores* de la *tactique*).

Identifier la meilleure tactique

La comparaison des *tactiques applicables* est implémentée selon le principe suivant. Une fois l'analyse d'une *tactique* terminée (i.e. contraintes *runtime* contrôlées avec l'objet *Checker* et le *score* calculé par le biais de l'objet *ScoreCalculator*), celle-ci est ajoutée à une queue de *tactiques* nommée *tactics queue* (cf. Figure 7.10). Cette queue de *tactiques applicables*, tout comme la *queue de symptômes*, est implémentée à l'aide d'une *queue à priorité* (i.e. *java.util.PriorityQueue<Tactic>*).

Les *tactiques* contenues dans cette queue vont être triées en fonction de la *stratégie d'élasticité* définie par l'administrateur (i.e. les *scores* respectifs pour chaque *critère*). La tête de la *tactics queue* va ainsi contenir la *tactique* ayant la priorité maximum c'est-à-dire la *tactique* qui sera exécutée du fait qu'il s'agit de la meilleure selon les préférences de l'administrateur.

Chaque élément ajouté à la *tactics queue* est directement positionné à la bonne place en fonction de sa priorité. La détection des priorités au moment de l'ajout d'un nouvel élément (et le tri qui en résulte) est réalisée en implémentant l'interface *java.lang.Comparable<T>*. Dans notre cas, il s'agit de proposer une implémentation de la méthode *compareTo(Tactic t)* qui vise à comparer 2 *tactiques* entre elles en fonction de leurs *scores* respectifs selon les *critères* d'adaptation (évalués dans l'ordre défini par la *stratégie d'élasticité*). L'implémentation de la méthode *compareTo(Tactic t)* est donnée dans le Code 7.10.

```
// Methode lexicographique pour comparer deux tactiques
// Methode utilisee pour trier la tactics queue
@Override
public int compareTo(Tactic t) {
  Map<Criteria, Integer> tScores = t.getScores();
  int res = 0;
  for (java.util.Map.Entry<Criteria, Integer> entry :
      this.getScores().entrySet()) {
    res = tScores.get(entry.getKey()).compareTo(entry.getValue());
    if (res != 0)
      break;
  }
  return res;
}
```

Code 7.10 – Implémentation de *compareTo(Tactic t)* pour trier la *tactics queue* selon les *scores*.

Dans notre implémentation, le calcul des *scores* des *tactiques* est réalisé au moment du *parsing* du *Context Filter* mais évalué au moment du *Strategy Filter* à travers le tri de la *tactics queue* selon la *stratégie d'élasticité* définie. Bien que la version actuelle considère une unique *stratégie* statique, il est possible d'envisager de changer de *stratégie d'élasticité* à l'exécution (i.e. définir un nouvel ordre total sur les *critères*) ce qui va impacter le tri de la *tactics queue* et potentiellement le choix de la *tactique* qui va être exécutée (i.e. tête de la queue). Cela permettrait par exemple de définir deux *stratégies d'élasticité* pouvant s'appliquer à différents moments de la journée (e.g. privilégier la *réactivité* et la *QoS* en période de pointe et la *QoF* et les *coûts* en période calme).

7.4.4 Exemple

La Figure 7.11 donne un aperçu du *Strategy Filter* et vient achever le scénario d'exemple déroulé tout au long de ce chapitre. On retrouve les 3 *tactiques applicables* (cf. *SOI*, *SDS* et *[SOI||SDS] ;SUS*) remontées par le *Context Filter* face au *symptôme* de type *TierOverloaded(Tier tier)*. La dernière étape du processus de décision (i.e. *Strategy Filter*) doit permettre d'identifier, parmi les 3 *tactiques applicables*, la" meilleure" *tactique* qui sera exécutée, selon les préférences de l'administrateur représentées par une *stratégie d'élasticité*.

On retrouve le *mapping action-critère* (cf. *(1)*), c'est-à-dire les tendances sur lesquelles le *ScoreCalculator* s'appuie pour le calcul des *scores* des différentes actions constituant les *tactiques*. Une fois les différentes *tactiques* analysées, on obtient des *scores* globaux pour chaque *critère* des *tactiques* (cf. *(2)*). Dans cet exemple, nous présentons seulement 3 des 6 *critères d'adaptation* considérés dans notre travail (cf. sous-section 5.1.3) à savoir le *coût*, la *QoF* et la *réactivité*. Pour rappel, les objectifs d'adaptation sont de maximiser la *QoF* et la *réactivité* tout en minimisant les *coûts*.

Nous allons détailler dans un premier temps les *scores* obtenus pour les deux premières *tactiques SOI* et *SDS* (cf. *(2)*) qui sont constituées d'une unique action. Il s'agit, à partir des informations contenues dans le tableau de tendances (cf. *(1)*) de calculer le *scores* de ces deux *tactiques*. Nous considérons dans cet exemple que le *tier* remonté par le *symptôme TierOverloaded* contient une unique VM. De même, nous considérons ici que l'action SO_{infra} contenue dans la *tactique SOI* revient à ajouter une unique VM au *tier* et que l'action SD_{soft} contenue dans *SDS* revient à diminuer l'Off_{soft} de 2 niveaux.

Les *scores* calculés pour la *tactique SOI* sont les suivants : $score_{reactivity} = -1$ et $score_{cost} = -1$ (i.e. une unique VM est démarrée). On remarquera la valeur $\emptyset$ pour le *critère QoF* sur lequel l'action SO_{infra} n'a pas d'impact direct. De même, on retrouve la valeur $\emptyset$ pour la *tactique SDS* dans le cas du *critère cost* du fait de l'unique action SD_{soft} qui la constitue. La *tactique SDS* obtient la valeur $score_{QoF} = 2 \times -1 = -2$ (i.e. diminution de l'Off_{soft} de 2 niveaux). En revanche, dans le cas du *critère reactivity*, la valeur est de $+1$ du fait que estimons ici que la valeur de l'Off_{soft} du SD_{soft} n'a pas d'impact direct sur la *réactivité*.

Pour ce qui est de la *tactique [SOI||SDS] ;SUS*, $score_{QoF} = 0$ du fait des valeurs opposées des actions SD_{soft} et SU_{soft} pour le *critère QoF* (i.e. -1 et $+1$). Le $score_{cost} = -1$ qui correspond au résultat de l'unique action de la *tactique* ayant un impact sur le *critère cost* à savoir SO_{infra}. Enfin, $score_{reactivity} = +1$ du fait que l'on va retourner le $max_{reactivity}(SOI, SDS)$ car les actions SO_{infra} et SD_{soft} sont exécutées en parallèle. Ainsi, on considère que la *tactique [SOI||SDS] ;SUS* admet la même *réactivité* que la première action de dimensionnement qui sera exécutée, à savoir l'action SD_{soft}.

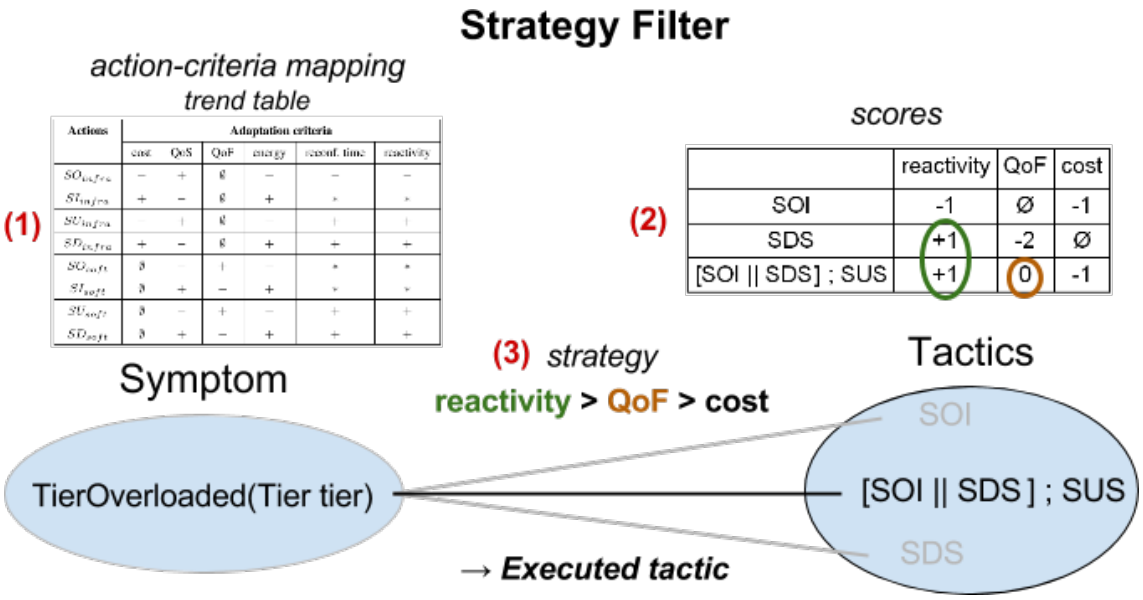


FIGURE 7.11 – Scénario d'exemple : *Strategy Filter*.

On retrouve enfin la *stratégie d'élasticité* définie par l'administrateur qui stipule un ordre total sur les trois *critères* considérés à savoir *réactivité* > *QoF* > *coût* (cf. *(3)*). Cette *stratégie* va impacter le tri de la *tactics queue* (i.e. ordre d'évaluation des *critères* dans la comparaison). En vert sur la figure, on peut voir que le premier *critère réactivité* a permis d'écarter la *tactique SOI* mais n'a pas été suffisant pour départager les deux autres *tactiques*. L'évaluation du second *critère QoF* permet quant à lui d'identifier que la *tactique [SOI||SDS] ;SUS* est préférable à la *tactique SDS*, c'est donc cette première qui sera présentée en tête de la *tactics queue* et donc exécutée en vue d'adapter le *système* face au *symptôme*.

Les *stratégies d'élasticité* offrent un moyen simple et efficace d'orienter la résolution des compromis induits par l'élasticité multi-couche en intégrant les intentions de l'administrateur humain sous formes de politiques de haut niveaux.

7.4.5 Définition d'un calcul de *scores* avancé : pistes exploratoires

Dans notre travail, nous basons notre calcul des *scores* sur un simple *mapping action-critère* pour lequel les types d'actions sont positionnés en fonction de leur impact global (i.e. tendance) sur les *critères*. Cette version, bien que simple à mettre en œuvre, peut être qualifiée de naïve. Pour cette raison, nous avons envisagé d'affiner ce *mapping* et le calcul des *scores* qui en résulte en tentant de représenter quantitativement l'impact des actions sur les *critères*. Cela revient à associer à chaque case du *mapping* (i.e. couple *action-critère*) une fonction permettant de calculer l'impact d'une action sur le *critère*. Le but d'une telle démarche est d'être plus précis quant à la projection de l'exécution d'une *tactique*.

L'objectif est de remplacer les valeurs du Tableau 5.2, à savoir les tendances globales, par des fonctions permettant de calculer une valeur numérique de l'impact sur le *critère*. Pour ce faire, il s'agit de considérer non plus les actions de base (i.e. types d'action) uniquement mais de prendre en considération les instructions (i.e. *ScaleFunction*) contenues dans les *scripts .elas* (i.e. les paramètres portées par les actions : *ParamScaleFunction*) ainsi que le contexte d'exécution (i.e. état des ressources sur lesquelles vont porter les actions). Nous allons donner ci-dessous quelques exemples de fonction pour certains couples *action-critère* en s'appuyant sur le système présenté dans la Figure 7.12 et des instructions de notre langage *ElaScript*.

Critère de coût

1. $SO_{infra} - cost$ et $SI_{infra} - cost$: nous souhaitons ici définir l'impact des actions SO_{infra} et SI_{infra} sur les *coûts*. Plusieurs informations doivent être prises en considération à commencer par le nombre d'instances qui vont être démarrées/arrêtées. De même, le coût des instances varient en fonction de leur Off_{infra} (e.g. une *Small instance* est moins coûteuse qu'une *Large instance*). Ces deux informations peuvent être récupérées dans l'appel de la fonction elle-même (i.e. paramètres) et/ou par introspection de l'état des nœuds du graphe de ressources. Ainsi, la fonction de *coût* pour les actions SO_{infra} et SI_{infra} peut être définie comme suit :

   $cost_{SO_{infra}} = n \times p(o)$ avec n le nombre d'instances ajoutées/retirées, o l'Off_{infra} souhaité et p le coût associé au type d'instance o (e.g. le coût d'une *Small instance*).

   Nous considérons ici que le coût p associé à chaque type d'instance est connu à l'avance et stocké dans le *K* du gestionnaire autonomique. De même, nous considérons que les instances ajoutées (ou retirées) dans une même action SO_{infra} (ou SI_{infra}) sont homogènes (i.e. ayant la même Off_{infra}).
   Les valeurs de n et de o peuvent être récupérées dans l'appel de la fonction sous forme de paramètres (e.g. $tier1.soi(3, "medium")$ avec $n = 3$, $o = medium$ et p correspondant au coût d'une VM de type o) ou identifiées lors de l'introspection du graphe de ressources (e.g. $appli.soi(2, "small")$ qui revient à ajouter 2 VMs de type $small$ à tous les nœuds fils de $appli$ à savoir les tiers $tier1$ et $tier2$ soit $n = 2 \times 2 = 4$). De la même manière que pour l'action SO_{infra}, on peut définir la fonction $cost_{SI_{infra}} = -(n \times p(o))$.

   Le résultat des fonctions ci-dessus donne l'impact numérique sur le *critère* de *coût* à savoir une valeur positive pour le SO_{infra} (i.e. augmentation des *coûts* synonyme d'impact négatif sur le *critère*) et une valeur négative pour le SI_{infra} (i.e. diminution des *coûts* synonyme d'impact positif).

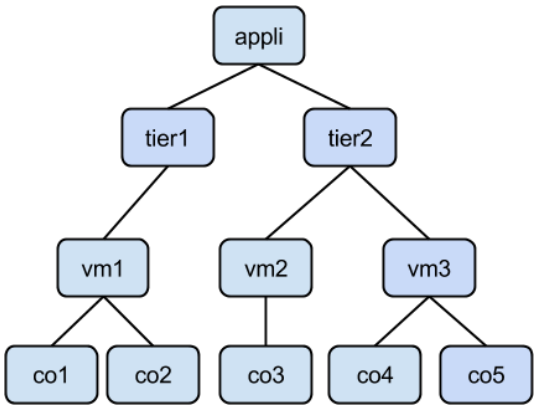


FIGURE 7.12 – Exemple de graphe de ressources.

2. $SU_{infra} - cost$ et $SD_{infra} - cost$: nous nous intéressons là encore au *critère* de *coût* mais cette fois en ce qui concerne les actions de dimensionnement vertical de l'infrastructure. Il s'agit donc de tenir compte des informations telles que le nombre d'instances dont on va changer l'Off_{infra}, leur Off_{infra} courante et l'Off_{infra} cible. En considérant des VMs homogènes (i.e. même Off_{infra}), la fonction de *coût* pour les actions SU_{infra} et SD_{infra} peut être définie comme suit :

   $cost_{SU_{infra}} = cost_{SD_{infra}} = n \times (p(o_{source}) - p(o_{target}))$ avec n le nombre d'instances concernées, o_{source} leur Off_{infra} initiale, o_{target} l'Off_{infra} cible et p le coût associé aux Off_{infra}.

   Les fonctions SU_{infra} et SD_{infra} décrites ci-dessus vont respectivement retourner une valeur positive et négative. En effet, dans le cas du SU_{infra}, o_{target} est une Off_{infra} supérieure à o_{source}, donc plus coûteuse, et à l'inverse le SD_{infra} implique une diminution de l'Off_{infra} (i.e. $o_{target} < o_{source}$) et donc une réduction des coûts. De même que pour les actions SO_{infra} et SI_{infra}, les informations peuvent être récupérées dans les paramètres de la fonction (e.g. o_{target}), par introspection de l'état des nœuds du graphe de ressources (e.g. n, o_{source}) ou dans le *K* (i.e. p le coût associé aux Off_{infra}).

   Prenons les deux instructions *ElaScript* suivantes : $vm1.sdi(2)$ et $tier2.sui$ qui indiquent respectivement que l'on souhaite réduire l'Off_{infra} de la $vm1$ de 2 niveaux (par rapport à son Off_{infra} courante) et que l'on souhaite augmenter l'Off_{infra} de toutes les VMs du $tier2$ de 1 niveau. Dans le cas de la première instruction, $n = 1$ du fait que l'action est appelée sur une unique VM, l'information o_{source} est récupérée par introspection (i.e. $vm1.getOffering()$), puis o_{cible} en est déduite (i.e. $o_{cible} = o_{source} - 2$). Pour la seconde instruction portant sur une ressource de type Tier et dont l'appel à SU_{infra} ne contient aucun paramètre, toutes les informations sont récupérées par introspection du graphe de ressources. Par exemple, on déduit que $n = 2$ à savoir le nombre de nœuds fils du nœud $tier2$ (i.e. $tier2.getVms().size()$).

Critère de temps de reconfiguration

3. $action - reconfTime$ et $tactic - reconfTime$: nous nous intéressons cette fois au *critère* de *temps de reconfiguration* qui désigne le temps nécessaire à la mise en place de l'adaptation. Chaque action admet un *temps de reconfiguration*, identifié par calibrage et stocké dans le *K*. Le *temps de reconfiguration* est dépendant de l'architecture de l'application. Par exemple, le démarrage d'une nouvelle VM et les services associés (i.e. action SO_{infra}) peut varier en fonction du tier concerné (e.g. métier, base de données, etc.).
   Le calcul du *temps de reconfiguration* d'une *tactique* doit considérer les *temps de reconfiguration* des actions sous-jacentes. Il s'agit globalement de faire la somme des *temps de reconfiguration* des actions qui la constituent. Néanmoins, le langage *ElaScript* offre la possibilité de définir des blocs d'instructions s'exécutant en parallèle (e.g. $[bloc1||bloc2]$) dont il faut tenir compte dans le calcul du *temps de reconfiguration* de la *tactique*.

   La Figure 7.13 donne deux exemples de *tactiques* sous forme de processus (i.e. formalisme *BPMN 2.0* [bpm16]) dans le but d'illustrer les calculs des *critères temps de reconfiguration* et *réactivité* des *tactiques*. La première *tactique T1* (cf. Figure 7.13a) correspond à 2 actions (cf. *action1* et *action2*) exécutées en séquence (cf. opérateur de *séquence ElaScript ";"*). Ainsi, le *temps de reconfiguration* de cette *tactique* va naturellement correspondre à $reconfTime_{T1} = reconfTime_{action1} + reconfTime_{action2}$.

La *tactique T2* (cf. Figure 7.13b) est quant à elle constituée de 3 actions (cf. *action3*, *action4* et *action5*) dont les deux premières sont exécutées en parallèle (cf. opérateurs de *parallélisme* et *synchronisation ElaScript "[", "||"* et *"]"*). Pour rappel, la sémantique de l'opérateur de *synchronisation* est que les blocs d'instructions sous-jacents (cf. *action3* et *action4*) doivent avoir terminés respectivement leur exécution avant de poursuivre. Ainsi, le *temps de reconfiguration* considéré dans le cas de blocs d'instructions parallèles correspond à la valeur du bloc le plus long (i.e. $max(reconfTime_{blocs})$). De ce fait, le *temps de reconfiguration* de la *tactique T2* peut être représenté comme suit : $reconfTime_{T2} = max(reconfTime_{action3}, reconfTime_{action4}) + reconfTime_{action5}$.

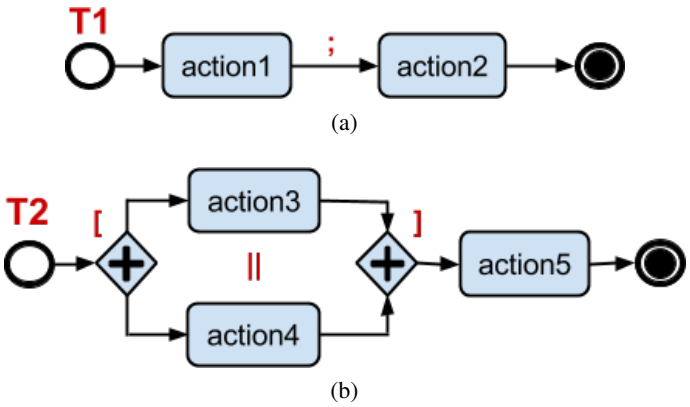


FIGURE 7.13 – Exemple de processus de 2 *tactiques* : formalisme *BPMN 2.0*.

Le type de raisonnement évoqué ci-dessus a été abordé dans certains travaux traitant de la composition de services web. [ZBN+04] s'intéresse notamment à l'*execution duration* induite par la composition de services web qui peut faire intervenir des tâches s'exécutant de manière séquentielle ou parallèle. En suivant leur approche, on pourrait représenter une *tactique* sous forme de graphe où les nœuds représenteraient les actions auxquelles seraient associées une *durée* (i.e. *temps de reconfiguration*) et calculer à partir de ce graphe le *temps de reconfiguration* de la *tactique* en s'appuyant sur une méthode de chemin critique (i.e. *Critical Path Algorithm* [Pin12]).

Nous verrons dans la section suivante que le *critère temps de reconfiguration* peut jouer un rôle important vis-à-vis de la capacité de réaction du gestionnaire autonomique si celui-ci se limite à un unique cycle d'adaptation (i.e. une reconfiguration à la fois).

Critère de réactivité

4. $action - reactivity$ et $tactic - reactivity$: le *critère réactivité* se distingue de *temps de reconfiguration* du fait qu'il s'agit ici de la rapidité de mise en place de l'adaptation c'est-à-dire la capacité de la *tactique* à réagir rapidement au problème porté par le *symptôme*. Pour être plus clair, il s'agit de considérer la *réactivité* de la première *action* contenue dans une *tactique ElaScript*.

   Reprenons l'exemple fourni dans la Figure 7.13 pour illustrer le calcul du *critère réactivité* ainsi que pour mettre en lumière la distinction entre *temps de reconfiguration* et *réactivité*. Dans le cas de la *tactique T1*, la *réactivité* va naturellement être égale à celle de *action1* à savoir $reactivity_{T1} = reactivity_{action1}$. En revanche, dans le cas de la *tactique T2* débutant par deux blocs d'instructions parallèles, il s'agit de déterminer parmi les actions de ces deux blocs (cf. *action3* et *action4*), laquelle admet la meilleure *réactivité*. Ainsi, la *réactivité* de *T2* devient $reactivity_{T2} = max(reactivity_{action3}, reactivity_{action4})$.

Critère de QoS

5. $action - QoS$ et $tactic - QoS$: le calcul de l'impact de l'adaptation sur la *QoS* est davantage complexe du fait que ce *critère* est corrélé à de nombreux paramètres du système (e.g. charge de travail, degré de sollicitation actuel des ressources, etc.). Une solution possible est de s'appuyer sur un modèle de performances issu du résultat de tests de calibrage réalisés en amont et stocké dans le *K*.

   Prenons l'exemple des *temps de réponse* observés pour un *tier*. Les tests de calibrage peuvent permettre d'identifier la relation entre une configuration du *tier* (e.g. la quantité *cpu*, *ram* et la Off_{soft}) et les *temps de réponses* obtenus. On peut ainsi imaginer une fonction $perf_{tier}(cpu, ram, Off_{soft})$ qui va retourner les *temps de réponse* estimés pour une certaine configuration du *tier* (i.e. quantité *cpu* et *ram*). Il s'agit alors de comparer les résultats de la configuration source (i.e. initiale, $conf_{init}$) et de la configuration cible (i.e. projection après adaptation, $conf_{target}$) en faisant la différence des résultats obtenus (i.e. delta entre les deux configurations qui va traduire l'impact de l'adaptation, $perf_{tier}(conf_{init}) - perf_{tier}(conf_{target})$). Le résultat de cette comparaison peut ensuite être traduit en *score* en définissant des intervalles sur les valeurs obtenues (i.e. segmentation). De même, il est possible de dériver l'impact sur l'application elle-même en considérant la somme des impacts sur les tiers qui la constitue $\sum_{n=1}^{R} = perf_n(cpu, ram, Off_{soft})$.

Bilan et discussion

Nous avons évoqué ci-dessus des pistes exploratoires en vue de rendre le calcul des *scores* davantage précis en s'appuyant sur des fonctions prenant en considération l'état des ressources, les paramètres des instructions contenues dans les *scripts*, etc. Bien que cette approche semble bénéfique par rapport à la version naïve qui tend à s'appuyer seulement sur les tendances globales des actions sur les *critères*, cela peut paradoxalement rendre notre concept de *stratégie d'élasticité* obsolète. En effet, le fait de calculer précisément l'impact des différentes *actions* sur les *critères d'adaptation* va entrainer une discrimination importante des *tactiques* selon chaque *critère*. De ce fait, le premier *critère* va souvent permettre d'identifier directement une meilleure *tactique* sans considérer les autres *critères* ce qui peut s'avérer parfois discutable.

Prenons par exemple la *stratégie* suivante $S1 = c1 > c2$ qui concerne 2 *critères* $c1$ et $c2$ (que l'on souhaite maximiser) ainsi que les 2 *tactiques T1* et *T2* vues précédemment (cf. Figure 7.13). Imaginons que *T1* surpasse de justesse *T2* pour le *critère* $c1$ (i.e. $c1(T1) \geq c1(T2) \implies T1ST2$) mais que *T2* surpasse très largement *T1* pour le *critère* $c2$ (i.e. $c2(T2) > c2(T1) \iff T2PT1$). Dans ce cas, *T1* sera directement privilégiée alors que $T2$ semble elle-aussi pertinente (voir davantage). En ce sens, on perd un peu l'intérêt de la décision multi-critère. L'ordre total de préférences sur les *critères* (indiquant l'ordre d'évaluation, ici $c1 > c2$) et plus généralement la méthode *lexicographique* (cf. sous-section 5.1.4) atteignent ici leurs limites.

Une possibilité serait de considérer non plus des *vrai-critères* mais de faire appel à des *quasi-critères* ou des *pseudo-critères* [Bou90]. Pour rappel, un *vrai-critère* induit un pré-ordre total sur les actions. Contrairement à un *vrai-critère*, un *quasi-critère* permet de définir un seuil d'indifférence pour lequel aucune des actions ne surpasse l'autre (noté aIb). De la même manière, un *pseudo-critère* permet de définir des seuils de préférences permettant d'indiquer qu'une action surpasse légèrement une autre sans pour autant constituer une préférence stricte (i.e. aPb). En définissant de tels seuils, il devient possible d'indiquer le degré de précision/confiance que l'on attribue aux *scores* de chaque *critère d'adaptation* tout en gardant l'intérêt des *stratégies d'élasticité*.

7.5 Vers la parallélisation de l'analyse

Dans cette section, nous allons mener une réflexion concernant une extension possible de notre modèle de décision, qui consiste actuellement à analyser et traiter les *symptômes* remontés par la phase de *surveillance* un à un. Nous allons nous intéresser à la possibilité de paralléliser plusieurs processus de décision ce qui revient à traiter plusieurs *symptômes* en parallèle. L'objectif de cette parallélisation est de rendre le système davantage réactif au contexte d'exécution dynamique dans lequel il évolue, en permettant notamment de mener plusieurs adaptations en parallèle.

7.5.1 Motivations

Dans un environnement fortement dynamique, de nombreux *symptômes* peuvent survenir à l'exécution, empêchant le gestionnaire autonomique de tous les traiter. Concrètement, cela va se traduire dans notre *framework autonomique* par le fait que la *symptoms queue* va contenir de nombreux éléments (i.e. générés et poussés par le *M-cycle*) qui finiront par être purgés de celle-ci (i.e. *péremption* atteinte) sans avoir été traités (i.e. *APE-cycle*).

Pour rappel, la *symptoms queue* est triée en fonction des *scores* associés aux *symptômes* (i.e. *priority queue*), assurant ainsi de traiter les *symptômes* prioritaires en premier (cf. sous-section 6.1.2). Cependant, dans un environnement hautement dynamique, certains *symptômes*, bien que importants, ne seront jamais traités du fait que l'adaptation est limitée à un unique *APE-cycle* (i.e. un seul *thread*). Dans cette situation, on voit véritablement apparaître la pertinence du *critère d'adaptation temps de reconfiguration* (cf. sous-section 5.1.4) qui s'avère primordial si l'on souhaite par exemple privilégier, au travers d'une *stratégie d'élasticité*, des *tactiques* ayant des *temps de reconfiguration* courts pour ne pas bloquer l'adaptation du système trop longtemps (*mono-thread*).

L'accumulation de *symptômes* non traités est problématique. En effet, ces *symptômes* peuvent évoluer en des *symptômes* plus graves qui vont par exemple affecter davantage de ressources du système (i.e. nœuds du graphe) ou encore remonter dans la hiérarchie de ressources (e.g. d'un tier vers l'application). Le fait de ne pas traiter les *symptômes* le moment venu peut ainsi dégénérer en de nouveaux *symptômes* plus impactant pour le système qui vont nécessiter un effort d'adaptation supérieur. Cela va avoir un effet boule neige du fait qu'un effort d'adaptation plus important va généralement se traduire par des reconfigurations plus longues (i.e. *APE-cycle*), ce qui va augmenter la latence avant de pouvoir traiter d'autres *symptômes*, qui vont eux-même finir par dégénérer, etc.

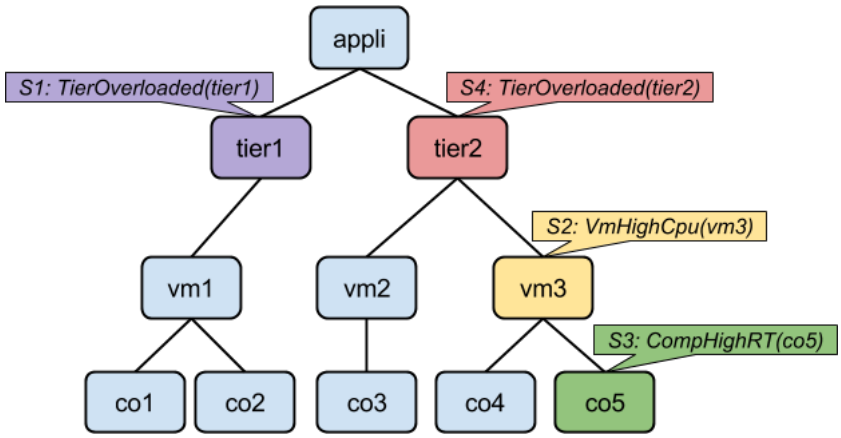
Afin de pallier aux risques évoqués ci-dessus, il serait intéressant de pouvoir traiter plusieurs *symptômes* en parallèle. Il s'agit alors d'entreprendre plusieurs adaptations en parallèle, à savoir de multiples *APE-cycle* (i.e. plusieurs *threads*). Cela implique de considérer les ressources concernées par ces différentes reconfigurations (i.e. sous-graphes) afin de ne pas exécuter des adaptations incompatibles.

7.5.2 Illustration

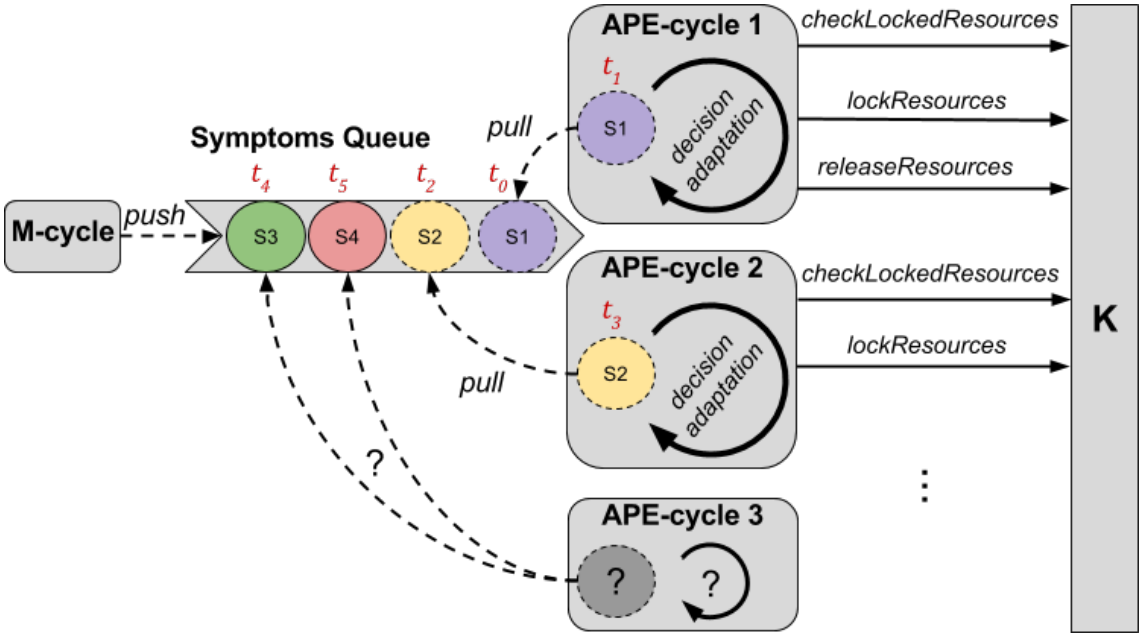
Nous allons présenter ci-dessous un scénario de motivation afin d'illustrer l'intérêt de paralléliser le traitement des *symptômes* ainsi que pour identifier les problèmes soulevés par la mise en place d'une adaptation parallèle. Le cas d'usage est illustré dans la Figure 7.14.

La Figure 7.14a donne une représentation schématique de notre système à l'exécution. Il s'agit ici d'un graphe de ressources, ou plus précisément d'un arbre, du fait que les ressources de type *PM* ont été omises afin de simplifier l'exemple. La figure laisse apparaître 4 *symptômes S1, S2, S3 et S4* portant sur 4 ressources distinctes et générés à l'exécution dans cet ordre par le *M-cycle*.

La Figure 7.14b, associée à l'instance du modèle de ressources, donne une représentation schématique des interactions entre le *M-cycle*, la *symptoms queue* et le(s) *APE-cycle(s)*. On retrouve les *symptômes S1, S2, S3 et S4* dans la *symptoms queue*.



(a) Ressources et *symptômes*.



(b) *M-cycle*, *symptoms queue* et *APE-cycles*.

FIGURE 7.14 – Parallélisation des adaptations : scénario d'exemple.

Nous allons présenter ci-dessous notre scénario d'un point de vue temporel (cf. t_i en rouge sur la figure) puis détaillerons les différentes étapes par la suite :

- t_0 : *symptôme S1* généré par le *M-cycle* et poussé dans la *symptoms queue* ;
- t_1 : *S1* défilé puis traité par l'*APE-cycle 1* ;
- t_2 : *S2* poussé dans la *symptoms queue* ;
- t_3 : *S2* défilé puis traité par l'*APE-cycle 2* ;
- t_4 : *S3* poussé dans la *symptoms queue* ;
- t_5 : *S4* poussé dans la *symptoms queue* et positionné devant *S3* (i.e. *sort*), qui n'a pas encore été défilé et qui admet un *score* inférieur (i.e. *S3* concerne une unique ressource de type *Component* tandis que *S4* concerne un *Tier*).

Les premières étapes t_0 et t_1 reviennent simplement à défiler un premier *symptôme S1* et à déclencher un nouveau cycle d'adaptation *APE-cycle 1*. Ce cycle va prendre la forme d'un *thread* chargé de mener à bien la prise de décision (i.e. *filtres*) puis la reconfiguration du système face au problème remonté par *S1*. Considérons désormais les étapes t_2 et t_3 concernant le *symptôme S2*. Une fois *S1* défilé, *S2* se retrouve en tête de la *symptoms queue*. Se pose alors la question de savoir s'il est possible de traiter *S2* sans interférer avec l'exécution du *APE-cycle 1*.

Lors du traitement d'un *symptôme* par un *APE-cycle*, celui-ci va tout d'abord vérifier si certaines ressources portées par le *symptôme* sont déjà concernées par une autre adaptation (i.e. en considérant le(s) sous-arbre(s) issus des nœuds remontés par le *symptôme*). Ce contrôle est réalisé par un appel au *K* (cf. *checkLockedResources*) dans lequel est stocké l'état des nœuds du système à savoir *libre* (*free*) ou *verrouillé* (*locked*). Si toutes les ressources sont *libres* (i.e. $SymptomResourcesSet \cap LockedResourcesSet = \emptyset$), on considère que l'on peut traiter le *symptôme* dans un nouvel *APE-cycle* sans risque de perturbations avec d'autres traitements en cours (cf. sous-section 7.5.3). Il s'agit alors de *verrouiller* les ressources du *symptôme* (cf. *lockResources*) le temps de l'adaptation jusqu'à la fin du cycle où celles-ci vont être relâchées (cf. *releaseResources*). En revanche, si certains nœuds sont déjà *verrouillés*, on considère que le *symptôme* ne peut pas être traité. Il s'agit alors d'attendre que certaines ressources soient relâchées avant de traiter le *symptôme*. Une manière de procéder est de ré-interroger le *K* après chaque fin d'*APE-cycle* (i.e. ré-évaluer la faisabilité du traitement du *symptôme*). Il est aussi possible que d'autres *symptômes* admettant un *score* supérieur surviennent entre temps. Dans ce cas, ces nouveaux *symptômes* vont se retrouver en tête de la *symptoms queue* et être évalués et traités en priorité.

Dans notre exemple, les *symptômes S1* et *S2* concernent respectivement les ressources *tier1* et *vm3* (cf. Figure 7.14a). Notons $A1$ et $A2$ les arbres issus de ces deux nœuds et composés respectivement des ressources $\{tier1, vm1, co1, co2\}$ et $\{tier2, vm2, vm3, co3, co4, co5\}$. *S1* est déjà en cours de traitement par l'*APE-cycle 1* ce qui implique que les ressources de $A1$ sont considérées comme *verrouillées*. Dans ce cas, les ensembles $A1$ et $A2$ sont disjoints (i.e. $A1 \cap A2 = \emptyset$). Cela laisse présager que les *symptômes S1* et *S2* vont engendrer des adaptations (i.e. *tactiques*) non conflictuelles (i.e. ne faisant pas intervenir de ressources communes). En ce sens, on estime que l'on peut traiter *S2*, c'est-à-dire créer un nouvel *APE-cycle* (cf. *APE-cycle 2*), sans attendre la fin du traitement du *APE-cycle 1*. Ainsi, ces deux cycles vont pouvoir s'exécuter en parallèle sans interférence. Considérons désormais les étapes t_4 et t_5 correspondant à la génération des *symptômes S3* et *S4*. Ces deux *symptômes* concernent respectivement les nœuds *co5* et *tier2* du système. Au moment de défiler le symptôme *S3*, le contrôle réalisé va identifier que *co5* a été verrouillé par l'*APE-cycle 2* (i.e. *co5* est un nœud fils de *vm3*) empêchant ainsi de traiter *S3* en parallèle de *S2*. Le *symptôme S4*, généré entre temps, va se retrouver à son tour en tête de la *symptoms queue*, du fait qu'il admet un *score* supérieur, déclenchant un nouveau contrôle. Celui-ci va pouvoir être traité si l'*APE-cycle 2* a terminé son exécution (i.e. nœuds libérés), sinon, il va falloir attendre à nouveau le relâchement de certaines ressources.

7.5.3 Pistes exploratoires

Nous allons présenter ici le fruit de nos réflexions concernant la mise en place d'une adaptation parallèle pour notre *framework* autonomique de gestion de l'élasticité multi-couche. Il ne s'agit pas de spécifications à proprement parlé mais plutôt de pistes exploratoires pour la mise en place d'une telle gestion parallèle des traitements.

Contrôle orienté *symptômes*

Le contrôle évoqué dans l'exemple ci-dessus, visant à savoir s'il est possible ou non d'exécuter des adaptations parallèles, considère uniquement les ressources portées par les *symptômes* (i.e. les sous-arbres respectifs). Il s'agit de verrouiller toutes les ressources pouvant potentiellement faire l'objet d'une reconfiguration dans les *tactiques* (i.e. tous les nœuds des sous-arbres).

La Figure 7.15 illustre le contrôle orienté *symptômes* au sein du gestionnaire autonomique sous forme d'un diagramme de séquence *UML*. On notera tout d'abord le nouvel élément *PreAnalyze* chargé d'orchestrer le contrôle des *symptômes* et l'exécution des multiples *APE-cycles*. De plus, on remarquera l'apparition de la méthode *peek()* appelée sur la *SymptomQueue*. Contrairement à la méthode *pull()* qui retourne et supprime la tête de la queue, la méthode *peek()* va simplement retourner l'élément en tête de la queue. Cela permet d'amorcer le contrôle basé sur les ressources portées par le *symptôme* sans retirer celui-ci de la queue.

On retrouve les interactions avec le *Knowledge* concernant l'état des nœuds du système (i.e. libre ou verrouillé). Il s'agit d'interroger (cf. *checkLockedResources()*), de verrouiller (cf. *lockResources()*) ou encore de libérer (cf. *releaseResources()*) les nœuds potentiellement impactés par les reconfigurations dans les *tactiques* associées aux *symptômes*. Si les ressources du *symptôme* en tête de la queue (i.e. potentiellement atteignables dans les *scripts* correspondants) n'interfèrent pas avec les ressources actuellement verrouillées par d'autres adaptations (cf. $symp.res \cap lockedResources = \emptyset$), on déclenche un nouvel *APE-cycle* pour traiter celui-ci.

Dans le cas où les ressources concernées par le *symptôme* peuvent interférer avec une adaptation en cours (i.e. *APE-cycle*), 3 solutions sont indiquées. La première (cf. *solution1*) consiste à attendre la fin d'un *APE-cycle*, c'est-à-dire que certaines ressources soient libérées, avant de ré-évaluer s'il est possible de traiter le *symptôme*. La seconde solution (cf. *solution2*) correspond au fait de récupérer un nouveau *symptôme* en tête de la queue (i.e. si un *symptôme* avec une priorité supérieure a été généré et poussé dans la queue ou si le *symptôme* précédent a été purgé de celle-ci). Enfin, la dernière solution (cf. *solution3*) consiste à considérer le *symptôme* suivant dans la queue. Nous reviendrons sur la *solution3* plus tard dans cette section.

Pour rappel, les *scripts* de reconfiguration des *tactiques* n'ont accès qu'aux ressources remontées par le *symptôme*, et éventuellement leurs nœuds fils au travers des actions d'adaptation (i.e. pas de possibilité de remonter aux nœuds parents). Le contrôle orienté *symptômes* revient à considérer les sous-arbres complets (i.e. issus des nœuds/ressources du *symptôme*) comme verrouillés (i.e. jusqu'aux feuilles de type *Component*), ce qui permet d'anticiper toutes les adaptations potentielles sur les ressources correspondantes.

Le contrôle évoqué ci-dessus a l'avantage de pouvoir être implémenté facilement. Néanmoins, bien que cette approche permettre d'éviter les interférences en anticipant largement (i.e. strictement) les ressources potentielles qui seront considérées dans les *tactiques* sous-jacentes, elle n'est pas idéale pour autant et peut s'avérer naïve et trop limitante (e.g. *symptôme* portant sur le nœud *appli* bloquant toutes les ressources du système). En effet, cette approche peut limiter l'exécution de certaines *tactiques* qui auraient pu résoudre le nouveau *symptôme* sans pour autant interférer avec les adaptations en cours (i.e. qui ne concernent pas les mêmes ressources).

Contrôle orienté *tactiques*

La solution revenant à verrouiller toutes les ressources potentiellement atteignables par les *scripts* associés à un *symptôme*, bien que permettant de se prémunir des interférences, peut être qualifiée de stricte. En effet, cela va se traduire par le fait bloquer certaines adaptations qui pourraient avoir lieues. Plutôt que de considérer toutes les ressources potentiellement atteignables, une solution envisageable est de considérer les ressources véritablement impactées par les *tactiques* en cours d'exécution.

Le contrôle basé sur les ressources impactées, que nous qualifions de contrôle orienté *tactiques* en opposition au contrôle orienté *symptômes*, va considérer les *tactiques éligibles* face au *symptôme* remonté. Une première version naïve pourrait être de verrouiller toutes les ressources concernées par les *tactiques éligibles*. Néanmoins, cette solution va tomber dans les même travers que l'approche orientée *symptômes* où l'on va verrouiller "inutilement" certaines ressources sans pour autant les solliciter dans la future reconfiguration du système. Pour cette raison, nous pensons qu'il est préférable de verrouiller uniquement les ressources qui vont être concernées par la *tactique* qui sera exécutée c'est-à-dire la *tactique* identifiée à l'issue du processus de décision (i.e. *Analyze*).

Du point de vue du contrôle, une implémentation possible peut revenir à ajouter un nouveau filtre au processus de décision chargé de contrôler les ressources verrouillées ou encore de modifier le *Context Filter* (cf. section 7.3) pour filtrer les *tactiques* faisant intervenir des ressources verrouillées. En effet, il s'agit là aussi de considérer des contraintes *runtime*. Dans le cas du *Context Filter* actuel, les contraintes qui vont limiter les choix d'adaptation sont imposées par le système (i.e. *managed system*) à l'exécution ou en amont par l'administrateur humain. En revanche, dans le cas du contrôle réalisé pour la parallélisation des *APE-cycles*, l'adaptation va être contrainte à l'exécution par le gestionnaire autonomique (i.e. *autonomic manager*), c'est-à-dire l'adaptation elle-même.

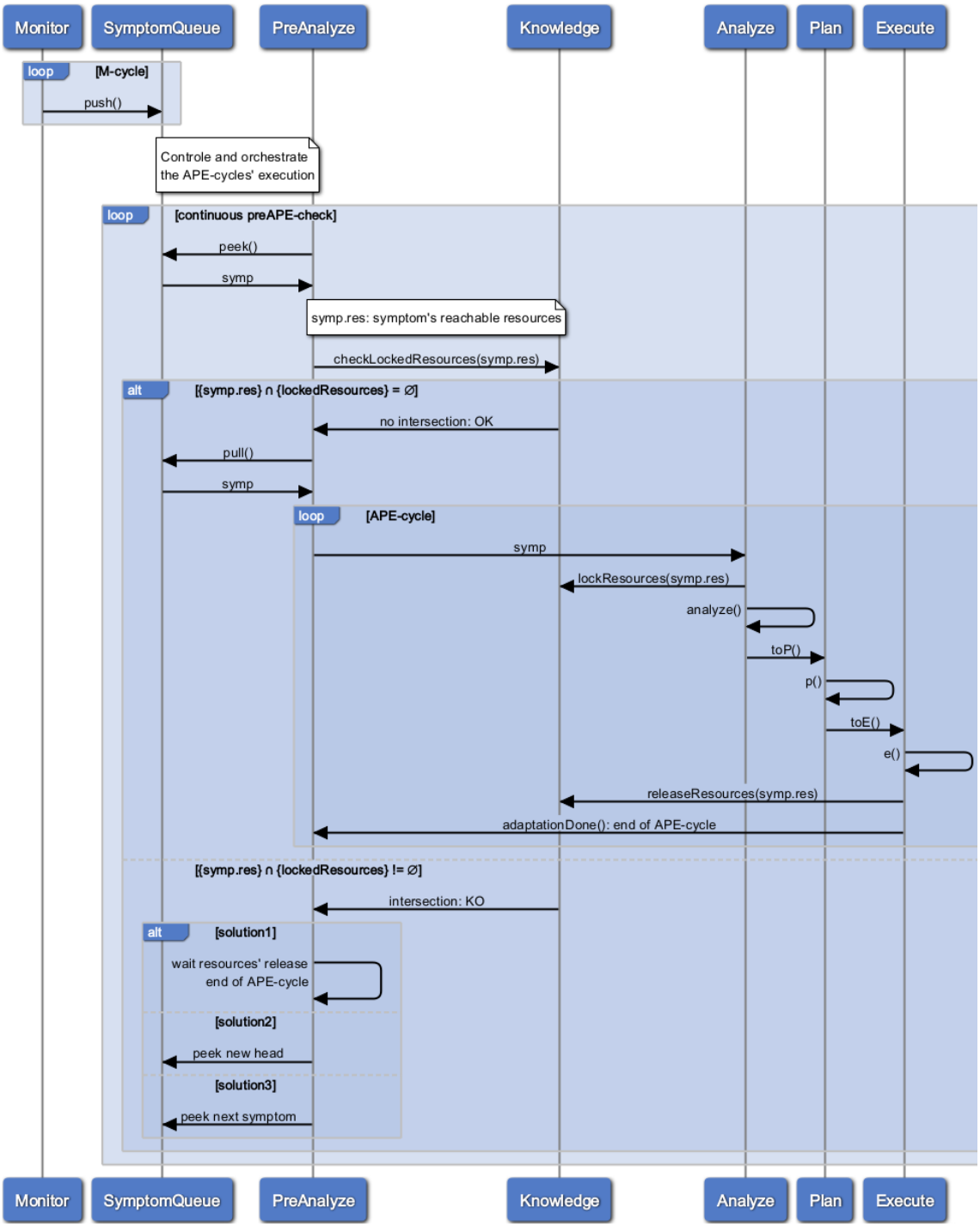


FIGURE 7.15 – Adaptation parallèle : contrôle orienté *symptômes*.

La Figure 7.16 illustre le contrôle orienté *tactiques* au sein du gestionnaire autonomique sous forme d'un diagramme de séquence *UML*. On peut voir que lorsqu'un *symptôme* est récupéré (i.e. *peek()*), un *APE-cycle* est amorcé. Il s'agit alors de démarrer l'*Analyze* en considérant les *ressources* actuellement verrouillées par d'autres *APE-cycles* (cf. *checkLockedResources()*). Si le processus de décision correspondant retourne une *tactique* (cf. *tactic*), on défile réellement le *symptôme* de la queue pour traitement (cf. *pull*) et on verrouille les ressources concernées (cf. *lockResources(tactic.res)*). L'*APE-cycle* poursuit alors son exécution jusqu'à l'adaptation du système et le relâchement des ressources impliquées par la reconfiguration (cf. *releaseResources(tactic.res)*). Dans le cas où aucune *tactique* n'est retournée par le processus de décision, on va avorter l'*APE-cycle* amorcé (i.e. le *symptôme* reste dans la queue) et ainsi se retrouver dans le même cas que pour le contrôle orienté *symptômes* (cf. *solution 1, 2 et 3*).

La solution orientée *tactiques* semble plus délicate à implémenter que la version orientée *symptômes* du fait qu'il s'agit d'amorcer un *APE-cycle* (i.e. *Analyze*) et d'effectuer un éventuel retour en arrière (i.e. *rollback*) si aucune solution (i.e. *tactique*) n'est trouvée face au *symptôme*. Néanmoins, cette approche assure de ne pas contraindre vainement l'adaptation en bloquant des ressources inutilement.

Traitement des *symptômes* non bloquants

Nous avons vu dans les approches précédentes que les *symptômes*, consécutivement positionnés en tête de la *symptoms queue*, sont traités l'un après l'autre dans des *APE-cycles* s'exécutant en parallèle tant que les ressources concernées par les *symptômes* ne rentrent pas en conflit avec les ressources des autres *symptômes* en cours de traitement (i.e. contrôle orienté *symptômes*) ou avec les ressources impactées par les *tactiques* en cours d'exécution (i.e. contrôle orienté *tactiques*).

Dans le cas où un conflit de ressources est identifié concernant le *symptôme* en tête de queue, plusieurs solutions sont possibles. Une première solution revient à attendre la libération des ressources bloquantes c'est-à-dire la terminaison d'un *APE-cycle* (cf. *solution1* des Figures 7.15 et 7.16) tandis qu'une seconde solution revient à attendre la venue d'un nouveau *symptôme* en tête de la queue (cf. *solution2* des Figures 7.15 et 7.16). Les deux solutions évoquées reviennent à attendre un nouvel événement (i.e. fin d'un *APE-cycle* ou nouveau *symptôme*) ce qui implique d'interrompre le déclenchement de nouvelles adaptations pour une durée incertaine alors que la queue contient potentiellement d'autres *symptômes* qu'il serait possible de traiter.

Une autre solution possible est de considérer et de traiter les *symptômes* non bloquants de la *symptoms queue* (cf. *solution3* des Figures 7.15 et 7.16). Cela revient à inspecter les éléments suivants dans la queue sans se limiter uniquement au *symptôme* en tête de celle-ci lorsque celui-ci devient bloquant. Il s'agit alors d'évaluer les éléments suivants jusqu'à trouver un *symptôme* qu'il est possible de traiter. D'une certaine manière, cette approche remet en cause le tri de la *symptoms queue* reposant sur les *scores* des *symptômes*. En effet, le fait de progresser dans la queue et de traiter un *symptôme* "lointain" n'est pas anodin. Le résultat du traitement d'un *symptôme*, bien que ne rentrant pas en conflit avec les reconfigurations courantes, va contraindre davantage le contexte d'adaptation et ainsi empêcher potentiellement de traiter le *symptôme* en tête de queue ayant une plus grande priorité (i.e. *score*). On peut alors se poser la question s'il est préférable d'attendre que les ressources soient libérées pour traiter le *symptôme* en tête de queue (cf. *solution1*) ou de traiter un autre *symptôme* en prenant le risque de repousser le traitement du premier en contraignant davantage le contexte d'adaptation.

Afin de résoudre ce compromis entre l'attente et l'anticipation, il est possible d'imaginer des solutions hybrides où l'on peut limiter l'inspection en avant de la queue à un certains nombres d'éléments en définissant par exemple qu'il est possible de traiter jusqu'au troisième *symptôme* en tête de queue, s'ils sont tous bloquants, on attend la libération des ressources. Une autre possibilité est de limiter l'inspection dans la queue aux *symptômes* admettant un *score* égale ou proche du *symptôme* en tête de queue (e.g. *scoreHead - scoreSymptom* > 2). La mise en place d'une telle gestion de la *symptoms queue* est complémentaire aux contrôles orientés *symptômes* et *tactiques*.

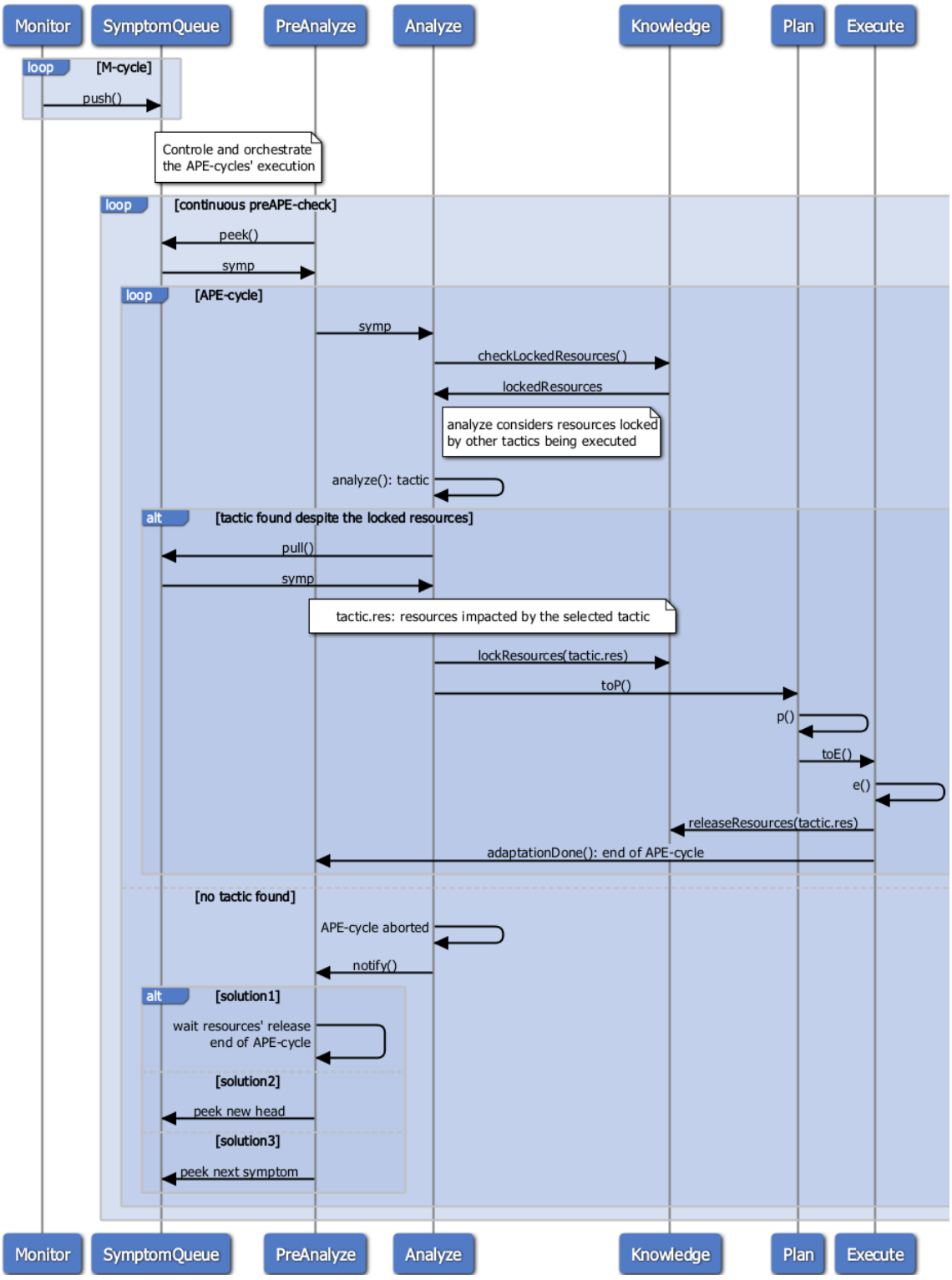


FIGURE 7.16 – Adaptation parallèle : contrôle orienté *tactiques*.

III

Évaluations et conclusion

8

Évaluations des contributions

Nous nous intéressons dans ce chapitre aux évaluations effectuées dans le cadre de nos travaux. La section 8.1 porte sur les expérimentations menées récemment pour une soumission à *Cluster 2016* [rec12]. Nous portons un regard particulier sur l'impact énergétique de l'*élasticité logicielle* ainsi que sur la *QoS*. La section 8.2 présente quant à elle les expérimentations réalisées ainsi que les résultats obtenus pour notre article publié à *CAC 2015* [cac15]. Cet article présente le modèle d'élasticité multi-couche vu dans cette thèse et valide expérimentalement l'approche autonomique. Enfin, nous portons une réflexion sur le passage à l'échelle de notre solution de gestion autonomique de l'élasticité multi-couche dans la section 8.3.

Contents

8.1 Évaluations : empreinte énergétique

Nous avons évoqué plus tôt dans ce manuscrit que l'*élasticité logicielle* offrait la possibilité de passer à l'échelle sans nécessairement avoir recours à l'élasticité de l'infrastructure qui implique de démarrer davantage de ressources IaaS, synonyme d'augmentation de la consommation énergétique ainsi que de coûts supplémentaires du point de vue du fournisseur de service (i.e. *SaaS provider*). Ainsi, l'objectif principal des expérimentations que nous allons présenter ici est de mettre en avant l'impact de l'*élasticité logicielle* d'un point de vue énergétique ainsi que ses répercussions en termes de QoS, de performance, de coût, etc.

8.1.1 Protocole d'expérimentation

Nous allons décrire dans cette partie toutes les composantes de notre protocole d'expérimentation.

Environnement

La Figure 8.1 donne une vision globale de l'architecture mise en place dans le cadre de ces expérimentations.

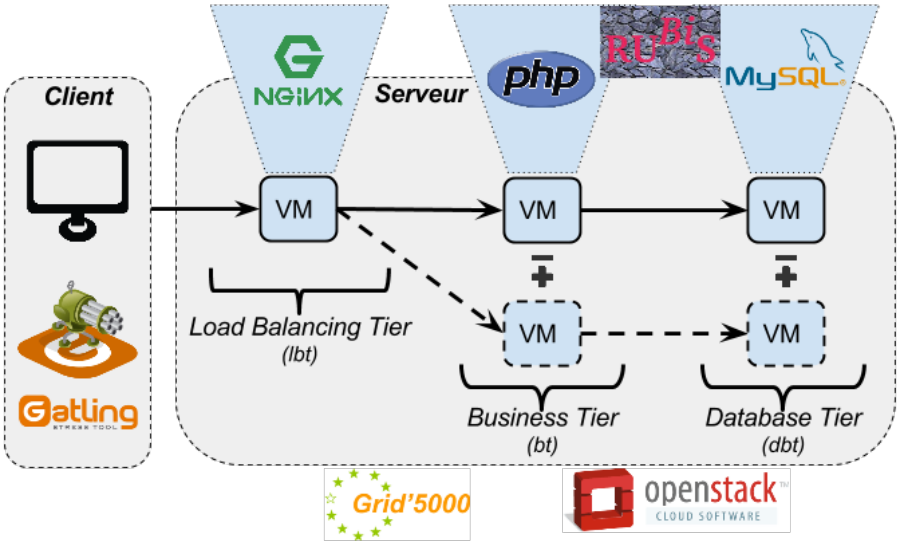


FIGURE 8.1 – Architecture globale : application et infrastructure.

Application : Notre application repose sur le banc d'essai *RUBiS* (i.e. *Rice University Bidding System* [rub15c]). *RUBiS* est une implémentation (*PHP* et *MySQL*) de site de vente aux enchères qui repose sur une architecture multi-tier et propose les trois grandes fonctionnalités suivantes : la vente, la navigation et les enchères. *RUBiS* distingue trois profils d'utilisateurs : visiteur, acheteur et vendeur. Dans le cadre de nos expérimentations, nous avons eu recours à la fonctionnalité de navigation en tant que visiteur. Les sites *e-commerce* constituent un bon cas d'utilisation pour nos travaux du fait de la charge de travail variable qu'ils peuvent essuyer. L'application web proposée par *RUBiS* n'inclut pas la notion d'*élasticité logicielle*. Pour cette raison, le banc d'essai a été modifié par nos soins, en suivant l'approche de [KMÅHR14], en vue d'ajouter de l'*élasticité logicielle verticale* (i.e. *Scale Up* et *Scale Down*). Notre implémentation de l'*élasticité logicielle verticale* repose sur le fait de proposer différents niveaux de recommandation d'articles à l'utilisateur lors de sa navigation. La recommandation permet de suggérer à l'utilisateur des articles en fonction de son panier courant, de ses achats passés, des achats des autres utilisateurs, etc. Ce type de fonctionnalité est implémentée par des composants (i.e. moteurs de recommandation) mettant en œuvre différents algorithmes (e.g. basés sur le comportement des clients, sur la nature des produits, etc.). La fonctionnalité de recommandation, bien que importante pour le vendeur, repose sur des algorithmes complexes sollicitant grandement les ressources IaaS sous-jacentes.

Nous avons modifié l'application *RUBiS* pour qu'elle puisse proposer 3 Off_{soft} différentes se traduisant par différentes configurations de l'application, avec des niveaux de *QoF* distincts, plus ou moins consommateurs de ressources. Ces *offerings* sont implémentées en faisant varier le nombre d'algorithmes de recommandation utilisés. Ainsi, on distingue les Off_{soft} *noReco*, *oneReco* et *twoReco* qui mettent en œuvre zéro, un ou deux algorithmes de recommandation. Les différentes Off_{soft} vont se traduire par des traitements plus ou moins complexes et consommateurs des ressources. Le premier algorithme de recommandation (utilisé par *oneReco* et *twoReco*) va retourner une recommandation dite *user-based* [rec12] pour laquelle on va s'intéresser aux similarités entre l'utilisateur actif et d'anciens utilisateurs (e.g. les utilisateurs du même âge aimant la musique achètent des casques audio). Le second algorithme (utilisé pour *twoReco*) va quant à lui retourner des recommandations *item-based* [KMÅHR14], c'est-à-dire qu'il tend à rechercher en premier lieu des similarités entre produits (i.e. *items*) et ensuite de faire une recommandation à l'utilisateur (e.g. la housse de protection correspondant à la guitare ajoutée au panier par l'utilisateur).

Infrastructure : Notre infrastructure repose sur *Grid5000* [g5k15] et plus précisément sur le *cluster Taurus* du site de Lyon dont les nœuds sont équipés de sondes énergétiques (i.e. Wattmètre) qu'il est possible d'interroger via une API en vue de récupérer la consommation énergétique courante des PMs utilisées. Nous avons eu recours à 2 PMs *Dell PowerEdge R720* (CPU : 2cpus 6cores, Intel Xeon E5-2630, 2.30Ghz / RAM : 32Go / Disk : 600Go HDD) du site de Lyon (réseau : 10Go/s Ethernet) dans le cadre de ces expérimentations.

La couche de virtualisation repose sur *OpenStack* [ope15b] (*OpenStack Grizzly* installé sur une PM jouant le rôle de *cloud controller*). Nous avons considéré deux types de VM pré-configurées (i.e. Off_{infra}) proposés par *OpenStack* : *xlarge* (CPU : 8cores / RAM : 16Go) et *small* (CPU : 1core, RAM : 2Go), toutes deux fournies avec l'OS *ubuntu-12.04-server*.

Répartiteur de charge : Nous avons modifié quelque peu l'architecture de *RUBiS* en ajoutant un tier de répartition de charge (cf. *Load Balancing Tier - lbt*) qui repose sur l'outil *Nginx* [ngi15]. Le but est simplement de répartir la charge de travail en entrée (i.e. *workload*) aux multiples VMs qui constituent le second tier (cf. *Business Tier - bt*).

Injecteur de charge : Nous avons eu recours à l'outil *open source Gatling* [gat15] en tant qu'injecteur de charge pour simuler les requêtes *HTTP* clientes et stresser l'application *RUBiS*.

Scénario d'évaluation

Expériences réalisées : Nous avons effectué 4 expériences que nous allons détailler ci-dessous :

- **expérience 1 :** la première expérience consiste à passer au banc d'essai l'application *RUBiS* sans *élasticité logicielle*. Ainsi, aucune adaptation SaaS n'est réalisée, c'est-à-dire que l'Off_{soft} de l'application est fixée au départ et restera la même tout au long de l'expérience. Dans le cas de cette expérience, l'Off_{soft} est fixée à *noReco* (i.e. 0 recommandation) ;
- **expérience 2 :** aucune *élasticité logicielle* avec Off_{soft} fixée à *oneReco* (i.e. une recommandation suggérée à l'utilisateur) tout le long de l'expérience ;
- **expérience 3 :** aucune *élasticité logicielle* avec Off_{soft} fixée à *twoReco* (i.e. deux recommandations suggérées à l'utilisateur) tout le long de l'expérience ;
- **expérience 4 :** contrairement aux expériences précédentes, l'expérience 4 met en place l'*élasticité logicielle* au travers d'un contrôleur que nous appelons *QoS-aware*. Ce contrôleur repose sur les règles à base de seuils (cf. section 2.3.3) et permet de reconfigurer dynamiquement l'application (i.e. changer l'Off_{soft} initialement paramétrée à *twoReco*) en fonction de la QoS observée. Ainsi, la partie "condition" des règles à base de seuils concernent la métrique de temps de réponse (*response time* - RT). Ces temps de réponse sont collectés au niveau du *lbt* et plus précisément dans le fichier *access.log* du répartiteur de charge *Nginx* qui contient plusieurs informations pour chaque requête : adresse IP du client, URL appelée, temps de réponse, codes de statut *HTTP*, etc.

Nous définissons deux seuils, nommés *lowRTth* et *highRTth*, dont résultent trois états possibles auxquels vont correspondre les 3 Off_{soft} *noReco*, *oneReco* et *twoReco*. Ainsi, nous définissons les 3 règles suivantes qui vont réguler l'adaptation du système :

— ***Règle 1*** *:*
If(currentRT < lowRTth) for time_window
Then switch Off_{soft} *to twoReco*

— ***Règle 2*** *:*
If(lowRTth < currentRT < highRTth) for time_window
Then switch Off_{soft} *to oneReco*

— ***Règle 3*** *:*
If(currentRT > highRTth) for time_window
Then switch Off_{soft} *to noReco*

Les trois règles reposent sur le compromis *QoS-QoF* (cf. Figure 5.1). Pour rappel, un contenu fonctionnel riche (e.g. *twoReco*) va solliciter davantage les ressources IaaS (e.g. CPU, RAM, etc.), ce qui risque, à ressource IaaS constante comme c'est le cas dans cette expérience, de faire baisser la QoS (i.e. $QoF \propto \frac{1}{QoS}$). Ainsi, on va ajuster la *QoF* de l'application *RUBiS* en fonction de la QoS observée, c'est-à-dire réduire celle-ci lorsque les temps de réponse sont élevés et réciproquement. Le fonctionnement général du contrôleur *QoS-aware* est donné dans l'Algorithme 2.
Il faut noter que les parties "condition" des règles décrites ci-dessus peuvent être définies et remontées sous forme d'événements (i.e. *symptômes perCEPtion*) et que les parties "action" peuvent être décrites avec le langage *ElaScript* formant ainsi 3 *tactiques d'élasticité*. Nous donnons dans la Figure 8.2 la définition des requêtes *EPL* (cf. sous-section 6.1.4) correspondant aux *symptômes* nommés *AppLowRT*, *AppNormalRT* et *AppHighRT* ainsi que les plans de reconfiguration *ElaScript* associés dans la Figure 8.3.

Algorithme 2 Prise de décision du contrôleur *QoS-aware*.

INPUT :
$currentRT$: current reponse time observed
$lowRTth$, $highRTth$: : low and high thresholds of reponse time
app : application
OUTPUT : reconfiguration plan for the application

```
if currentRT < lowRTth then :                    ▷ Good QoS
    app.switchToTwoReco() ;
else if currentRT > highRTth then :              ▷ Bad QoS
    app.switchToNoReco() ;
else                                             ▷ Normal QoS
    app.switchToOneReco() ;
```

Configuration initiale : La configuration initiale de l'infrastructure est identique pour les 4 expériences. Les tiers *lbt* et *bt* sont respectivement constitués de 1 VM et de 3 VMs de type *small* (i.e. Off_{infra} = *small*). Le tier *dbt* est constitué d'une unique VM de type *xlarge*. Le choix de l'Off_{infra} *xlarge* pour le tier *dbt* s'explique par le fait que l'application *RUBiS* modifiée par nos soins (i.e. ajout des recommandations) sollicite davantage les ressources de ce dernier tier (i.e. *Data-intensive applications*). De plus, nous avons convenu de cette configuration afin d'utiliser toutes les ressources de la PM sous-jacente, à savoir les 12 cœurs de CPU. Toutes les VMs sont pré-configurées avec les composants des tiers associés (e.g. *MySQL* pour le *dbt*). Comme nous l'avons évoqué lors de la description des expériences, l'application est quant à elle paramétrée avec différentes Off_{soft}. Enfin, le contrôleur *QoS-aware* chargé de l'adaptation de l'application dans le cas de l' *expérience 4* est isolé sur sa propre PM et déployé sur une VM de type *small*.

```
String appLowRT_req = "select app, avg(value)" +
" from AppRT.win:time(" + timeWindow + ") where avg(value)<" + lowRTth;
```

(a) Requête *EPL* correspondant au *symptôme AppLowRT*.

```
String appNormalRT_req = "select app, avg(value)" +
  " from AppRT.win:time(" + timeWindow + ")" +
    " where avg(value) between " + lowRTth + " and " + highRTth;
```

(b) Requête *EPL* correspondant au *symptôme AppNormalRT*.

```
String appHighRT_req = "select app, avg(value)" +
" from AppRT.win:time(" + timeWindow + ") where avg(value)>" + highRTth;
```

(c) Requête *EPL* correspondant au *symptôme AppHighRT*.

FIGURE 8.2 – *Symptômes* correspondant aux seuils des règles.

Scénario de charge : Nous avons soumis notre système à 2 scénarii de charge pour les 4 expériences évoquées ci-dessus. Les 2 scénarii s'appuient sur des traces réelles de charge de travail des sites *Wikipedia* [wik16] et *FIFA World Cup '98* [fif16] ramenées à de valeurs moindres pour une durée totale de *96min*. Nous reposons sur ces deux scénarii de charge bien connus du monde scientifique du fait qu'ils constituent des types de *workload* distincts qui, comme nous allons le voir, présentent des résultats différents. Dans notre cas, la valeur de la charge de travail à un instant t est définie par le débit d'entrée du système, c'est-à-dire le nombre de requêtes/sec envoyé à l'application par les utilisateurs/clients (i.e. injecteur de charge *Gatling*).

```
1 begin
2 when AppLowRT(app)
3 do
4     // Switch the application's offering to twoReco: 2
5     app.sus(2);
6 end
```

(a)

```
1 begin
2 when AppNormalRT(app)
3 do
4     // Switch the application's offering to oneReco: 1
5     app.sds(1);
6 end
```

(b)

```
1 begin
2 when AppHighRT(app)
3 do
4     // Switch the application's offering to noReco: 0
5     app.sds(0);
6 end
```

(c)

FIGURE 8.3 – *Tactiques ElaScript* correspondant aux règles.

8.1.2 Résultats

Nous allons présenter les résultats obtenus pour nos deux scenarii de charge (i.e. *Wikipedia* et *FIFA*) concernant les 4 expériences évoquées précédemment que nous référencerons respectivement comme les implémentations *noreco* (i.e. 0 recommandation), *onereco* (i.e. 1 recommandation), *tworeco* (i.e. 2 recommandations) et *QoSaware* (i.e. avec *élasticité logicielle verticale* permettant de changer dynamiquement l'Off_{soft} pour calculer et retourner 0, 1 ou 2 recommandation(s)).

Scénario Wikipedia

Nous allons dans un premier temps nous intéresser aux résultats des expériences concernant le scénario *Wikipedia*. La plupart des résultats sont issus des rapports générés à la suite d'une expérimentation par notre injecteur de charge *Gatling* [gat15].

Résultats généraux : La Figure 8.4 présente les rapports statistiques obtenus pour chacune des expériences réalisées et dans laquelle les sous-figures 8.4a, 8.4b, 8.4c et 8.4d correspondent respectivement aux implémentations *noreco*, *onereco*, *tworeco* et *QoSaware*.

Une première remarque intéressante concerne le nombre de requêtes tombées en erreur (i.e. code statut *HTTP* différent de *200*) dans le cas de l'implémentation *tworeco* (cf. *24%* requêtes *KO* Figure 8.4c) qui s'avère être trop consommatrice de ressources IaaS (i.e. CPU et RAM) entrainant ainsi l'indisponibilité de l'application *RUBiS* (i.e. majoritairement des code statut *HTTP 408 request timeout* et *504 gateway timeout*) lorsque la charge de travail est importante. De même, on remarquera que cette implémentation admet des temps de réponse moyen supérieur *10sec* (cf. *Mean*) ce qui est beaucoup trop important pour un site E-commerce. Les implémentations *noreco* et *onereco* admettent des résultats similaires (i.e. très peu d'erreur), avec des temps de réponse sensiblement plus longs pour l'implémentation retournant 1 recommandation. L'implémentation *QoSaware*, quant à elle, présente un taux d'erreur de *4%* avec des temps de réponse moyen de l'ordre de *2.5sec*. Nous reviendrons par la suite sur les résultats de l'implémentation *QoSaware*. La Figure 8.5 indique la répartition des requêtes en termes de temps de réponse (et éventuellement la proportion de celles-ci tombées en erreur) pour les 4 expériences *noreco* (cf. Figure 8.5a), *onereco* (cf. Figure 8.5b), *tworeco* (cf. Figure 8.5c) et *QoSaware* (cf. Figure 8.5d).

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1809956	1809956	0	0%	318.66	1	8	22	56	73	249	16	18

(a)

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1810328	1810328	0	0%	318.732	1	13	37	68	101	987	23	23

(b)

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1807776	1370418	437358	24%	315.916	0	794	7498	60006	60010	60042	10726	22315

(c)

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1810448	1729127	81321	4%	318.739	0	79	572	14443	60007	60032	2603	10270

(d)

FIGURE 8.4 – Statistiques globales - *Wikipedia*.

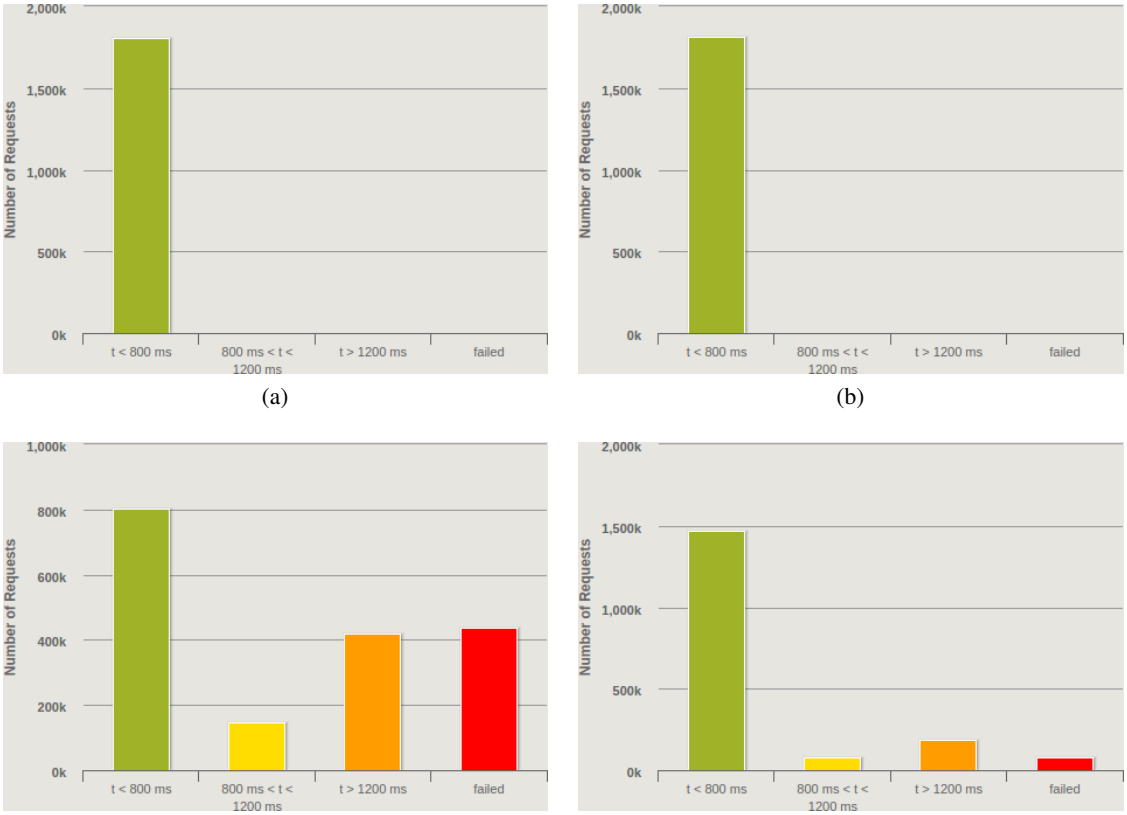
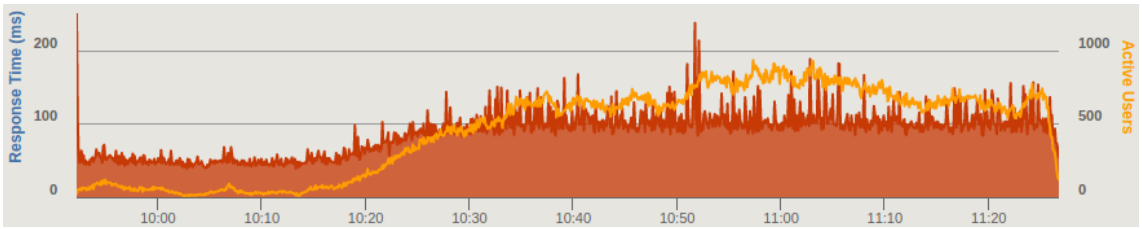


FIGURE 8.5 – Répartition des requêtes - *Wikipedia*.

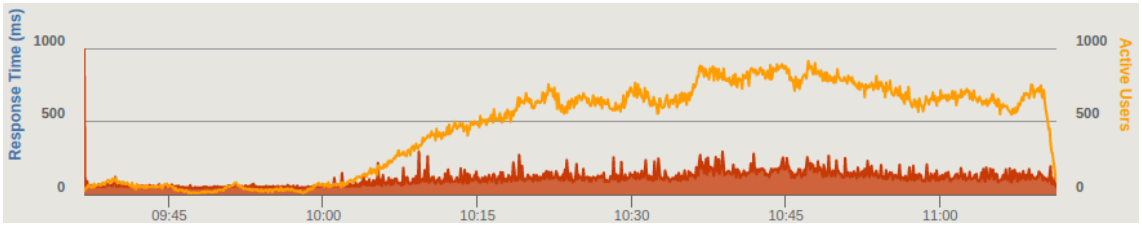
QoS (temps de réponse et disponibilité) : Les Figures 8.6 et 8.7 mettent en avant les résultats en termes de *QoS* en présentant respectivement les *temps de réponse* observés et la disponibilité (i.e. *availability*) de l'application pour nos 4 expériences. On retrouve pour chacune des figures le nombre d'utilisateurs actifs (cf. *Active Users*, courbes orange). Il ne s'agit pas de la charge de travail entrante (i.e. *workload*) mais du nombre d'utilisateurs dont les requêtes sont en cours de traitement ou en attente (ce qui explique la variation selon les différentes expériences). Ainsi, plus les ressources IaaS sont surchargées, plus les temps de réponse vont grimper ce qui entraine un nombre d'utilisateurs actifs plus important, etc.

On retrouve les bons résultats évoqués précédemment pour les implémentations *noreco* et *onereco* en termes de temps de réponse (cf. Figures 8.6a et 8.6b) et de disponibilité (cf. Figures 8.7a et 8.7b). On remarquera que l'implémentation *noreco* admet une meilleure *QoS* que *onereco*, en revanche, cette dernière offre une meilleure *QoF* (i.e. calcule/retourne une recommandation par requête). On retrouve ici le compromis *QoF*-*QoS* à ressource IaaS constante évoqué dans la sous-section 5.1.1 (cf. Figure 5.1). Concernant l'implémentation *tworeco*, la *QoS* est très mauvaise lorsque la charge devient importante. On remarquera des temps de réponse oscillant entre *30sec* et *60sec*, ce après quoi la requête est considérée en erreur (i.e. *time out*). Cette durée maximum d'attente pour une requête peut être paramétrée au niveau du serveur *HTTP* ou du répartiteur de charge. La mauvaise *QoS* s'explique du fait que l'implémentation *tworeco* calcule et retourne 2 recommandations distinctes pour chaque requête.

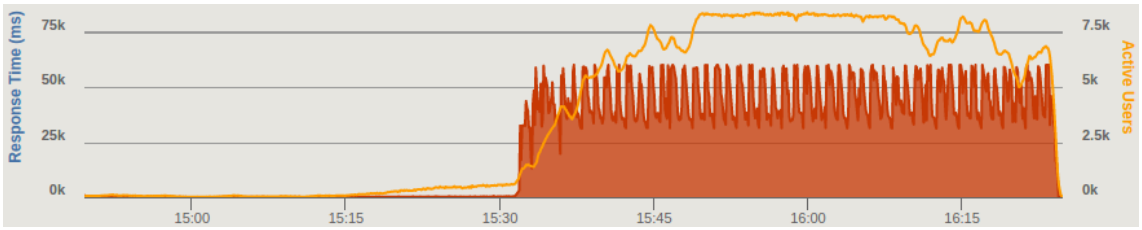
Nous avons noté lors de nos tests de calibrage que les deux algorithmes de recommandation utilisés unitairement, bien que sollicitant les ressources IaaS, ne surchargeaient pas complètement celles-ci (e.g. implémentation *onereco* dont on constate l'*overhead* de la recommandation tout en admettant une bonne *QoS*). Cependant, nous avons aussi noté que l'utilisation conjointe des deux algorithmes dans une même implémentation, comme c'est le cas de l'implémentation *tworeco*, faisait croître la consommation de ressources IaaS de manière quasi exponentielle et non linéaire. Cela explique les mauvais résultats obtenus en termes de *QoS* lorsque la charge entrante est importante et que l'application est paramétrée avec l'Off_{soft} qui calcule et retourne 2 recommandations (cf. implémentation *tworeco* et dans une moindre mesure *QoSaware*).



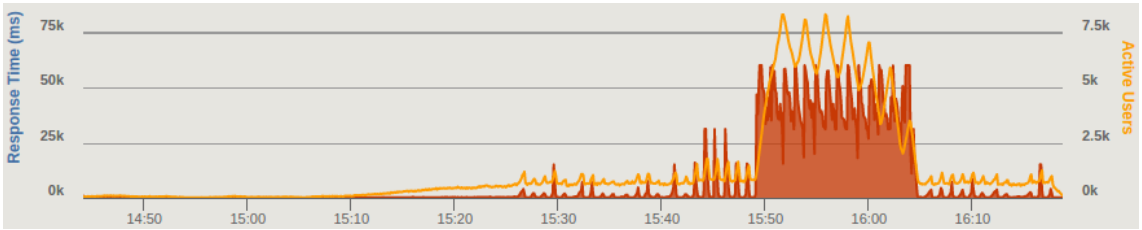
(a)



(b)



(c)

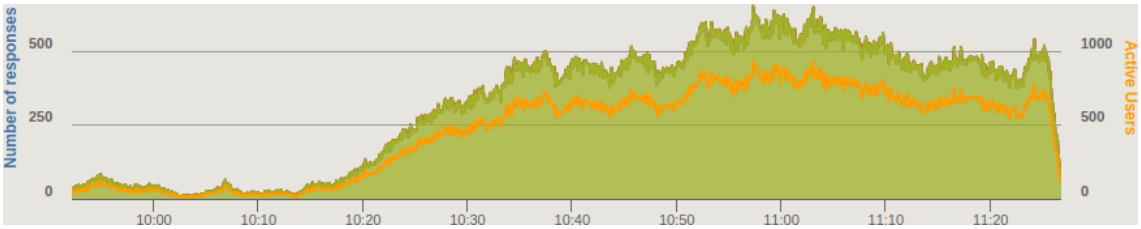


(d)

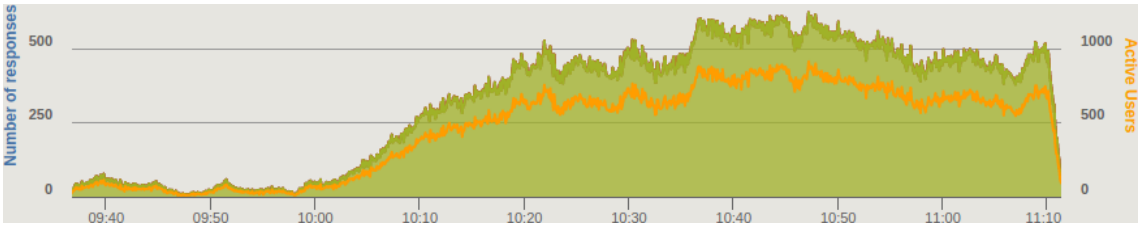
FIGURE 8.6 – Temps de réponse (requêtes *OK*) - *Wikipedia*.

Dans le cas du scénario *Wikipedia*, la charge entrante grimpe considérablement et reste importante pendant la deuxième moitié du scénario (i.e. à partir de *48min*). En ce sens, il ne s'agit pas de courts pics de charge comme c'est le cas du scénario *FIFA* que nous aborderons ensuite. De cette charge importante et continue va résulter la création d'une file d'attente pour les requêtes en attente de traitement (i.e. pouvant rester jusqu'à *60sec* dans cette file d'attente avant d'être considérées en erreur). La taille de cette file d'attente est étroitement liée au nombre d'utilisateurs actifs (cf. courbe orange).

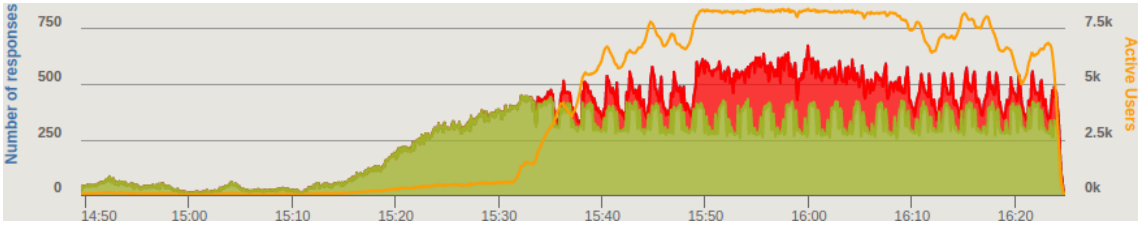
On peut ainsi voir sur les Figures 8.6a, 8.6b, 8.7a et 8.7b que le nombre d'utilisateurs actifs à chaque instant est quasiment identique du fait que la charge entrante et la sollicitation des ressources IaaS ne sont pas suffisamment importantes pour nécessiter la création d'une telle file d'attente (cf. *Active Users < 1000*). Ainsi, toutes les requêtes sont traitées dans les temps et l'application est toujours disponible (cf. Figures 8.7a et 8.7b). En revanche, les implémentations *tworeco* et *QoSaware* voient le nombre d'utilisateurs actifs augmenter considérablement (cf. *Active Users > 7500*) ce qui correspond à une file d'attente importante du fait que les ressources IaaS sont surchargées (i.e. sollicitées par les deux algorithmes de recommandation) et donc une baisse de la *QoS*. On remarquera cependant que l'implémentation *QoSaware* a vu sa *QoS* chuter moins de *15min* (cf. *15 :50-16 :04*, Figures 8.6d et 8.7d) tandis que l'implémentation *tworeco* a connu des difficultés en termes de *QoS* pendant presque *1h* (cf. *15 :30-16 :25*, Figures 8.6c et 8.7c).



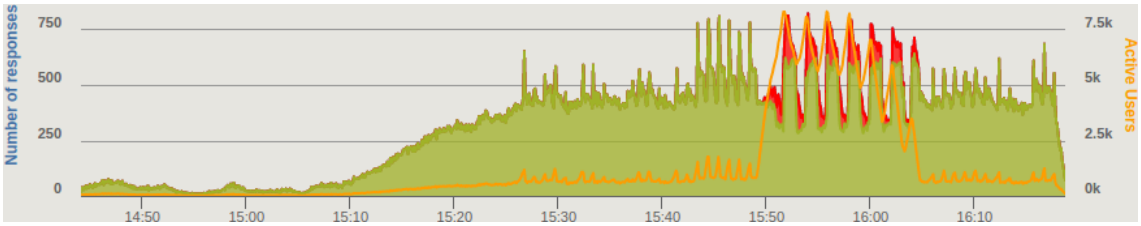
(a)



(b)



(c)



(d)

FIGURE 8.7 – Disponibilité - *Wikipedia*.

Nous avons pu observer que l'Off_{soft} calculant et retournant 2 recommandations s'avère beaucoup trop gourmande en termes de ressources IaaS face à la charge entrante du scénario *Wikipedia* (i.e. deuxième moitié du scénario) ainsi que l'infrastructure modeste (et non élastique) utilisée pour ces expériences. Cependant, comme nous allons le voir, l'implémentation *QoSaware* a été en mesure d'adapter l'application (i.e. *QoF*) et ainsi limiter les répercussions de cette charge entrante en termes de *QoS*. De plus, nous verrons que l'implémentation *QoSaware* offre des résultats beaucoup plus satisfaisants dans le cas du scénario *FIFA* qui admet de courts pics de charge.

***QoF* et consommation énergétique** : Nous allons nous intéresser ici à la *QoF* de l'application *RUBiS* ainsi qu'à la dimension énergétique de l'*élasticité logicielle*.

La Figure 8.8 présente la *QoF* de l'application pour le scénario *Wikipedia* c'est-à-dire les Off_{soft} utilisées pour les 4 expériences réalisées. On constate que les trois premières implémentations, à savoir *noreco* en bleu, *onereco* en orange et *tworeco* en vert, admettent la même Off_{soft} tout le long de l'expérience (resp. *0*, *1* et *2*). L'implémentation *QoSaware* en rouge, quant à elle, admet différentes Off_{soft} du fait qu'elle met en œuvre l'*élasticité logicielle verticale* au travers d'un contrôleur basé sur les temps de réponse observés (cf. Algorithme 2).

Le contrôleur va ainsi prendre une décision de reconfiguration toutes les *20sec*. Il s'agit respectivement de passer l'Off_{soft} à *0*, *1* ou *2* lorsque la moyenne des temps de réponse (i.e. 20 dernières secondes) est supérieure au *highRTth* (i.e. *1sec*), entre *lowRTth* et *highRTth* ou inférieure à *lowRTth* (i.e. *0.5sec*). Dans le cas de nos expériences, nous avons considéré le 95^e *centile* pour agréger les temps de réponse toutes les *20sec* (et non la moyenne).

La Figure 8.8 montre que l'implémentation *QoSaware* commence à déclencher des reconfigurations à partir de *45min*, ce qui correspond au moment où la charge entrante du scénario *Wikipedia* devient importante, et ce jusqu'à la fin du scénario. On remarquera aussi que l'Off_{soft} devient constante à *0* pendant environ *10min*. Ces *10min* correspondent pourtant au moment où l'on essuie des problèmes de *QoS* (cf. Figures 8.6d et 8.7d) alors qu'on pourrait s'attendre à une bonne *QoS* du fait que la *QoF* est à son minimum (i.e. compromis *QoF*-*QoS*).

Cela s'explique du fait que notre implémentation exécute les requêtes suivant l'Off_{soft} paramétrée au moment de leur réception et non avec la nouvelle Off_{soft}. Dans notre cas, les requêtes "estampillées" avec l'Off_{soft}=*2* vont ainsi être placées en file d'attente puis exécutées ultérieurement avec cette Off_{soft} malgré le contexte courant qui n'est pas propice à une telle *QoF* (i.e. Off_{soft} courante vaut *0*). En ce sens, on voit clairement apparaître un point d'amélioration de l'implémentation *QoSaware* qui viserait à exécuter les requêtes en fonction de l'Off_{soft} courante (et non l'Off_{soft} au moment de la réception qui correspond à un contexte d'exécution passé).

Enfin, l'implémentation *QoSaware* donne ici l'impression d'osciller fortement. Cela peut s'expliquer en partie du fait que nous avons fixé une prise de décision toutes les *20sec*. En effet, nous avons ramené le scénario *Wikipedia* originel d'une durée de *24h* à *96min* pour nos expériences. Ainsi, les *20sec* correspondent en réalité à *5min*, ce qui est davantage raisonnable. Cependant, on pourrait aussi imaginer ajouter une période de calme suite à une reconfiguration (i.e. une instruction *wait* dans les *tactiques ElaScript* après les actions SU_{soft} et/ou SD_{soft}).

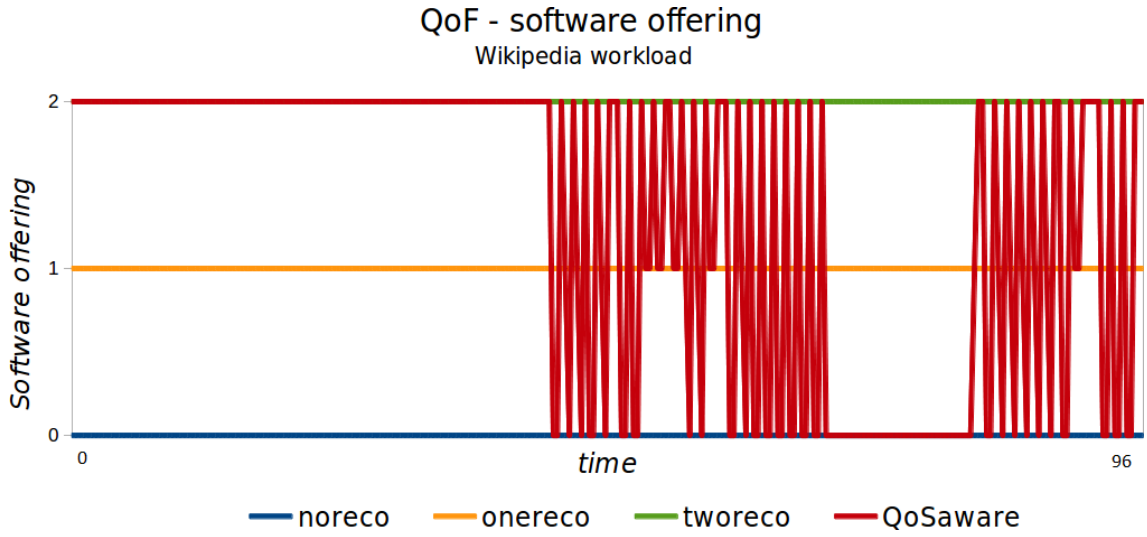


FIGURE 8.8 – *QoF* de l'application - *Wikipedia*.

La Figure 8.9 correspond aux mesures de consommation énergétique réalisées sur la PM hébergeant l'application *RUBiS*. Ces mesures sont rendues possibles du fait que nous nous appuyons sur le *cluster Taurus* du site de *Lyon* de *Grid5000* [g5k15] dont les nœuds sont équipés de sondes énergétiques (i.e. Wattmètre). On retrouve les 4 implémentations *noreco* en bleu, *onereco* en orange, *tworeco* en vert et enfin *QoSaware* en rouge.

De manière générale, les résultats montrent que toutes les implémentations admettent une consommation énergétique semblable au début de l'expérience (cf. *100-120 watt*). Puis, celles-ci voient leurs consommations croître du fait de la sollicitation croissante des ressources IaaS (i.e. CPU, RAM, etc.) lorsque la charge du scénario *Wikipedia* augmente. Cependant, on remarquera que cette augmentation de la consommation énergétique diffère en fonction des implémentations. Intéressons nous tout d'abord aux trois premières implémentations qui ne présentent aucune capacité d'adaptation (i.e. non élastique), concernant la deuxième moitié du scénario *Wikipedia*. Sans grande surprise, l'implémentation *noreco* est l'implémentation qui consomme le moins (cf. *140 watt*), suivie de *onereco* (cf. *155 watt*) puis *tworeco* (cf. *170 watt* avec un pic à *200 watt*).

Nous avons présenté l'*élasticité logicielle* plus tôt dans cette thèse comme le fait de fournir des applications offrant différents niveaux de service (i.e. Off_{soft}) plus ou moins consommateurs de ressources. On voit ici apparaître le fait que l'énergie (au travers de la sollicitation des ressources IaaS telles que le CPU, la RAM, etc.) est elle aussi une ressource impactée par la *QoF*.

Intéressons nous désormais à l'implémentation *QoSaware* qui met en place l'*élasticité logicielle verticale*. On remarquera que l'évolution de la consommation énergétique est quasiment identique à celle de l'implémentation *tworeco* pendant la première moitié du scénario du fait que les deux implémentations utilisent la même Off_{soft}=*2*. Cependant, lorsque le contrôleur commence ses reconfigurations (i.e. oscillant entre l'Off_{soft} *0* et *2*), la consommation énergétique devient moindre et se rapproche de celle de l'implémentation *onereco* (cf. *160 watt*). Puis, à la 60^{eme}*min*, on voit la consommation énergétique baisser pour atteindre le même niveau que l'implémentation *noreco* (cf. *145 watt*) qui correspond au moment où le contrôleur bascule l'Off_{soft} à *0* pour une durée d'environ *10min*. Enfin, les dernières minutes de l'expérience montre une consommation énergétique d'environ *160 watt* du fait que l'Off_{soft} varie entre *0* et *2*. Les résultats obtenus pour l'implémentation *QoSaware* laissent présager des adaptations intégrant d'éventuelles préoccupations énergétiques. Par exemple, il peut s'agir de maintenir une certaine *QoS* face à une demande variable (i.e. *fluctating workload*) tout en respectant des contraintes économiques et/ou énergétiques et ce en jouant sur la *QoF* de l'application. Nous reviendrons plus tard sur cette perspective de passage à l'échelle frugal.

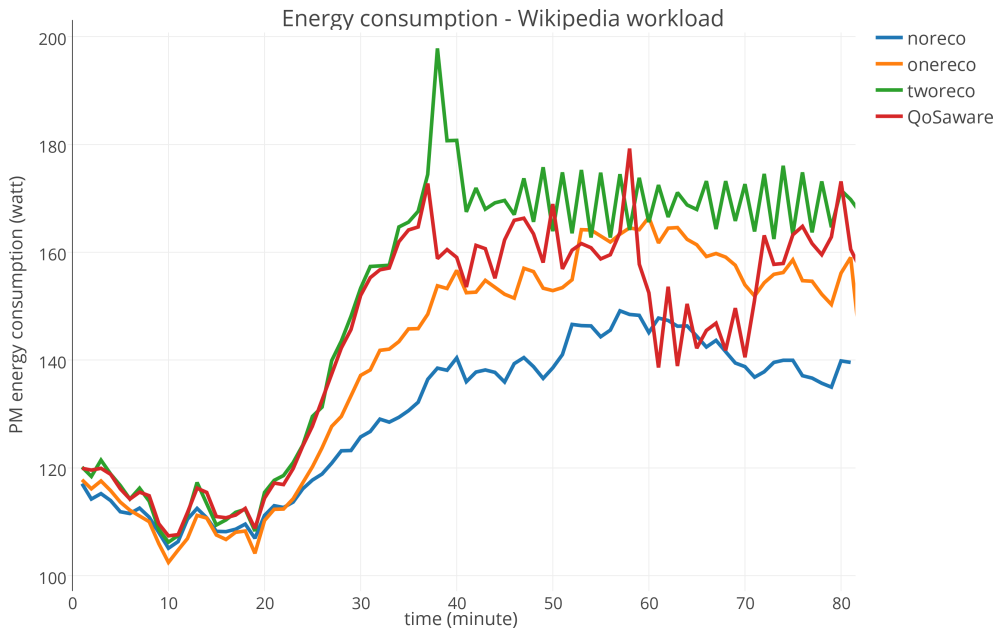


FIGURE 8.9 – Consommation énergétique - *Wikipedia*.

Scénario FIFA

Nous allons présenter ici les résultats des expériences concernant le scénario *FIFA*. Contrairement au scénario *Wikipedia* dont la charge de travail correspond à une augmentation lente et importante, il s'agit ici d'une demande admettant de courts pics de charge plus ou moins importants. Nous avons là aussi ramené la durée du scénario à *96min*. De même, l'implémentation et le paramétrage du contrôleur *QoSaware* est identique que pour le scénario *Wikipedia* présenté précédemment (i.e. mêmes valeurs pour les seuils, mêmes fenêtres de temps, etc.). Comme nous allons le voir, les expériences réalisées avec ce scénario offrent des résultats davantage probants concernant les gains issus de l'*élasticité logicielle*, mise en œuvre dans le cas de l'implémentation *QoSaware* au travers du contrôleur présenté dans l'Algorithme 2.

Résultats généraux : La Figure 8.10 présente les rapports statistiques obtenus pour chacune de nos expériences. Intéressons nous tout d'abord aux résultats concernant les requêtes tombées en erreur (cf. colonnes *KO* et *%KO*) qui indiquent que l'implémentation *tworeco* (cf. Figure 8.10c) admet *2%* de requêtes *KO* soit près de *26000* requêtes. Les implémentations *noreco* (cf. Figure 8.10a) et *onereco* (cf. Figure 8.10b), quant à elles, admettent respectivement *723* et *554* requêtes tombées en erreur. Enfin, l'implémentation *QoSaware* (cf. Figure 8.10d) présente les meilleurs résultats avec seulement *78* requêtes *KO* sur *1150004*, soit plus de *99.99%* de disponibilité. On remarquera également que la moyenne des temps de réponse observés (cf. colonne *Mean*) pour l'implémentation *tworeco* est environ 10 fois supérieure aux autres implémentations (i.e. *QoSaware* compris).

La Figure 8.11 présente la répartition des requêtes en termes de temps de réponse et/ou d'erreur pour les 4 expériences réalisées. On retrouve notamment les résultats de l'implémentation *tworeco* (cf. Figure 8.11c) qui offre certes une *QoF* maximum tout au long de l'expérience mais ce au dépend de la *QoS* (i.e. compromis *QoF*-*QoS* à ressource IaaS constante).

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1151936	1151213	723	0%	200.028	1	21	42	467	1455	60010	119	918

(a)

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1153236	1152682	554	0%	200.227	1	17	38	573	1619	60010	133	1146

(b)

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1150371	1124449	25922	2%	199.742	2	55	910	7890	34964	60024	1745	6449

(c)

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1150004	1149926	78	0%	199.67	1	28	73	895	1946	60008	189	758

(d)

FIGURE 8.10 – Statistiques globales - *FIFA*.

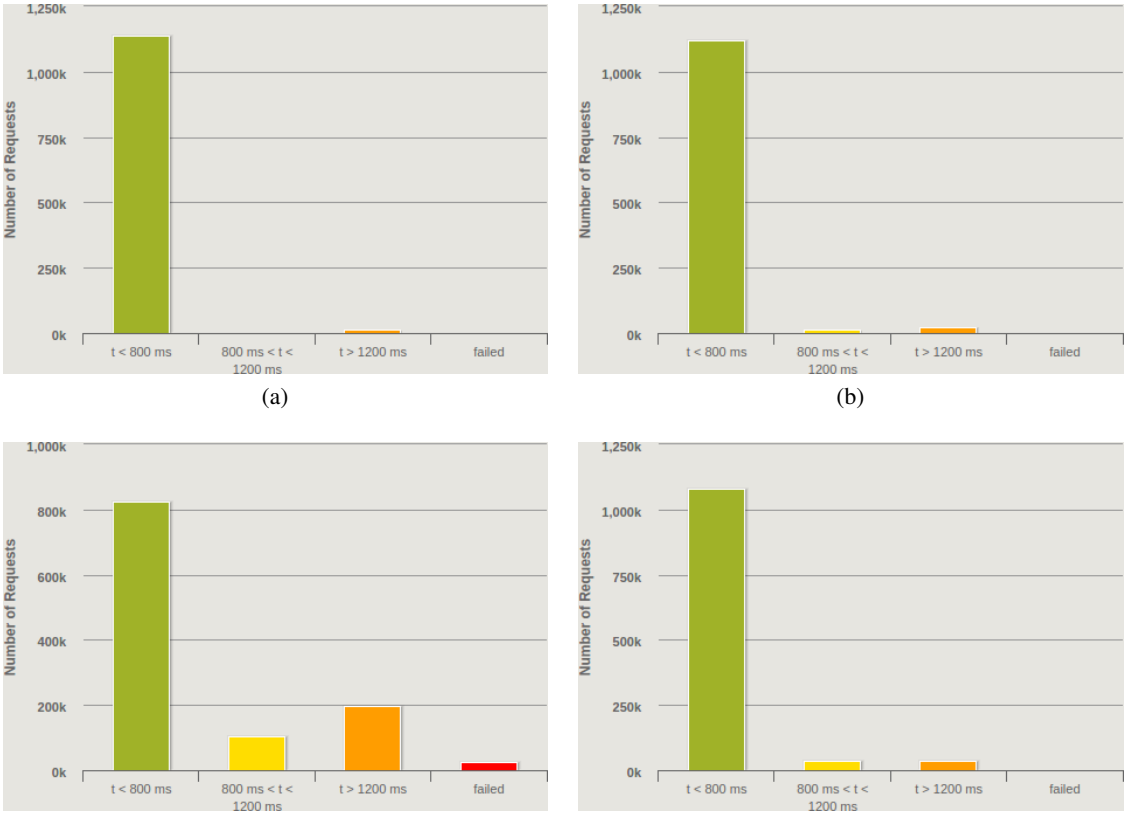


FIGURE 8.11 – Répartition des requêtes - *FIFA*.

QoS (temps de réponse et disponibilité) : Les Figures 8.12 et 8.13 illustrent respectivement les résultats en termes de temps de réponse observés et de disponibilité de l'application *RUBiS* pour nos 4 implémentations. En ce qui concerne les temps de réponse, on retrouve les mêmes tendances que pour les résultats évoqués précédemment à savoir que l'implémentation *tworeco* (cf. Figure 8.12c), du fait des calculs supplémentaires réalisés par les deux algorithmes de recommandation, admet une *QoS* moindre. Ainsi on peut observer des temps de réponse qui augmentent de manière considérable lors des pics de charge (e.g. *15 :30*) jusqu'à atteindre parfois les *60sec* synonyme de requêtes tombées en erreur (i.e. *timeout*). On constate ainsi 9 pics de charge pour lesquels l'implémentation *tworeco* n'a pas été en mesure d'assumer une *QoS* convenable. Pour les trois autres implémentations (cf. Figures 8.12a, 8.12b et 8.12d), seuls deux pics de charge ont fait l'objet d'une telle dégradation des performances.

La Figure 8.13, mettant en avant la disponibilité observée pour les différentes expériences, ne fait que confirmer cette tendance en termes de *QoS*. En effet, on voit apparaître l'incapacité de l'implémentation *tworeco* (cf. Figure 8.13c) à essuyer les pics de charge importants (cf. *2%* de requêtes tombées en erreur en rouge sur la figure) du fait que celle-ci voit ses ressources IaaS sous-jacentes (i.e. CPU et RAM) rapidement surchargées du fait de l'utilisation conjointe des deux algorithmes de recommandation. À contrario, les implémentations *noreco* (cf. Figure 8.13a) et *onereco* (cf. Figure 8.13b) restent quant à elles disponibles. De même, l'implémentation *QoSaware* (cf. Figure 8.13d) reste disponible malgré le fait qu'elle propose elle-aussi 2 recommandations en réponse à la plupart des requêtes (i.e. *QoF* maximum de manière ponctuelle).

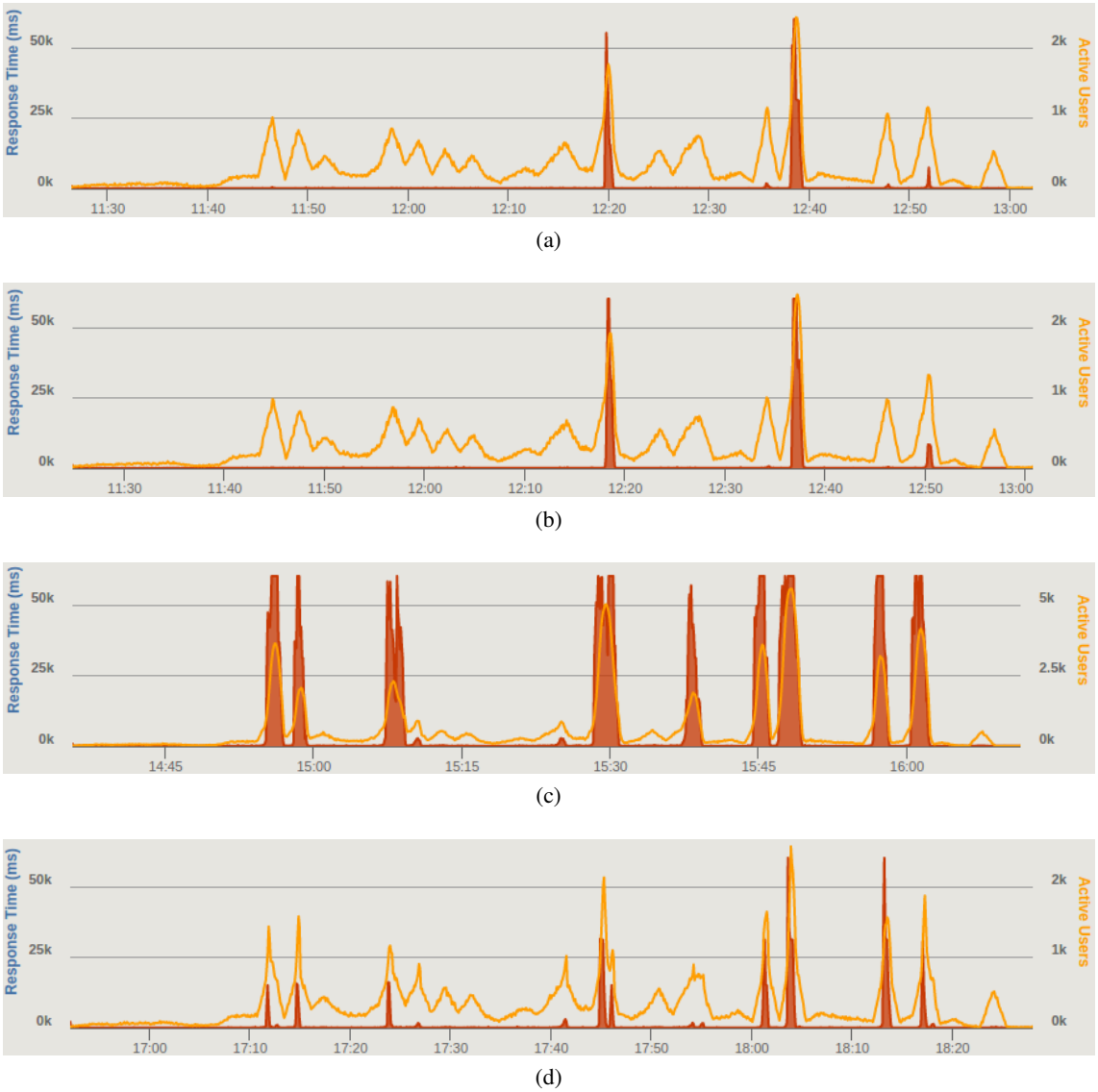
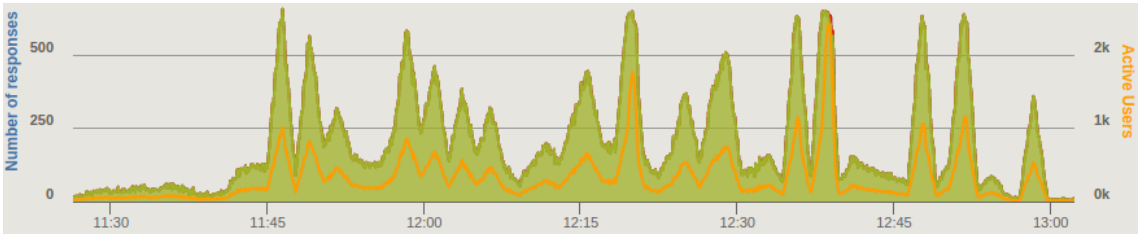


FIGURE 8.12 – Temps de réponse (requêtes *OK*) - *FIFA*.

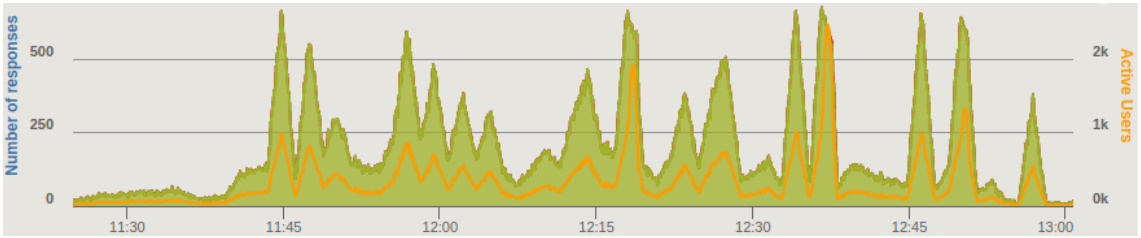
***QoF* et consommation énergétique** : Nous allons traiter ici de la *QoF* de l'application *RUBiS* pour les 4 expériences réalisées à savoir les Off_{soft} utilisées pour chaque implémentation (cf. Figure 8.14). De même, nous allons nous pencher sur la dimension énergétique de l'*élasticité logicielle* (cf. Figure 8.15) en considérant les mesures de consommation énergétique réalisées sur la PM hébergeant l'application *RUBiS*.

Intéressons nous tout d'abord à la Figure 8.14 et plus particulièrement à la courbe rouge qui correspond à l'implémentation *QoSaware* mettant en œuvre l'*élasticité logicielle*. On remarquera que les décisions du contrôleur *QoSaware* ont entrainé très majoritairement l'utilisation de l'Off_{soft}=*2* du fait que la *QoS* était bonne la plupart de l'expérience (i.e. temps de réponse moyens inférieurs à *0.5sec*, cf. Code 8.2a). Néanmoins, lors des pics de charge ayant entrainés une dégradation de la *QoS* (i.e. augmentation des temps de réponse, cf. Codes 8.2b et 8.2c), le contrôleur a déclenché plusieurs reconfigurations de l'application (i.e 28 adaptations à savoir *14* SD_{soft} et *14* SU_{soft} correspondant aux *tactiques d'élasticité* vues dans la Figure 8.3) mais de manière beaucoup plus ponctuelle que dans le cas du scénario *Wikipedia*.

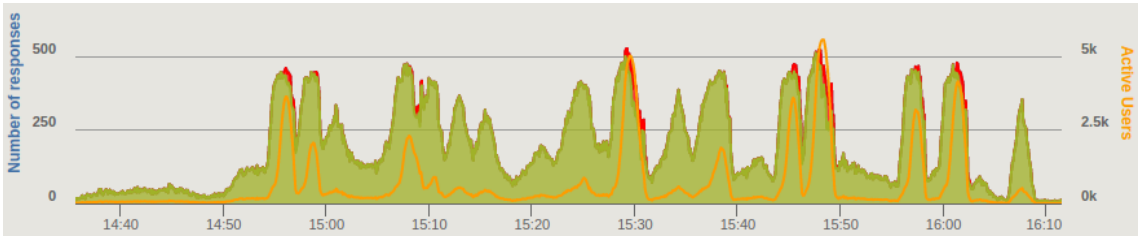
En ce sens, le contrôleur mis en place pour ces expérimentations (paramétré de manière identique) semble plus adapté à des charges de travail de type *FIFA*, c'est-à-dire admettant des pics de charges courts et ponctuels, plutôt qu'à des charges de travail de type *Wikipedia* qui bénéficierait sans doute davantage de l'élasticité de l'infrastructure (e.g. SO_{infra}).



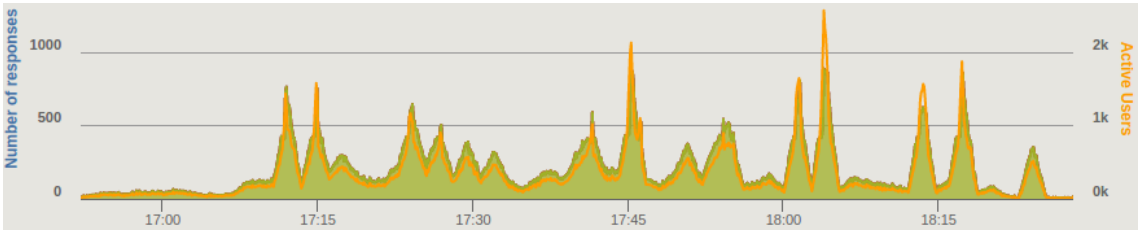
(a)



(b)



(c)



(d)

FIGURE 8.13 – Disponibilité - *FIFA*.

La Figure 8.15 présente les mesures de consommation énergétique réalisées sur la PM hébergeant l'application *RUBiS*. On retrouve les 4 implémentations *noreco* en bleu, *onereco* en orange, *tworeco* en vert et enfin *QoSaware* en rouge. Concernant les trois premières implémentations, on constate les mêmes tendances que pour le scénario *Wikipedia*, c'est-à-dire que l'implémentation *noreco* admet la plus petite consommation énergétique (cf. entre *110 watt* en *idle* et *150 watt*), suivie de *onereco* (cf. entre *110-170 watt*) et *tworeco* (cf. entre *110-190 watt*).

L'implémentation *QoSaware*, du fait des reconfigurations SaaS entreprises (e.g. *tactique* Figure 8.3c dans le cas de temps de réponse trop important), admet une consommation énergétique entre *110 watt* et *170 watt*, c'est-à-dire moins importante que l'implémentation *tworeco* et proche de celle de l'implémentation *onereco*. De plus, l'implémentation *QoSaware* montre les meilleurs résultats en termes de *QoS* (cf. disponibilité Figure 8.10) tout en maximisant la *QoF* (cf. Figure 8.14, essentiellement 2 recommandations calculées et retournées) lorsque le contexte d'exécution le permet (i.e. charge entrante).

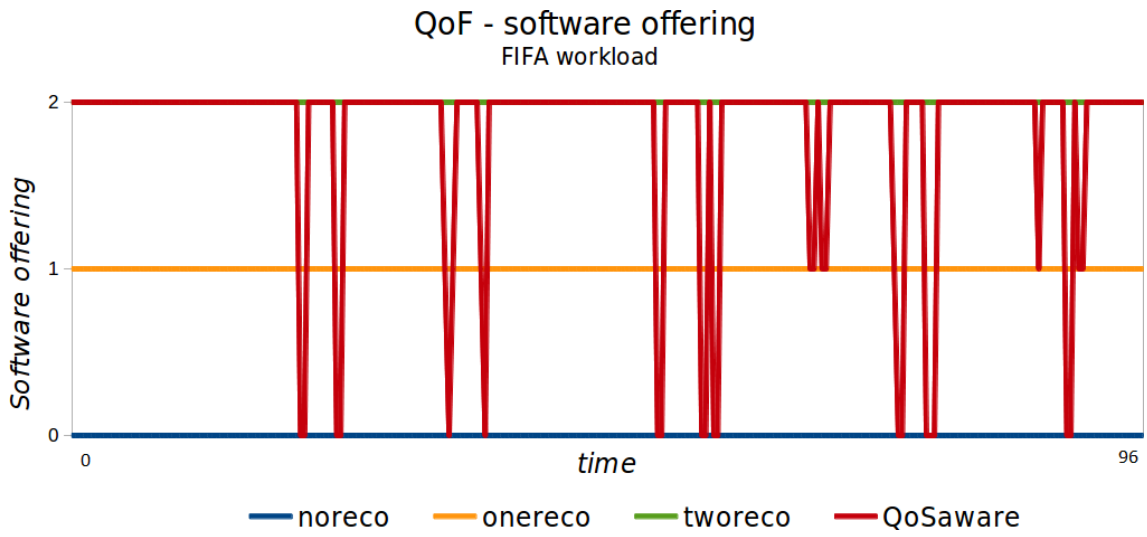


FIGURE 8.14 – *QoF* de l'application - *FIFA*.

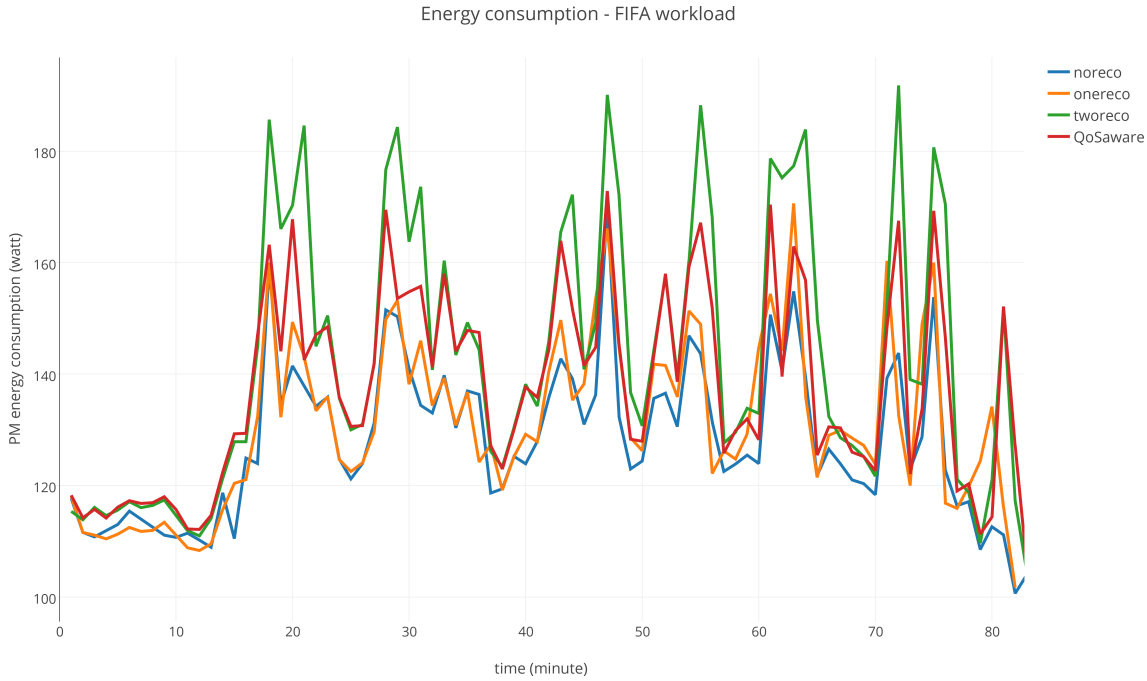


FIGURE 8.15 – Consommation énergétique - *FIFA*.

8.1.3 Discussion

Nous avons présenté dans cette section une partie des évaluations réalisées récemment en vue d'une prochaine soumission à *Cluster 2016* [rec12]. L'objectif de ces expérimentations était de porter un regard particulier sur la consommation énergétique pouvant résulter des applications SaaS ainsi que la capacité de telles applications à maintenir un niveau de *QoS* acceptable face à une demande variable (i.e. *workload*) et ce sans avoir recours à l'élasticité de l'infrastructure. En effet, la plupart des travaux s'intéressant aux problématiques énergétiques portent quasiment exclusivement sur la couche IaaS [MBS11] [SSGW11] [MBS12] (cf. Tableau 3.1 récapitulatif de l'état de l'art). Néanmoins, les résultats de nos expériences montrent que des gains sont aussi possibles sur la couche SaaS. Les Tableaux 8.1 et 8.2 offrent un récapitulatif des résultats énergétiques obtenus respectivement pour le scénario *Wikipedia* et *FIFA*.

Une première remarque est que les applications SaaS, en fonction des implémentations choisies, admettent des consommations énergétiques distinctes. En ce sens, on voit clairement apparaître que la couche SaaS joue un rôle dans la consommation énergétique globale d'un système Cloud. Nous prenons l'implémentation *tworeco* comme référentiel (i.e. mode nominal) dans les Tableaux 8.1 et 8.2. On voit apparaître des gains énergétiques de l'ordre de *13,9%* et *10,5%* (resp. *Wikipedia* et *FIFA*) entre l'implémentation *tworeco* et *noreco*. Bien que la *QoF* de cette dernière soit minimum (i.e. 0 recommandation), elle admet une *QoS* "idéale" pour les deux scénarii alors que l'implémentation *tworeco* a montré ses limites en termes de *temps de réponse* et de *disponibilité* lorsque la charge de travail était trop importante. Cependant, l'implémentation *noreco* semble inappropriée dans le contexte d'un site e-commerce réel où les recommandations jouent un rôle essentiel (i.e. augmentation des ventes et donc du profit). De la même manière, on constate des gains énergétiques supérieurs à *7%* en ce qui concerne l'implémentation *onereco* offrant une *QoF* modérée (i.e. 1 recommandation). Enfin, on observe que l'implémentation *QoSaware* présente elle aussi des gains énergétiques de l'ordre des *5%* et ce en maintenant un bon niveau de *QoS* tout en maximisant la *QoF* (i.e. essentiellement 2 recommandations).

Ces résultats viennent confirmer nos intuitions de départ (cf. chapitre 4) concernant les ajustements possibles à réaliser sur la couche *SaaS* et qui nous ont amené à considérer l'*élasticité logicielle* dans le but d'améliorer la capacité d'adaptation globale du Cloud ainsi que l'efficience de celui-ci.

Tableau 8.1 – Récapitulatif consommation énergétique des 4 expériences - *Wikipedia*.

implémentation	**total conso. expé.** (ramenée sur 24h)	**conso. moyenne**	**gain énergétique**
noreco	3117 watt	130 watt/heure	13,9%
onereco	3367 watt	140 watt/heure	7,3%
tworeco	3632 watt	151 watt/heure	-
QoSaware	3456 watt	144 watt/heure	4,7%

Tableau 8.2 – Récapitulatif consommation énergétique des 4 expériences - *FIFA*.

implémentation	**total conso. expé.** (ramenée sur 24h)	**conso. moyenne**	**gain énergétique**
noreco	3095 watt	128 watt/heure	10,5%
onereco	3152 watt	132 watt/heure	7,7%
tworeco	3416 watt	143 watt/heure	-
QoSaware	3280 watt	136 watt/heure	4,9%

Dans le cas de nos travaux, l'*élasticité logicielle* offre une alternative à l'ajout systématique de ressources pour le passage à l'échelle. Les résultats préliminaires ont en effet montré qu'il était possible de continuer de fournir le service aux utilisateurs avec un niveau de service et de performance acceptable sans nécessairement ajouter des ressources IaaS, synonyme d'augmentation des coûts (e.g. infrastructure, licences, etc.) et de l'empreinte énergétique.

Nous nous sommes concentrés uniquement sur l'*élasticité logicielle* dans le cadre de cette évaluation en considérant une infrastructure modeste (i.e. 1 unique PM pour héberger l'application *RUBiS*) et non élastique comme il en existe beaucoup. Bien que les gains énergétiques peuvent sembler faibles (e.g. *21 watt/heure* entre l'implémentation *tworeco* et *noreco* pour le scénario *Wikipedia*), il faut préciser qu'il s'agit ici des gains obtenus par heure et par PM. Imaginons que l'application *RUBiS* soit déployée pour 100 clients (e.g. *eBay France*, *eBay UK*, *eBay USA*, etc.) dont les infrastructures sont isolées (i.e. 1 PM par client). En s'appuyant sur les résultats obtenus dans le cadre du scénario *Wikipedia*, cela revient à une économie de près de *700Wh*, soit environ *17kW/jour* entre l'implémentation *tworeco* et *QoSaware* alors que cette dernière maintient un niveau de *QoS* acceptable (i.e. éviter les pénalités suite à la violation de SLA). Il faut préciser que les gains énergétiques (et économiques) liés à l'*élasticité logicielle* sont fortement dépendants de l'application mais que cette nouvelle capacité d'adaptation laisse présager une économie d'échelle importante.

8.2 Évaluations : tactiques d'élasticité

Notre article, s'intitulant *Experimental Analysis on Autonomic Strategies for Cloud Elasticity*, vise à présenter le modèle d'élasticité multi-couche introduit dans le chapitre 5. Comme son nom l'indique, il s'agit d'un article mettant en avant des expérimentations, basées sur notre modèle, en vue d'identifier des *stratégies* pertinentes de gestion de l'élasticité.

8.2.1 Protocole d'expérimentation

Nous allons décrire dans cette partie toutes les composantes de notre protocole d'expérimentation.

Environnement

Nous allons présenter ici les différents éléments de l'environnement considéré pour nos expérimentations dont la Figure 8.16 donne une vision globale de l'architecture.

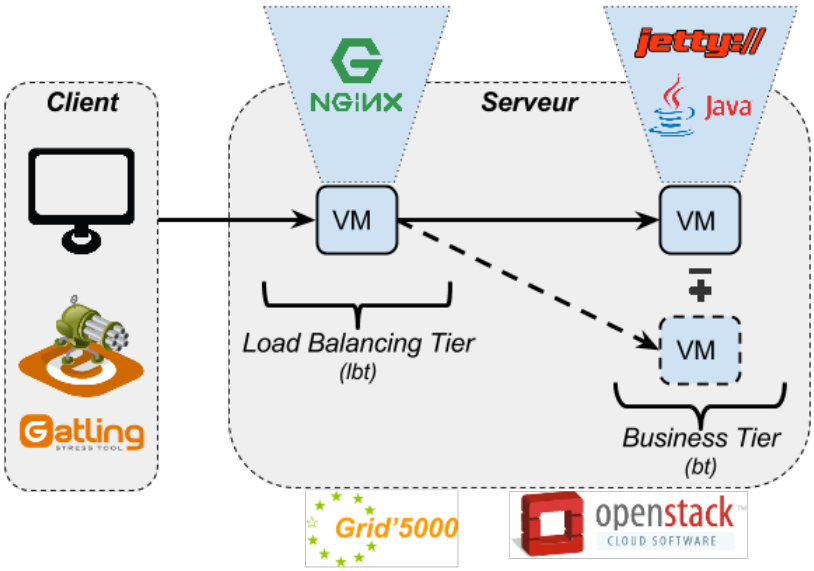


FIGURE 8.16 – Architecture globale : application et infrastructure.

Application : Nous considérons ici une application SaaS synthétique dont l'architecture admet 2 tiers distincts. Chaque tier est composé de VMs. Le premier tier (*lbt*) correspond à la répartition de charge (*load balancing*) et repose sur *Nginx* [ngi15] pour répartir la charge de travail (*workload*) aux multiples VMs qui constituent le second tier. Le second tier, appelé *business tier* (*bt*), est quant à lui constitué d'une application *Java* et d'un serveur *HTTP* reposant sur *Jetty* [jet15]. Les expériences menées visent essentiellement à surveiller et adapter le *business tier*.

L'application synthétique met en place l'*élasticité logicielle verticale* (i.e. SU_{soft} et SD_{soft}) au travers d'un composant du *business tier* admettant 5 Off_{soft} différentes se traduisant par différentes configurations de l'application, avec des niveaux de *QoF* distincts, plus ou moins consommateurs de ressources. Ces Off_{soft} sont implémentées en faisant varier graduellement le degré d'intensité de sollicitation CPU du composant en vue de simuler un traitement plus ou moins complexe. Ainsi, plus l'Off_{soft} du composant est élevée, plus le CPU sera sollicité.

Infrastructure : Notre infrastructure repose sur *Grid5000* [g5k15], une grille de calcul multi-site dédiée à la recherche. Nous avons eu recours à 7 PMs (2 CPUs Intel Xeon L5420, 4 cores/CPU, 15Go RAM, 298Go HDD) du site de Nancy (réseau : 20 Go/s Ethernet switch) dans le cadre de ces expérimentations.

La couche de virtualisation repose sur *OpenStack* [ope15b], un ensemble de logiciels *open source* permettant de déployer des infrastructures de type IaaS (*OpenStack Grizzly* installé sur une PM jouant le rôle de *cloud controller*). Dans le cadre de nos expérimentations, nous avons considéré un unique type de VM pré-configurée (i.e. Off_{infra}) proposé par *OpenStack* (i.e. *small instance* : 1CPU, 2Go RAM, 20Go Disque, ubuntu-12.04-server).

Injecteur de charge : Nous avons eu recours à l'outil *open source Gatling* [gat15] en tant qu'injecteur de charge pour simuler les requêtes *HTTP* clientes et stresser l'application.

Scénario d'évaluation

Objectifs : Les expérimentations menées dans le cadre de cet article visaient à comparer le comportement de différentes *tactiques d'élasticité* (*tactics*) face à un même scénario de charge. Ainsi, nous avons effectué 3 expériences, faisant intervenir différentes *tactiques*. Un objectif sous-jacent de ces expérimentations est que nous souhaitions obtenir des indications quant à l'identification de *critères* de comparaison entre les *tactiques d'élasticité* (cf. sous-section 5.1.3). En effet, comme nous allons le voir, certaines *tactiques*, bien répondant à un même *symptôme*, ne sont pas équivalentes et peuvent impacter de manière distincte différents *critères* comme les coûts, la QoS, etc. Ainsi, nous estimons que les résultats de ce type d'expérimentation peuvent aider l'administrateur Cloud à émettre des *préférences* quant aux choix d'adaptation du système et ainsi définir des *stratégies d'élasticité* (cf. sous-section 5.1.4).

Surveillance et symptômes considérés : Nous avons défini 2 *événements simples* (cf. sous-section 6.1.2), considérés comme des *symptômes* nécessitants une adaptation. Ces *événements simples* vont s'intéresser aux temps de réponse observés sur notre application. Dans notre cas, il s'agit de collecter à un intervalle régulier (i.e. *window* = 15 secondes) les temps de réponse du *business tier* (i.e. *événements primitifs*), d'en effectuer une moyenne que l'on va ensuite confronter à deux seuils, nommés *upper_threshold* et *lower_threshold*, qui correspondent respectivement à une valeur de 400ms et de 20ms. Les deux requêtes *CEP* sont décrites dans le Code 8.1 avec le langage *EPL* [epl15].

```
High_Response_Time : select source from
  TierResponseTimeEvent(tiername='bt').win:time(15 seconds)
    having avg(value) > 400;

Low_Response_Time : select source from
  TierResponseTimeEvent(tiername='bt').win:time(15 seconds)
    having avg(value) < 20;
```

Code 8.1 – Règles CEP : *High_Response_Time* et *Low_Response_Time*

Dans le cas du système mis en place pour ces expérimentations, les événements et les actions d'adaptation portent sur le *business tier*. La première règle *CEP* va générer des *événements simples* de type *High_Response_Time*, qui dénotent des temps de réponse trop élevés sur le *business tier* et indiquent un besoin de ressources IaaS supplémentaires ou encore le besoin de soulager les ressources IaaS existantes en réduisant l'Off_{soft} des composants logiciels sous-jacents. La seconde règle, quant à elle, va générer des *événements simples* de type *Low_Response_Time* indiquant des temps de réponse (trop) faibles et donc la possibilité de réduire la quantité de ressources IaaS (et donc les coûts) ou encore d'augmenter l'Off_{soft} de certains composants (et ainsi améliorer la *QoF* de l'application).

Expériences réalisées : Nous avons effectué 3 expériences pour lesquelles nous avons mis en place des actions d'adaptation différentes en réponse aux *symptômes* évoqués ci-dessus. Ainsi, nous avons développé 5 *tactiques d'élasticité*. Ces *tactiques* sont décrites dans la Figure 8.17 avec le formalisme de notre langage *ElaScript*. Les trois premières *tactiques* apportent une réponse au *symptôme* représenté par l'événement *High_Response_Time* tandis que les deux dernières *tactiques* répondent au problème de *Low_Response_Time*.

— **expérience 1 :** L'adaptation de la première expérience repose uniquement sur l'élasticité horizontale de l'infrastructure. Ainsi, on va appliquer respectivement un SO_{infra} (*soi*) et un SI_{infra} (*sii*) dans le cas d'un *High_Response_Time* et d'un *Low_Response_Time*. Il s'agit concrètement d'ajouter/retirer 1 VM au tier *bt* (cf. *tactiques* 8.17a et 8.17d).

— **expérience 2 :** L'adaptation de la seconde expérience repose, quant à elle, uniquement sur l'*élasticité logicielle verticale*. On va donc appliquer respectivement un SD_{soft} (*sds*) et un SU_{soft} (*sus*) dans le cas d'un *High_Response_Time* et d'un *Low_Response_Time*. Il s'agit ici de diminuer/augmenter l'Off_{soft} des composants des VMs du tier *bt* de 1 niveau (cf. *tactiques* 8.17b et 8.17e). On remarquera que l'on considère ici des *APIs* "récursives" du langage *ElaScript*, c'est-à-dire que les actions SD_{soft} et SU_{soft} appelées sur un tier vont être exécutées sur les nœuds fils (i.e. sur les composants par récursivité).

— **expérience 3 :** La dernière expérience admet une adaptation multi-couche au travers d'une *tactique* mettant en œuvre à la fois l'élasticité horizontale de l'infrastructure et l'*élasticité logicielle verticale* dans le cas d'un *High_Response_Time* (cf. *tactique* 8.17c). En revanche, un simple SI_{infra} est appliqué dans le cas d'un *Low_Response_Time* (cf. *tactique* 8.17d).

Configuration initiale : La configuration initiale de l'infrastructure et de l'application est identique pour les trois expériences réalisées. Le système, représenté par un graphe de ressources (cf. sous-section 5.2.1), est constitué d'une application pour laquelle chaque tier est constitué d'une unique VM (i.e. Off_{infra} = *small instance*) pré-configurée avec les composants associés aux tiers. L'application est paramétrée avec l'Off_{soft} maximale (i.e. 5/5), c'est-à-dire offrant le meilleur niveau de *QoF*.

Scénario de charge : Nous avons soumis notre système au même scénario de charge pour les trois expériences évoquées ci-dessus. Dans notre cas, la valeur de la charge de travail à un instant *t* est définie par le débit d'entrée du système, c'est-à-dire le nombre de requêtes/sec envoyé à l'application par les utilisateurs (i.e. injecteur de charge *Gatling*).

Le scénario appliqué pour nos expériences, d'une durée totale de 40min, se décompose en 3 phases admettant des catégories de charge de travail distinctes (cf. Figure 8.18). On retrouve le terme *workload patterns* [KYTA12] dans la littérature pour définir ces différents types de demande qui varient en termes d'amplitude et de fréquence. Dans notre cas, cela nous permet d'évaluer l'impact des différentes *tactiques* considérées selon différents types de charge entrante. Les trois phases de notre scénario sont les suivantes :

— La *Phase 1* consiste à faire grimper fortement la charge de travail pendant 10min, avec une accélération importante (*0.4req/sec²*), puis de revenir à la charge initiale en 3min ;
— La *Phase 2* revient elle aussi à effectuer une montée en charge pendant 10min puis un retour à la normale en 3min, néanmoins, la montée en charge admet une accélération deux fois moins importante que pour la première phase (*0.2req/sec²*) ;
— Enfin, la *Phase 3* correspond à 3 courts pics de charge successifs, intervenant à intervalle régulier, et admettant une accélération très importante (*1.6 req/sec²*).

8.2.2 Résultats

Métriques observées

Dans le cadre de ces expérimentations, nous avons collecté un certain nombre de métriques afin de comparer les différents *tactiques d'élasticité*. La Figure 8.19 contient les résultats obtenus pour nos trois expériences, regroupées par colonne. Comme nous l'avons évoqué précédemment, nous nous concentrons ici sur l'état des ressources et les performances du *business tier*. Nous nous sommes intéressés aux quatre métriques suivantes, constituant les quatre lignes de notre Figure 8.19 (cf. courbe rouge) :

— *Taille de l'infrastructure* : le nombre de VM utilisées par le *business tier* (cf. Figures 8.19a, 8.19b et 8.19c) ;
— *Offering de l'application* : l'Off_{soft} des composants qui constituent le *business tier* (cf. Figures 8.19d, 8.19e et 8.19f). Pour rappel, on distingue cinq Off_{soft} admettant différents niveaux de *QoF* allant de 1 (*QoF* minimum) à 5 (*QoF* maximum). Dans notre cas, les Off_{soft} de tous les composants sont homogènes (i.e. toutes les VMs du tier ont la même Off_{soft}) ;
— *Nombre de requêtes échouées* : la quantité de requêtes tombées en erreur (i.e. code de statut *HTTP* différent de 200) chaque seconde (cf. Figures 8.19g, 8.19h et 8.19i). Cette métrique donne une indication quant à la disponibilité (*availability*) du service ;
— *Temps de réponse moyen* : le temps de réponse moyen des requêtes traitées (sans erreur) par seconde (cf. Figures 8.19j, 8.19k et 8.19l).

La première expérience (cf. colonne 1 Figure 8.19) vise à ajuster uniquement la quantité de ressources IaaS en fonction de la demande en appliquant respectivement les *tactiques* 8.17a et 8.17d. Ainsi, comme on peut le voir en rouge sur la Figure 8.19d, l'Off_{soft} reste la même tout le long de l'expérience. La seconde expérience, dont les résultats figurent sur la seconde colonne, vise quant à elle à reconfigurer uniquement l'Off_{soft} en appliquant les *tactiques* 8.17b et 8.17e.

```
begin
when High_Response_Time(businessTier)
do
  // Add 1 VM to the business tier
  businessTier.soi;
end
```

(a) Tactic1 : SO_{infra}

```
begin
when High_Response_Time(businessTier)
do
  // Decrease the business tier's components' offerings of 1 level
  businessTier.sds;
end
```

(b) Tactic2 : SD_{soft}

```
begin
when High_Response_Time(businessTier)
do
  [
    // Add 1 VM to the business tier
    businessTier.soi;
  || // In parallel
    // Decrease the business tier's components' offerings of 4 levels
    businessTier.sds(4);
  ]; // Wait until the VM is started and ready to accept request
  // Switch back the components' offerings in their nominals states
  businessTier.sus(4);
end
```

(c) Tactic3 : $(SO_{infra} || SD_{soft})\,;\, SU_{soft}$

```
begin
when Low_Response_Time(businessTier)
do
  // Remove one random VM to the business tier
  businessTier.sii;
end
```

(d) Tactic4 : SI_{infra}

```
begin
when Low_Response_Time(businessTier)
do
  // Increase the business tier's components' offerings of 1 level
  businessTier.sus;
end
```

(e) Tactic5 : SU_{soft}

FIGURE 8.17 – *Tactiques d'élasticité* implémentées - Syntaxe *ElaScript*

.

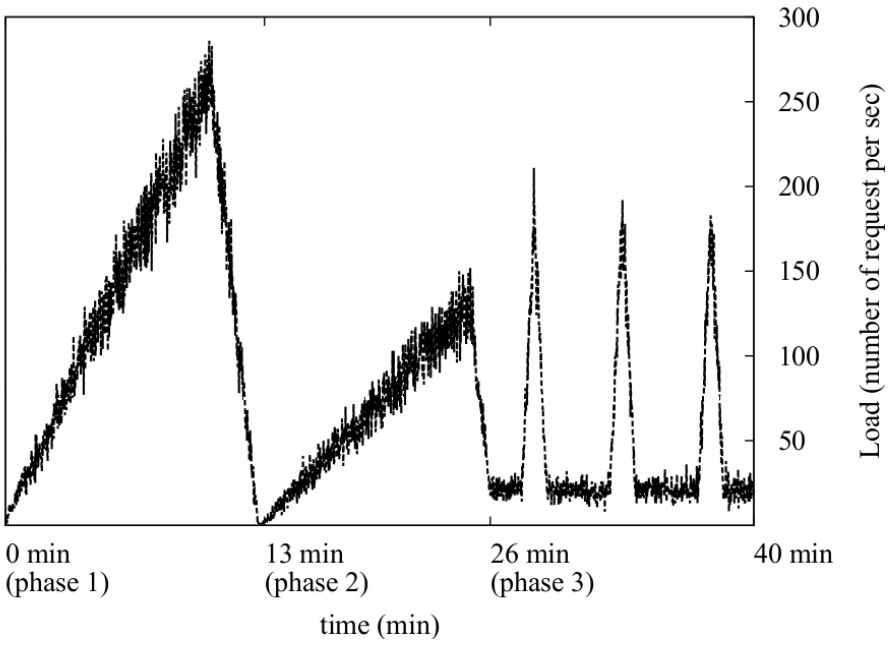


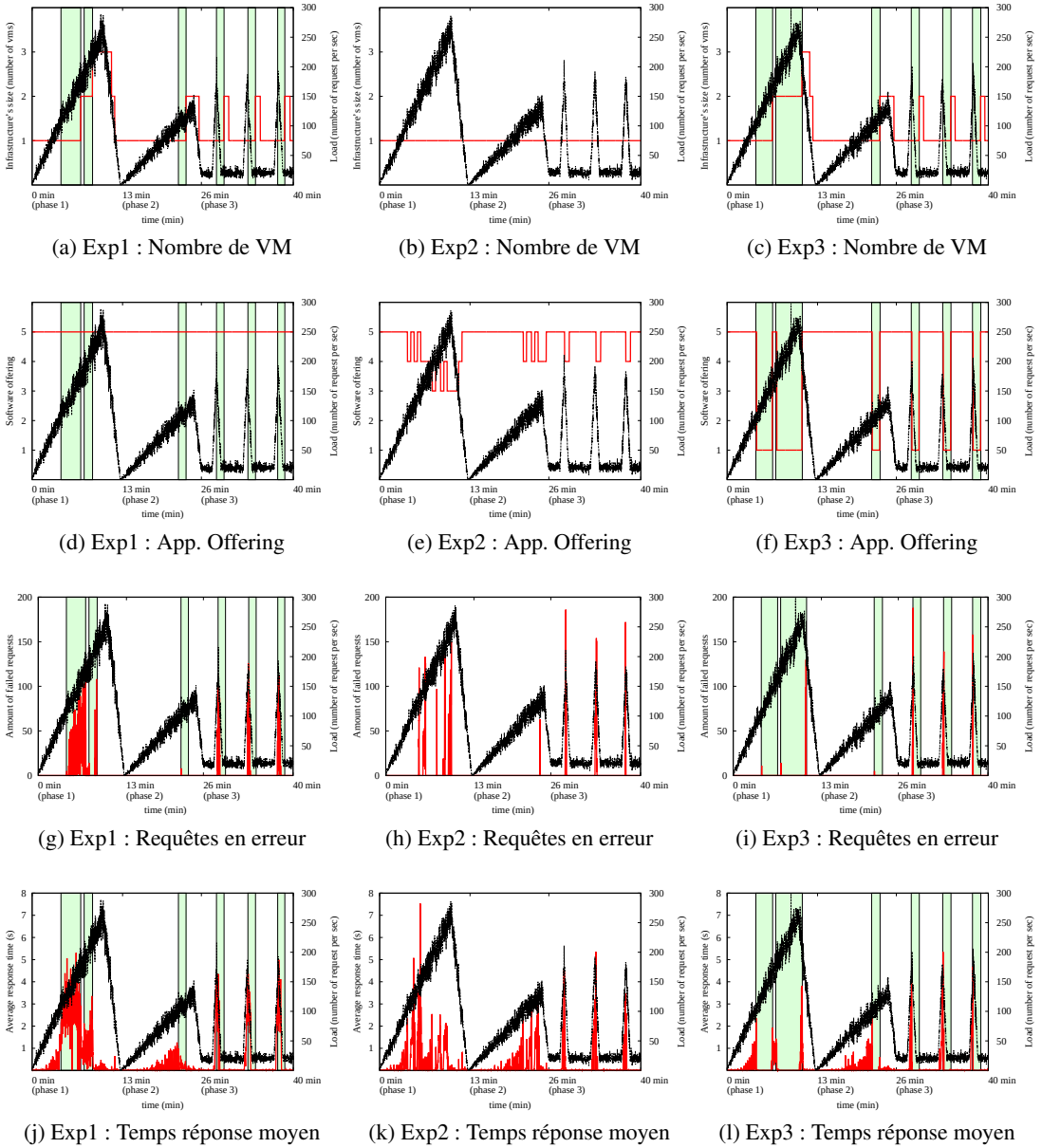
FIGURE 8.18 – Scénario de charge appliqué.

On peut ainsi constater sur la Figure 8.19b que la quantité de ressource IaaS est fixée à 1 VM pour le *business tier*. Enfin, la dernière colonne représente les résultats de la troisième expérience qui admet une adaptation multi-couche en appliquant les *tactiques* 8.17c et 8.17d.

Les courbes rouges représentent les résultats obtenus pour la métrique concernée (i.e. une métrique différente par ligne) tandis que la zone verte présente sur les figures de l'*expérience 1* et de l'*expérience 3* indique le temps de reconfiguration dans le cas d'un SO_{infra}, c'est-à-dire le temps d'initiation d'une VM. Le temps de reconfiguration des autres actions d'adaptation (e.g. SD_{soft}) n'apparait pas du fait que cette durée soit quasi instantanée (i.e. de l'ordre de la seconde). La courbe noire représente notre scénario de charge de travail qui se découpe en 3 phases. Nous allons par la suite commenter les résultats obtenus pour nos trois expériences en fonction des différentes phases du scénario.

Phase 1

La première phase du scénario correspond à une montée en charge importante. Il est possible de constater, dans le cas de la première expérience (cf. Figure 8.19a), que 2 VMs sont successivement ajoutées (cf. *t=4min* et *t=7min*) au *business tier* (i.e. *tactique* 8.17a) en réponse à cette montée en charge. Le temps de reconfiguration induit par le SO_{infra} est important, ce qui se traduit par un impact négatif sur la QoS et les performances (cf. *t=4-8min* Figures 8.19g et 8.19j). En effet, on peut observer sur la Figure 8.19g qu'un nombre important de requêtes est tombé en erreur durant la mise en place de l'adaptation (cf. zone verte). De plus, la Figure 8.19j montre que les temps de réponse augmentent de manière importante durant cette période. Néanmoins, une fois les ressources prêtes à recevoir des requêtes (i.e. VM allumée et services associés démarrés), le système retrouve un état stable en termes de QoS et de performance. En ce sens, l'élasticité horizontale de l'infrastructure mise en place pour l'*expérience 1* montre que le système peine à passer à l'échelle suffisamment rapidement dans le cas d'une augmentation rapide et importante de la charge.



(a) Exp1 : Nombre de VM (b) Exp2 : Nombre de VM (c) Exp3 : Nombre de VM

(d) Exp1 : App. Offering (e) Exp2 : App. Offering (f) Exp3 : App. Offering

(g) Exp1 : Requêtes en erreur (h) Exp2 : Requêtes en erreur (i) Exp3 : Requêtes en erreur

(j) Exp1 : Temps réponse moyen (k) Exp2 : Temps réponse moyen (l) Exp3 : Temps réponse moyen

FIGURE 8.19 – Résultats des expériences.

L'*expérience 1* montre aussi que les ressources sont rapidement libérées (i.e. SI_{infra}) lorsque la demande chute brusquement à la fin de la première phase. Il faut préciser que notre cas d'utilisation admet un serveur sans état (*stateless*) qui traite chaque requête comme une transaction indépendante et sans relation avec les requêtes précédentes. Cela permet de mettre en place un SI_{infra} sans attendre, par exemple, la fin des sessions utilisateurs courantes ou leur migration vers une autre VM. Ainsi, bien que quasi instantané dans notre cas d'utilisation, l'action SI_{infra} peut nécessiter un traitement non négligeable dans d'autres circonstances (e.g. VM contenant une base de données à répliquer).

La seconde expérience reposant sur l'*élasticité logicielle verticale*, bien qu'admettant une infrastructure limitée (i.e. 1 seule VM pour le *business tier*), offre sensiblement de meilleurs résultats en termes de QoS et de performance pour la première phase de notre scénario. Comme on peut le constater sur la Figure 8.19e, de nombreuses reconfigurations sont effectuées (i.e. 12 reconfigurations). Il s'agit ici de réduire successivement l'Off_{soft} d'un niveau en appliquant un SD_{soft} lors d'un *High_Response_Time* (et respectivement un SU_{soft} lors d'un *Low_Response_Time*).

Les résultats en termes de QoS et performance (cf. Figures 8.19h et 8.19k) indiquent qu'il serait préférable de baisser l'Off_{soft} de 2 ou 3 niveaux d'un coup lors d'un *High_Response_Time* (i.e. changer la *tactique* 8.17b) et ainsi soulager davantage les ressources IaaS sous-jacentes. Cependant, l'*expérience 2* montre que la granularité des actions d'adaptation de l'*élasticité logicielle verticale* permet de réagir rapidement à une montée en charge importante et brutale et ainsi passer à l'échelle sans nécessairement avoir recours à l'élasticité de l'infrastructure, synonyme d'augmentation des coûts.

La troisième expérience, mettant en œuvre à la fois l'élasticité horizontale de l'infrastructure et l'*élasticité logicielle verticale*, montre les meilleurs résultats en termes de QoS et de performance (cf. Figures 8.19i et 8.19l). On peut constater que contrairement à l'*expérience 1*, le fait d'exécuter un SD_{soft} en parallèle d'un SO_{infra} (cf. *tactique* 8.17c) permet de soulager les ressources IaaS le temps de l'initiation de la VM, ce qui se traduit par une indisponibilité et des temps de réponse nettement inférieurs. Cependant, on peut noter des pics concernant les temps de réponses (cf. *t=6min* et *t=11min* Figure 8.19l) et dans une moindre mesure concernant l'indisponibilité (cf. *t=11min* Figure 8.19i) lorsque l'on revient dans le mode nominal (i.e. SU_{soft}) une fois la nouvelle VM opérationnelle. Ainsi, contrairement à l'*expérience 1*, la reconfiguration mise en place dans l'*expérience 3* impacte très faiblement la QoS et les performances. Ce gain se fait au détriment de la *QoF* mais de manière ponctuelle (i.e. le temps de l'initiation de la VM).

Phase 2

La seconde phase de notre scénario, qui correspond à une montée en charge deux fois moindre que pour la première phase, offre des résultats différents. En effet, l'*expérience 1* présente cette fois les meilleurs résultats en termes de passage à l'échelle. Une seule VM est ajoutée à la fin du pic de charge (cf. *t=23min* Figure 8.19a), l'application est disponible et les temps de réponse augmentent raisonnablement jusqu'à l'ajout de la VM. En ce sens, le SO_{infra}, bien que long à mettre en place, convient dans le cas d'une montée en charge admettant une accélération lente. On peut néanmoins noter que la baisse brutale de la charge de travail (cf. *t=25min*) entraine un SI_{infra} survenant 2 minutes seulement après le SO_{infra} ce qui n'est pas idéal compte tenu de la forte consommation de ressources (i.e. CPU) induite par l'étape d'initialisation d'une nouvelle VM.

Dans le cas de l'*expérience 2*, l'adaptation survient aussi vers la fin de la montée en charge quand la demande dépasse les *100req/sec* (cf. *t=21min*). La Figure 8.19e indique que 6 reconfigurations ont eu lieu (i.e. 3 SD_{soft} et 3 SU_{soft}) faisant varier successivement l'Off_{soft} de 5 à 4. Les temps de réponse observés (cf. Figure 8.19k) augmentent davantage que pour l'*expérience 1* malgré l'exécution des SD_{soft}. De plus, un pic d'indisponibilité a lieu quand la charge atteint *140req/sec* à *t=24min*. On peut constater que malgré une charge importante et des temps de réponse qui grimpent, des actions SU_{soft} sont exécutées (suite à des SD_{soft}). Cela peut s'expliquer par des seuils trop proches dans la définition des événements (i.e. *symptômes*) *High_Response_Time* et *Low_Response_Time*. Il serait possible d'éviter ce genre d'oscillation du système en changeant par exemple la valeur des seuils concernés ou encore en admettant une courte période de calme suite à l'exécution d'un SD_{soft}.

L'*expérience 3* montre des résultats proches de l'*expérience 1* en termes de QoS et de performances (cf. Figures 8.19i et 8.19l). Les deux expériences entrainent une augmentation des coûts d'infrastructure en ajoutant une VM, néanmoins, l'adaptation multi-couche de la *tactique* 8.17c induit une baisse "inutile" de la *QoF* (i.e. Off_{soft} basculée de 5 à 1 le temps de la mise en place du SO_{infra}) ce qui n'est pas le cas de la *tactique* 8.17a utilisée dans l'*expérience 1*. En ce sens, la *tactique* 8.17a est préférable à la *tactique* 8.17c dans le cas d'une augmentation modérée de la charge.

Phase 3

La dernière phase du scénario correspond à 3 courts pics de charge successifs, intervenant à intervalle régulier (cf. *t=29min*, *t=33min* et *t=37min*), et admettant une accélération très importante. L'*expérience 1* montre que le système ne parvient pas à s'adapter à cette charge fluctuante. Le dimensionnement horizontal de l'infrastructure montre ses limites en termes de réactivité. En effet, on peut constater sur la Figure 8.19a que 3 SO_{infra} sont déclenchés (cf. zones vertes) lorsque les pics de charge atteignent leur maximum (i.e. *200req/sec*), cependant, les VMs ne sont opérationnelles qu'une fois le pic de charge passé. Cela se traduit instantanément par une baisse considérable de la QoS et des performances. De plus, chaque SO_{infra} est directement suivi d'un SI_{infra} ce qui revient à arrêter les ressources sans même les avoir sollicitées (i.e. effet *ping-pong*). Le manque de réactivité du SO_{infra} ne permet pas de traiter des pics de charge aussi bref.

La seconde expérience, s'appuyant sur le dimensionnement vertical du logiciel, permet d'être plus réactif au changement rapide de la charge, néanmoins, le fait de baisser l'Off_{soft} d'un niveau seulement dans le cas du SD_{soft} (i.e. de 5 à 4) ne permet pas de soulager suffisamment les ressources IaaS sous-jacentes ce qui se traduit là-aussi par une baisse de la disponibilité et une augmentation des temps de réponse. Face à de tels pics de charge, il serait préférable de modifier la *tactique* 8.17b ou d'en créer une nouvelle qui viserait à baisser l'Off_{soft} de plusieurs niveaux d'un coup. Une autre possibilité serait de définir un nouvel événement (i.e. *symptôme*) correspondant à un pic brutal (e.g. portant sur l'accélération de la demande) ou de réduire la fenêtre d'agrégation de l'événement *High_Response_Time* (e.g. en passant la fenêtre d'observation de *15sec* à *10sec*).

La troisième expérience souffre des mêmes travers que l'*expérience 1* en ce qui concerne le manque de réactivité de l'action SO_{infra}. En ce sens, aucune des *tactiques* 8.17a, 8.17b et 8.17c ne semblent convenir dans le cas de brefs pics de charge. Néanmoins, contrairement aux deux autres *tactiques*, la *tactique* 8.17b n'induit pas une augmentation des coûts d'infrastructure ce qui la rend préférable bien que non idéale. Afin de rendre l'adaptation de cette *tactique* moins dommageable en termes de QoS et de performance, il serait bénéfique de diminuer de plusieurs niveaux l'Off_{soft} en modifiant la *tactique* existante ou en créant une nouvelle *tactique* (éventuellement associée à un nouvel événement (i.e. *symptôme*) s'intéressant à l'accélération de la demande).

8.2.3 Discussion

Les résultats des expériences montrent que certaines *tactiques d'élasticité*, bien qu'elles aient les mêmes intentions (e.g. réponse à un manque de ressources) ou qu'elles tendent à adresser un même *symptôme* (e.g. *High_Response_Time*), ne sont pas pour autant équivalentes. En effet, nous avons notamment pu observer que certaines des *tactiques* passées au banc d'essai convenaient davantage à certains types de charge de travail. De plus, les différentes *tactiques* admettent des résultats différents en termes de QoS, de *QoF*, de coûts ou encore de réactivité d'adaptation.

Un des objectifs de ce travail était d'identifier un certain nombre de *critères* (cf. sous-section 5.1.3) permettant de positionner différentes *tactiques d'élasticité* selon plusieurs aspects (i.e. coût, QoS, réactivité d'adaptation, etc.). Les expériences réalisées ont non seulement permis d'éprouver un ensemble de *tactiques* en les confrontant à différents types de charge de travail, mais aussi d'identifier les avantages et les inconvénients de celles-ci. Ce type d'expérience, qui s'apparente au processus de calibrage effectué dans le monde industriel, constitue un mal nécessaire pour la mise en place d'un processus de dimensionnement automatique efficace. En effet, le résultat de ces expériences donne à l'administrateur humain des informations essentielles quant à l'évolution du système face à un environnement dynamique et accroît ainsi sa compréhension du système. Il lui devient alors possible de transmettre cette connaissance au gestionnaire autonomique sous forme d'intention.

8.2.4 Vers des stratégies d'élasticité

L'objectif est d'offrir à l'administrateur humain un moyen de paramétrer sa politique d'élasticité sous forme de *stratégies* qui vont orienter la prise de décision du gestionnaire autonomique en préférant une adaptation (i.e. *tactique*) plutôt qu'une autre.

Le Tableau 8.3 vise à positionner les 3 *tactiques* 8.17a, 8.17b et 8.17c selon les 6 critères décrits dans la sous-section 5.1.3. Les 3 *tactiques* peuvent être définies de "concurrentes" du fait qu'elles répondent toutes au même *symptôme*, représenté par l'*événement simple High_Response_Time*, qui dénote des temps de réponse trop élevés et indique un besoin de ressources IaaS supplémentaires ou encore le besoin de soulager les ressources IaaS existantes en réduisant l'Off_{soft} des composants logiciels sous-jacents. Dans le tableau, les valeurs "+" et "-" indiquent que la *tactique* a respectivement une influence positive et une influence négative sur le *critère* considéré tandis que la valeur "0" dénote d'une influence à la fois positive et négative.

Le tableau indique par exemple que la *Tactic1* et la *Tactic3*, du fait de la mise en place d'une action de type SO_{infra}, ont un impact négatif sur les *critères* de coûts et de consommation énergétique mais vont permettre d'améliorer (i.e. impact positif) la QoS. De plus, la *Tactic1* ne montre pas de bons résultats en termes de réactivité, qui s'explique par le temps d'exécution du SO_{infra} (i.e. temps nécessaire au processus d'initialisation de la VM) contrairement à la *Tactic2* qui permet de soulager rapidement les ressources en réduisant l'Off_{soft} (i.e. au détriment de la *QoF*). La *Tactic3*, qui réduit l'Off_{soft} de manière temporaire (i.e. le temps du SO_{infra}) a un impact négatif puis positif sur le *critère* de *QoF* (cf. "0") mais bien que mettant en œuvre un SO_{infra} (i.e. temps de reconfiguration important, cf. "-"), la réactivité de cette *tactique* reste bonne du fait que l'adaptation permet de rapidement adresser le *symptôme* (i.e. *événement simple High_Response_Time*), en soulageant les ressources IaaS sous-jacentes par un SD_{soft}.

Tableau 8.3 – Positionnement des *tactiques d'élasticité* selon 6 *critères* : *High_Response_Time*.

Tactique considérée	Coût	QoS	QoF	Énergie	Temps Reconf.	Réactivité
SO_{infra} - Tactic1	-	+	+	-	-	-
SD_{soft} - Tactic2	+	+	-	+	+	+
$(SO_{infra} \| \| SD_{soft}); SU_{soft}$ - Tactic3	-	+	0	-	-	+

Le résultat des tests de calibrage va permettre à l'administrateur humain de définir les *critères* d'évaluation des *tactiques* et de positionner celles-ci selon chaque *critère* en émettant des *préférences*. Ces informations vont ensuite peupler la base de connaissance du gestionnaire autonomique (i.e. *Knowledge* de *MAPE-K*). L'administrateur doit pouvoir guider le gestionnaire autonomique en lui fournissant ses intentions sous forme de *stratégies d'élasticité* (cf. sous-section 5.1.4). Il s'agit concrètement d'émettre des *préférences d'élasticité* qui vont orienter la prise de décision du gestionnaire autonomique, c'est-à-dire le choix de l'adaptation à mettre en place dans le cas de *tactiques* "concurrentes" qui répondent au même *symptôme* (cf. section 7.4).

Intuitivement, un administrateur humain voudrait pouvoir choisir la *tactique idéale* qui revient à minimiser les coûts et la consommation énergétique tout en maximisant la QoS, la *QoF*, le temps de reconfiguration et la réactivité (i.e. atteindre l'objectif pour tous les *critères*). Néanmoins, l'existence d'une telle *tactique* relève de l'utopie du fait des nombreux compromis induits par l'élasticité multi-couche. Une *stratégie d'élasticité* revient alors à privilégier certains *critères* par rapport à d'autres. Il s'agit pour l'administrateur humain d'ordonner les *critères* de manière décroissante (i.e. méthode lexicographique, cf. sous-section 5.1.4), du plus important (i.e. considérable et décisif) au moins important (i.e. négligeable).

Nous donnons ci-dessous 3 exemples de *stratégies d'élasticité*, que nous appelons *Cost_First*, *Reactivity_First* et *QoF_First* :

— *Cost_First* : Coût>Énergie>QoS>TempsReconf>Réactivité>QoF ;
— *Reactivity_First* : Réactivité>QoF>QoS>TempsReconf>Coût>Énergie ;
— *QoF_First* : QoF>Coût>QoS>Réactivité>TempsReconf>Énergie.

Les trois *stratégies d'élasticité* évoquées ci-dessus vont orienter distinctement le choix de la *tactique* qui va être exécutée par le gestionnaire autonomique (cf. Tableau 8.4). En effet, la *stratégie Cost_First*, qui vise à minimiser les coûts d'infrastructure, va directement privilégier le choix de la *Tactic2* du fait qu'elle soit la seule à ne pas augmenter les coûts. La stratégie *Reactivity_First* va tout d'abord éliminer la *Tactic1* qui admet un impact négatif sur le *critère réactivité*. Celle-ci va ensuite éliminer la *Tactic2* qui vise à dégrader de manière permanente la *QoF* et ainsi préférer la *Tactic3* qui va dégrader que temporairement la *QoF*. Enfin, la *stratégie QoF_First* va directement choisir la *Tactic1* qui est la seule à ne pas impacter négativement la *QoF*.

Tableau 8.4 – Résultat des *stratégies d'élasticité* sur le choix des *tactiques*.

Stratégie d'élasticité	**Tactique exécutée**
Cost_First	*Tactic2* 8.17b
Reactivity_First	*Tactic3* 8.17c
QoF_First	*Tactic1* 8.17a

Les *stratégies d'élasticité* offre à l'administrateur humain un moyen à la fois simple et efficace de définir ses préférences quant aux adaptations à entreprendre sur le système en définissant aisément ses intentions au gestionnaire autonomique afin de d'orienter la prise de décision en cas de *tactiques* "concurrentes". En ce sens, le fait de positionner les *tactiques* selon certains *critères* et de définir des *stratégies d'élasticité* permet d'atteindre le 6^{eme} commandement indispensable pour un système informatique autonomique selon *Paul Horne* qui consiste à comprendre les intentions humaines.

8.3 Passage à l'échelle de la solution

Dans cette section, nous portons une réflexion sur le passage à l'échelle de notre solution de gestion autonomique de l'élasticité multi-couche. Bien que nous n'ayons pas réalisé de validation quantitative de la solution à proprement parler, nous allons présenter différents éléments qui laissent à penser que la solution passe à l'échelle. De plus, nous verrons que la solution permet, d'un point de vue conceptuel et technique, d'être paramétrée à des fins de passage à l'échelle.

Système administré

Un premier point essentiel concerne la taille du système dont on attend un comportement autonomique (i.e. graphe de ressources). L'envergure du système considéré a bien entendu un impact sur l'*"overhead"* du gestionnaire autonomique aussi bien pour les phases de *surveillance* (e.g. nombre d'*événements/symptômes* remontés), de *décision* (e.g. nombre de contraintes *runtime*) et *d'adaptation* (e.g. nombre de ressources impactées par une *tactique d'élasticité*). De même, la volumétrie de la base de connaissances du gestionnaire va être liée à la taille du système.

Il faut préciser que les solutions logicielles proposées par *Sigma* à ses clients reposent sur des architectures de "taille modeste" (i.e. de l'ordre de la dizaine de VMs) et n'adressent pas des centaines de milliers d'utilisateurs par seconde comme ça peut être le cas du réseau social *Facebook* qui comptait plus d'un milliard d'utilisateurs actifs par jour en septembre 2015 [fac16] et dont l'architecture dépasse l'entendement.

Bien que le contexte de l'entreprise *Sigma* induit des systèmes de "taille modeste", nous allons voir que notre solution de gestion autonomique de l'élasticité multi-couche a été conceptuellement prévue pour adresser des systèmes de taille variable.

Prise de décision régulée par l'administrateur

Notre solution, notamment au travers du paramétrage du processus de gestion de l'élasticité réalisé en amont par l'administrateur, permet de limiter considérablement la complexité de la prise de décision du gestionnaire autonomique.

Pour ce qui est de l'adaptation du système, le nombre d'actions d'élasticité est borné et connu à l'avance (cf. sous-section 5.1.2). De plus, le nombre de *tactiques d'élasticité* définies par l'administrateur face à un *symptôme* (i.e. *tactiques éligibles*) reste limité (e.g. 3 ou 4 *tactiques*). En ce sens, la première étape du processus de décision, à savoir le *tactics filter* (cf. section 7.2), limite véritablement le champ des possibles ce qui va réduire considérablement la complexité de la prise de décision. En effet, il s'agit ici de considérer uniquement les *tactiques éligibles* comme adaptations candidates et non pas de calculer une configuration cible "idéale" à partir du contexte d'exécution.

Le *tactics filter*, en limitant dès le départ les choix d'adaptation, permet d'assurer une complexité raisonnable pour le reste du processus de décision. Ainsi, la seconde étape de celui-ci représentée par le *context filter* (cf. section 7.3) va revenir à évaluer des contraintes *runtime* sur un ensemble fini de *tactiques* relativement petit.

Enfin, la dernière étape du *processus* de décision, à savoir le *strategy filter* (cf. section 7.4), va aussi bénéficier du nombre limité de *tactiques* à évaluer notamment pour le calcul des *scores* réalisé, permettant d'ordonner les *tactiques applicables* issues du *context filter* selon les préférences de l'administrateur définies sous forme de *stratégie d'élasticité*. De plus, le traitement réalisé par ce dernier filtre, c'est-à-dire l'algorithme de décision multi-critère basé sur une méthode lexicographique (cf. sous-section 5.1.4), est facilité par le nombre limité de *tactiques applicables* et de *critères d'adaptation* considérés.

Paramétrisation de la surveillance

En ce qui concerne la *surveillance* (i.e. *monitoring*) de notre solution, le nombre de métriques considérées et d'*événements* qui en résultent constitue un ensemble borné. De cet ensemble va découler un nombre raisonnable de *symptômes*, signes d'instabilité du système pouvant nécessiter une adaptation. Notre *framework* de gestion autonomique de l'élasticité multi-couche intègre certains mécanismes permettant de "réguler" l'*overhead* de la solution. Tout d'abord, notre modèle de *surveillance* et l'outil *perCEPtion* permet de paramétrer l'observation du système en spécifiant les événements intéressants à prendre en considération dans l'adaptation du système sous forme de requêtes *CEP* (cf. section 6.1). Il est alors possible de s'intéresser à un sous-ensemble de métriques dont l'évolution est porteuse de sens (e.g. consommation CPU pour un tier *CPU-intensive*) pour la reconfiguration du système.

Nous avons évoqué que l'interface entre les phases *M* et *A* du gestionnaire autonomique *MAPE-K* était mise en place au travers de la *symptoms queue*. Pour rappel, les éléments (i.e. *symptômes*) de la queue sont purgés en tenant compte de leur *durée de vie*. Chaque *symptôme*, une fois généré et ajouté à la queue, se voit attribuer une date de *péremption*. Il s'agit alors de purger de la queue tous les éléments ayant atteint leur date de *péremption*. Cette étape de *purge* est effectuée avant chaque ajout (i.e. par le *M*) et retrait (i.e. par le *A*) dans la queue. Outre le fait d'éviter de traiter des *symptômes* obsolètes, cette purge offre aussi un moyen d'éviter une empreinte trop importante de *perCEPtion* qui s'appuie sur le moteur *CEP Esper* [esp15b].

L'outil *Esper* a fait l'objet de tests de performance excédant largement nos besoins. Le paragraphe qui suit donne un aperçu des résultats des tests mis à disposition dans la documentation d'*Esper* concernant l'*overhead* de l'outil [esp16b].

> « *Esper exceeds over 500000 event/s on a dual CPU 2GHz Intel based hardware, with engine latency below 3 microseconds average with 1000 statements registered in the system - this tops at 70 Mbit/s at 85% CPU usage. Esper also demonstrates linear scalability from 100000 to 500000 event/s on this hardware, with consistent results accross different statements.* »
> (*Esper* documentation - 2016)

En ce qui concerne notre implémentation de la *symptoms queue* (i.e. *PriorityQueue<Symptom>*), on peut imaginer fixer une taille limite (e.g. 50 éléments) permettant par exemple de réduire l'empreinte RAM/CPU de celle-ci. Cela peut être réalisé sans impacter la capacité d'adaptation du système du fait que la queue est triée en fonction des priorités des *symptômes*. Ainsi, les éléments lointains dans la queue ne seront probablement jamais défilés (i.e. purgés avant). De la même manière, on peut jouer sur la taille des *Event Stream* ou d'autres paramètres en vue de limiter l'utilisation de ressources de l'outil *Esper* via sa configuration [esp16a].

Accroître la capacité d'adaptation du système par la parallélisation

Nous nous sommes penchés dans la section 7.5 sur la possibilité de paralléliser plusieurs processus de décision ce qui revient à traiter plusieurs *symptômes* en parallèle. Nous avons évoqué qu'un objectif de cette parallélisation était de rendre le système davantage réactif au contexte d'exécution dynamique dans lequel il évolue, en permettant notamment de mener plusieurs adaptations en parallèle. Nous nous sommes alors intéressés à fournir des pistes exploratoires quant à la mise en place d'une telle parallélisation. Cependant, nous n'avons pas évoqué l'intérêt de la parallélisation en termes de passage à l'échelle. En effet, cette solution pourrait permettre d'améliorer considérablement la réactivité de l'adaptation tout en considérant des systèmes de tailles variables. La mise en place d'une telle parallélisation permet de considérer à la fois le passage à l'échelle du système auto-administré mais aussi du gestionnaire autonomique lui-même.

9

Conclusion et perspectives

Dans ce chapitre, nous dressons le bilan des travaux réalisés dans le cadre de cette thèse. Nous présentons, dans un premier, un rappel des motivations qui nous ont amené à étendre le concept d'élasticité aux couches hautes du Cloud ainsi qu'une synthèse des contributions de cette thèse (cf. section 9.1). Enfin, nous dressons les amélioration envisageables des travaux actuels puis nous développons les perspectives futures (cf. section 9.2).

Contents

9.1 Conclusion

9.1.1 Contexte et problématique

Le Cloud computing révolutionne complètement la façon de gérer les ressources informatiques. Grâce à l'élasticité, les ressources peuvent être provisionnées en quelques minutes pour satisfaire un niveau de qualité de service formalisé par un contrat de niveau de service entre les différents acteurs (fournisseur et consommateurs). Le principal défi pour le fournisseur de services est de maintenir la satisfaction de ses consommateurs en assumant les contrats signés tout en minimisant le coût de ses services.

L'élasticité de l'infrastructure (*IaaS - Infrastructure as a Service*) est une caractéristique cruciale pour les fournisseurs SaaS (*Software as a Service*) dont les applications évoluent dans des environnements hautement variables. Il s'agit de répondre aux demandes des utilisateurs en temps normal, mais aussi dans les périodes de forte activité ou encore dans le cas d'événements imprédictibles bien que récurrents : pannes logicielles, matérielles, coupures de courant, etc. Le système doit être capable de supporter une montée en charge puis un retour à la normale, sans interruption de service, et si possible, sans répercussions sur le niveau de service.

Malgré les avantages indéniables de l'élasticité IaaS, le modèle actuel du Cloud est sujet à certaines limitations (physiques, conceptuelles et techniques) empêchant d'apprécier véritablement son potentiel théorique. Premièrement, les ressources sont limitées, ce qui signifie que l'on ne peut pas passer à l'échelle de manière infinie. Deuxièmement, le temps d'initialisation des ressources IaaS n'est pas négligeable ce qui induit parfois un manque de réactivité face aux changements d'états du système (e.g. pic de charge brutal). Troisièmement, les possibilités d'adaptation offertes ainsi que le modèle de facturation à l'heure actuel admettent une granularité trop importante se traduisant par un gaspillage financier pour le fournisseur et/ou le consommateur. Enfin, malgré les récents efforts pour améliorer l'efficacité énergétique du matériel et plus généralement des centres de données, l'évolution de l'impact énergétique des nouvelles technologies de l'information reste préoccupante.

Bien que de nombreux travaux scientifiques se soient intéressés à l'élasticité du Cloud en vue d'améliorer la capacité d'adaptation des systèmes, la pratique montre que les solutions industrielles souffrent aujourd'hui d'un manque d'outillage. En effet, les solutions reposent aujourd'hui sur des services de dimensionnement automatique basiques qui n'ont que très peu évolués depuis l'avènement du Cloud. On constate alors un manque d'adhérence entre les travaux scientifiques et les solutions industrielles du fait que les propositions scientifiques sont souvent trop complexes à mettre en œuvre dans un contexte industriel et généralement non outillées ce qui empêche leur industrialisation. En ce sens, la pauvreté des solutions d'élasticité industrielles actuelles laisse présager un besoin crucial d'équiper les administrateurs Cloud d'outils leur permettant de paramétrer le processus de gestion de l'élasticité de bout en bout et ainsi jouir pleinement des bienfaits de celle-ci.

9.1.2 Contributions

L'objectif général de cette thèse est de proposer un nouveau modèle d'*élasticité multi-couche* qui considère l'adaptation des ressources Cloud au sens large afin de pallier aux limites du modèle d'élasticité actuel (i.e. adaptation des ressources IaaS). Nous avons ainsi proposé le nouveau concept d'*élasticité logicielle* qui vise à adapter les ressources *SaaS* (i.e. composants logiciels). Dans le contexte industriel de cette thèse, nous avons entrepris d'outiller ce nouveau modèle d'*élasticité multi-couche* afin d'offrir aux administrateurs la possibilité de paramétrer le processus de gestion de l'élasticité qui en découle. Cette thèse propose les contributions suivantes :

1. *un modèle conceptuel de l'élasticité multi-couche* : nous avons défini et modélisé l'*élasticité logicielle* ainsi que les différents concepts relatifs à cette nouvelle capacité d'adaptation. Nous avons introduit un modèle de ressources Cloud qui consiste en une représentation du système sous forme de graphe de ressources. Enfin, nous avons proposé une modélisation de l'*élasticité multi-couche* qui revient à considérer les deux types d'élasticité (i.e. IaaS et SaaS) dans un même processus d'adaptation automatique.

2. *un modèle de surveillance outillé* : nous avons proposé un modèle conceptuel de surveillance reposant sur le traitement des événements complexes dans le but de définir une observation avancée des ressources Cloud au sens large. Une implémentation de ce modèle a été proposée au travers de l'outil *perCEPtion* qui permet à l'administrateur Cloud de manipuler le modèle de surveillance en définissant en amont les *symptômes*, signes d'incohérence, qui seront identifiés à l'exécution et pourront faire l'objet d'une adaptation du système.

3. *un modèle d'adaptation outillé* : nous avons présenté un modèle d'adaptation qui repose sur la définition et l'exécution de *tactiques d'élasticité*, désignant un plan de reconfiguration générique spécifiant l'adaptation à réaliser face à un type de *symptôme*. Afin de décrire efficacement de tels plans, potentiellement multi-couche, nous avons proposé *ElaScript*. *ElaScript* est un langage dédié à l'*élasticité multi-couche* permettant à l'administrateur Cloud de définir simplement et de manière concise des plans de reconfiguration complexes sous forme de *scripts*. Ce langage offre la possibilité de composer et d'orchestrer les multiples actions d'élasticité via des opérateurs de séquentialité, parallélisme et synchronisation.

4. *un modèle de décision d'adaptation* : nous avons proposé un modèle chargé de la prise de décision concernant les changements à apporter au système. Celui-ci prend la forme d'un processus constitué de trois étapes, représentées sous forme de *filtres* successifs et dont le point d'entrée est un *symptôme* remonté à l'exécution par notre outil *perCEPtion*. La décision revient à choisir la meilleure adaptation possible parmi les différentes *tactiques d'élasticité* définies au préalable par l'administrateur humain avec le langage *ElaScript*. Notre modèle de décision est paramétrable et offre la possibilité à l'administrateur d'émettre ses préférences d'adaptation sous forme de *stratégie d'élasticité*. Le processus de décision tient compte du contexte d'exécution courant du système et des contraintes associées ainsi que des préférences de l'administrateur en vue d'identifier la meilleure adaptation à entreprendre sur le système.

5. *un framework autonomique de gestion de l'élasticité multi-couche* : nous avons proposé le *framework ElaStuff*, une solution complète pour la gestion de l'*élasticité multi-couche*. Il s'agit d'un gestionnaire autonomique de type *MAPE-K* permettant d'assurer le processus de gestion de l'*élasticité multi-couche* de bout en bout, depuis son paramétrage par l'administrateur Cloud jusqu'à son exécution. Ce *framework* autonomique s'appuie sur les différents modèles évoqués ci-dessus ainsi que sur notre outil de surveillance *perCEPtion* et notre langage dédié *ElaScript*.

9.2 Perspectives

Les travaux réalisés dans le cadre de cette thèse laissent présager des améliorations possibles ainsi que certaines extensions.

Améliorer le processus de décision : le *mapping symptôme-adaptation* sur lequel s'appuie le *Tactics Filter* pourrait être généré automatiquement à partir des fichiers *".elas"* contenant les *tactiques d'élasticité*. Le *Context Filter*, visant à contrôler les contraintes *runtime* limitant les possibilités d'adaptation, pourrait s'appuyer sur les concepts de la *programmation par contrat* [Mey92] à travers une implémentation des contraintes avec le langage *OCL* [ocl15]. Enfin, il est possible d'améliorer le calcul des *scores* des *tactiques* du *Strategy Filter* en fonction des *critères* d'adaptation en s'appuyant sur des fonctions de calcul et un calibrage préalable (cf. sous-section 7.4.5).

Paralléliser les adaptations du système : une extension possible du *framework ElaStuff* serait de pouvoir paralléliser plusieurs processus de reconfiguration du système afin d'accroître sa capacité d'adaptation face à un environnement hautement dynamique. Cette exécution parallèle nécessite de mettre en place une coordination et une synchronisation des adaptations afin d'éviter les reconfigurations incohérentes, contreproductives ou encore interblocantes. Une piste possible serait de considérer les ressources remontées par les *symptômes* de notre outil de surveillance *perCEPtion* ainsi que les *tactiques* définies au préalable, puis de s'appuyer sur la représentation de graphe du système pour identifier les potentiels risques de conflits (e.g. intersection de sous-graphes). Cette extension visant à paralléliser plusieurs processus de reconfiguration a été traitée dans la section 7.5.

Étendre l'adaptation du Cloud : dans cette thèse, nous proposons d'étendre l'élasticité aux couches hautes du Cloud, et plus précisément à la couche SaaS, en considérant les ressources qui la constituent à savoir les composants logiciels. Cela se traduit par de nouvelles possibilités de reconfiguration faisant intervenir de nouvelles dimensions d'adaptation. Cependant, nous ne prétendons pas être exhaustif dans les adaptations possibles offertes par un environnement Cloud. En effet, une évolution possible serait de considérer d'autres types de ressources comme les *conteneurs logiciels* (e.g. *Docker* [DOC15]), que l'on peut associer à la couche *PaaS*, et qui admettent des propriétés intéressantes qui permettraient notamment d'accroître la réactivité du passage à l'échelle (par rapport à une VM). De même, il serait possible d'intégrer de nouvelles actions d'adaptation comme la migration de VM ou l'utilisation de *DVFS* [VLWYH09] qui exploite la capacité des processeurs modernes à opérer selon plusieurs fréquences CPU ou niveaux de voltage.

Proposer un modèle de type *Elasticity as a Service* : nous avons montré dans cette thèse que le fait de considérer l'élasticité multi-couche dans un processus global de gestion autonomique offrait des résultats encourageants. Bien que nous nous soyons concentrés sur l'*élasticité logicielle*, nous avons évoqué que de nombreux autres mécanismes pouvaient contribuer à améliorer la capacité d'adaptation d'un système Cloud. De même, nous avons ouvert la voie vers le fait de considérer tout type de ressources Cloud comme variables d'ajustement face à l'environnement dynamique dans lequel elles évoluent. De ce fait, une possibilité d'extension plus générale serait de proposer un modèle de fournisseur d'élasticité s'apparentant à une offre PaaS et permettant de configurer et piloter l'adaptation des ressources et des services du Cloud au sens large. Cela constituerait un service de gestion autonomique de l'élasticité (i.e. *Elasticity as a Service*) à la fois multi-couche et multi-nuage à destination des différents fournisseurs de Cloud (i.e. *XaaS - Anything as a Service*). Dans cette voie, on pourrait s'appuyer sur les travaux autour de la modélisation à l'exécution (i.e. *model@runtime* [BBF09] [Fou13]) pour constituer une représentation simplifiée d'un système Cloud réel (i.e. abstraction), qui en contient les aspects fondamentaux (i.e. architecture, contexte d'exécution, etc.) tout en permettant d'interagir avec celui-ci.

Modéliser et gérer la variabilité du Cloud : bien que n'apparaissant pas dans ce manuscrit, nous avons abordé les travaux autour de la variabilité qui a été fortement étudiée dans le domaine des lignes de produits logiciels [PBvDL05]. Dans ce domaine, les variations du système sont le résultat d'exigences fonctionnelles distinctes des utilisateurs. Quelques travaux se sont intéressés à la modélisation et la gestion de la variabilité dans un contexte Cloud en partant du principe que la variabilité d'un tel système, induite par les différentes configurations possibles, nécessite d'être modélisée afin de faciliter sa manipulation et son partage entre les différents acteurs du Cloud. Certains travaux se sont appliqués à intégrer la dimension non-fonctionnelle (i.e. performances, QoS, etc.) aux modèles de variabilités existants comme c'est le cas de [SMM+12] qui s'intéresse à une configuration transverse des différentes couches du Cloud. De même, [GGRTRC13] s'appuie sur les *feature model* [KCH+90] dans le but de modéliser la variabilité IaaS des différentes offres de machines virtuelles de *Amazon Web Service* [AWS16]. Enfin, [HHPS08] s'intéresse à la notion de variabilité dans le contexte dynamique d'applications SaaS déployées sur le Cloud.

Le besoin de modélisation de la variabilité prend tout son sens dans le contexte de notre travail autour de l'élasticité multi-couche pour laquelle les différents types de ressources concernés peuvent faire l'objet de configurations diverses et variées (e.g. Off_{infra}, Off_{soft}) admettant des résultats distincts en termes de coût, de QoS, de *QoF*, etc. Dans le cas de l'*élasticité logicielle*, il serait intéressant de s'appuyer sur une telle modélisation de la variabilité dans le but de générer automatiquement et dynamiquement les différentes variantes de l'application SaaS (i.e. Off_{soft}). Dans cette voie, [JLS10] propose une solution basée sur l'ingénierie dirigée par les modèles [Sch06] permettant de gérer la variabilité en générant automatiquement différentes variantes de l'application SaaS. Cependant, la variabilité n'est pas résolue de manière dynamique et consiste à déployer plusieurs instances de l'application en amont. Nous proposons donc de poursuivre les efforts amorcés dans cette voie en considérant une *résolution dynamique de la variabilité*.

Bibliographie

[AB14] Aakash Ahmad and Muhammad Ali Babar. Towards a pattern language for self-adaptation of cloud-based architectures. In *Proceedings of the WICSA 2014 Companion Volume*, page 1. ACM, 2014. iv, 51, 52, 72

[ABLS13] Vasilios Andrikopoulos, Tobias Binz, Frank Leymann, and Steve Strauch. How to adapt applications for the cloud environment. *Computing*, 95(6) :493–535, 2013. 20, 31

[act15] Apache activemq. http://activemq.apache.org/, 2015. 79, 80

[ADC10] Mohammed Alhamad, Tharam Dillon, and Elizabeth Chang. Conceptual sla framework for cloud computing. In *Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on*, pages 606–610. IEEE, 2010. 14

[AETE12] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 204–212. IEEE, 2012. iv, 24, 34, 49, 50, 72

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4) :50–58, 2010. 9

[AGN+13] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs scale-out for hadoop : Time to rethink ? In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 20. ACM, 2013. 103

[AJ98] M. Arlitt and T. Jin. 1998 world cup web site access logs. http://www.acm.org/sigcomm/ITA/, 1998. 48, 49

[ALHL14] Hanieh Alipour, Yan Liu, and Abdelwahab Hamou-Lhadj. Analyzing auto-scaling issues in cloud environments. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, pages 75–89. IBM Corp., 2014. 33

[ant15] Antlr : Another tool for language recognition. http://www.antlr.org/, 2015. 128

[apa15] Apache server. http://www.apache.org/, Août 2015. 35, 56, 96

[AWS16] Amazon Web Services (AWS). https://aws.amazon.com/fr/, 2016. 205

[AZR15] Microsoft Azure. https://azure.microsoft.com/fr-fr/pricing/details/virtual-machines/, Août 2015. 14, 22

[BAB12] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems*, 28(5) :755–768, 2012. 32

[Bar14] Olivier Barais. *Utilisation de la modélisation à l'exécution : objectif, challenges et bénéfices*. PhD thesis, Université de Rennes 1, 2014. 107

[BBC+03] David F Bantz, Chatschik Bisdikian, David Challener, John P Karidis, Steve Mastrianni, Ajay Mohindra, Dennis G Shea, and Michael Vanover. Autonomic personal computing. *IBM Systems Journal*, 42(1) :165–176, 2003. 26

[BBF09] Gordon Blair, Nelly Bencomo, and Robert B France. Models@ run. time. *Computer*, 42(10) :22–27, 2009. 107, 205

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software : Practice and Experience*, 36(11-12) :1257–1284, 2006. 71

[Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013. 129

[BHD13] Enda Barrett, Enda Howley, and Jim Duggan. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation : Practice and Experience*, 25(12) :1656–1674, 2013. iv, 22, 37, 72

[Bou90] Denis Bouyssou. Building criteria : a prerequisite for mcda. In *Readings in multiple criteria decision aid*, pages 58–80. Springer, 1990. 105, 162

[bpm16] Business process model and notation (bpmn). `http://www.bpmn.org/`, 2016. 160

[Bre12] Paul C Brebner. Is your cloud elastic enough ? : performance modelling the elasticity of infrastructure as a service (iaas) cloud applications. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 263–266. ACM, 2012. 33

[BRX13] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *Parallel and Distributed Systems, IEEE Transactions on*, 24(4) :681–690, 2013. iv, 54, 55, 72

[BYV08] Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopal. Market-oriented cloud computing : Vision, hype, and reality for delivering it services as computing utilities. In *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*, pages 5–13. Ieee, 2008. 9

[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. 9

[cac15] International conference on cloud and autonomic computing (cac). `http://autonomic-conference.org/`, 2015. 5, 172

[ccg14] International symposium on cluster, cloud and grid computing (ccgrid). `http://datasys.cs.iit.edu/events/CCGrid2014/`, 2014. 5

[CFH+05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005. 22

[CGB+02] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting software architectures : views and beyond*. Pearson Education, 2002. 35

[CGS06] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pages 2–8. ACM, 2006. 58, 72, 103

[CGS09] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on*, pages 132–141. IEEE, 2009. 57, 58, 72, 103

[CHL+08] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI*, volume 8, pages 337–350, 2008. 40, 41, 72

[clo15a] Apache cloudstack. https://cloudstack.apache.org/, 2015. 78

[clo15b] Amazon cloudwatch. https://aws.amazon.com/fr/cloudwatch/, Août 2015. 64

[CM12] Gianpaolo Cugola and Alessandro Margara. Processing flows of information : From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3) :15, 2012. 113

[CMTD13a] Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, and Schahram Dustdar. Multi-level elasticity control of cloud services. In *Service-Oriented Computing*, pages 429–436. Springer, 2013. 61, 67, 72, 100

[CMTD13b] Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, and Schahram Dustdar. Sybl : An extensible language for controlling elasticity in cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 112–119. IEEE, 2013. iv, 61, 67, 69, 70, 72

[CS13] Emiliano Casalicchio and Luca Silvestri. Autonomic management of cloud-based systems : The service provider perspective. In *Computer and Information Sciences III*, pages 39–47. Springer, 2013. iv, 64, 65, 72

[Dau13] Erwan Daubert. *Adaptation et cloud computing : un besoin d'abstraction pour une gestion transverse*. PhD thesis, INSA de Rennes, 2013. 71

[dep16] Api cloudstack. https://cloudstack.apache.org/api/apidocs-4.7/user/deployVirtualMachine.html, 2016. 98

[dev16] Définition de devops. http://www.lemagit.fr/definition/DevOps, 2016. 123

[DGVV12] Sourav Dutta, Sankalp Gera, Akshat Verma, and Balaji Viswanathan. Smartscale : Automatic application scaling in enterprise clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 221–228. IEEE, 2012. iv, 25, 34, 40, 43, 72

[Die02] Thomas G Dietterich. Machine learning for sequential data : A review. In *Structural, syntactic, and statistical pattern recognition*, pages 15–30. Springer, 2002. 41

[DKM+11] Xavier Dutreilh, Sergey Kirgizov, Olga Melekhova, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. Using reinforcement learning for autonomic resource allocation in clouds : Towards a fully automated workflow. In *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, pages 67–74, 2011. 22

[DKR14] Jörg Domaschka, Kyriakos Kritikos, and Alessandro Rossini. Towards a generic language for scalability rules. In *Advances in Service-Oriented and Cloud Computing*, pages 206–220. Springer, 2014. 69

[DLAL15] Simon Dupont, Jonathan Lejeune, Frederico Alvares, and Thomas Ledoux. Experimental analysis on autonomic strategies for cloud elasticity. In *2015 IEEE International Conference on Cloud and Autonomic Computing (ICCAC)*. IEEE, 2015. 5

[DOC15] Docker. https://www.docker.com/, 2015. 13, 98, 205

[dom16] Document object model (dom). https://www.w3.org/DOM/, 2016. 150

[dtd16] Définition de dtd. https://fr.wikipedia.org/wiki/Document_type_definition, 2016. 144

[DTM11] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. Elastic vm for cloud resources provisioning optimization. In *Advances in Computing and Communications*, pages 431–445. Springer, 2011. iv, 35, 56, 72

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext : implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010. 129

[eba15] Site e-commece ebay. http://www.ebay.com/, 2015. 53

[EC215a] Amazon Auto Scaling. https://aws.amazon.com/fr/autoscaling/, Août 2015. 24, 64, 69, 112

[EC215b] Amazon EC2. http://aws.amazon.com/fr/ec2/instance-types/, Août 2015. 14, 22, 31, 46, 64

[ecl15a] Eclipse ide. https://eclipse.org/, 2015. 69, 128, 129

[ecl15b] Eclipse ocl project. http://wiki.eclipse.org/OCL, 2015. 69

[Ehr13] Matthias Ehrgott. *Multicriteria optimization*, volume 491. Springer Science & Business Media, 2013. 105

[ela15] Elasticsearch. https://www.elastic.co/products/elasticsearch, 2015. 86

[emf15] Emf : Eclipse modeling framework. https://eclipse.org/modeling/emf/, 2015. 128

[epl15] Event processing language (epl). https://docs.oracle.com/cd/E13157_01/wlevs/docs30/epl_guide/overview.html, 2015. 118, 190

[esp15a] Api documentation esper. http://www.espertech.com/esper/release-5.3.0/esper-javadoc/, 2015. 120

[esp15b] Esper. http://www.espertech.com/esper/, 2015. 69, 117, 201

[esp16a] Esper configuration. http://www.espertech.com/esper/release-5.3.0/esper-reference/html_single/index.html/configuration, 2016. 201

[esp16b] Esper performance results. http://www.espertech.com/esper/release-5.3.0/esper-reference/html_single/index.html/performance-results, 2016. 201

[fac16] Nombre d'utilisateurs de facebook dans le monde. http://www.journaldunet.com/ebusiness/le-net/1125265-nombre-d-utilisateurs-de-facebook-dans-le-monde/, 2016. 200

[FGG10] Josep Oriol Fitó, Inigo Goiri, and Jordi Guitart. Sla-driven elastic cloud hosting provider. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 111–118. IEEE, 2010. 47, 72

[FHTG10] Markus Fiedler, Tobias Hossfeld, and Phuoc Tran-Gia. A generic quantitative relationship between quality of experience and quality of service. *Network, IEEE*, 24(2) :36–41, 2010. 97

[fif16] Workload trace : Fifa 1998 world cup. http://nbviewer.jupyter.org/urls/gitlab.com/soodeh/workloads/raw/master/worldcup.ipynb, 2016. 176

[FLWC12] Wei Fang, ZhiHui Lu, Jie Wu, and ZhenYin Cao. Rpps : a novel resource prediction and provisioning scheme in cloud data center. In *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, pages 609–616. IEEE, 2012. 40, 42, 43, 72

[Fou13] François Fouquet. *Kevoree : Model@ Runtime pour le développement continu de systèmes adaptatifs distribués hétérogènes*. PhD thesis, Université Rennes 1, 2013. 107, 205

[Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010. 109, 111, 122

[g5k15] Grid5000. https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home, 2015. 174, 182, 190

[gan15] Ganglia. http://ganglia.sourceforge.net/, 2015. 67

[Gar03] Lars Marius Garshol. Bnf and ebnf : What are they and how do they work. *acedida pela última vez em*, 16, 2003. 126

[gat15] Gatling. http://gatling.io/, 2015. 174, 177, 190

[GB12] Guilherme Galante and Luis Carlos E de Bona. A survey on cloud computing elasticity. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, pages 263–270. IEEE Computer Society, 2012. 33, 35, 36

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow : Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10) :46–54, 2004. iv, 35, 51, 52, 72, 103

[GdCdM+10] Pierre Gadonneix, Francisco Barnés de Castro, Norberto Franco de Medeiros, Richard Drouin, CP Jain, Younghoon David Kim, Jorge Ferioli, Marie-José Nadeau, Abubakar Sambo, Johannes Teyssen, et al. Energy efficiency : A recipe for success, 2010. 16

[GGRTRC13] Jesús Garcıa-Galán, OF Rana, Pablo Trinidad, and Antonio Ruiz-Cortés. Migrating to the cloud : a software product line based analysis. In *3rd International Conference on Cloud Computing and Services Science (CLOSER)*, pages 416–426, 2013. 14, 205

[GGW10] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press : Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE, 2010. iv, 22, 24, 40, 42, 50, 72

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Pearson Education, 1994. 118, 121

[goo15a] Google map. https://www.google.fr/maps, 2015. 78

[Goo15b] Google Compute Engine Autoscaler. https://cloud.google.com/compute/docs/autoscaler/, Août 2015. 24

[gra16] Gradle. http://gradle.org/, 2016. 129

[GSC09] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. Software architecture-based self-adaptation. In *Autonomic computing and networking*, pages 31–55. Springer, 2009. 57, 58, 72, 103

[GSL+12] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, Cornel Barna, and Gabriel Iszlai. Optimal autoscaling in a iaas cloud. In *Proceedings of the 9th international conference on Autonomic computing*, pages 173–178. ACM, 2012. 40, 72

[GSLI11] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. Exploring alternative approaches to implement an elasticity policy. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 716–723. IEEE, 2011. iv, 40, 72

[HGG+14] Rui Han, Moustafa M Ghanem, Li Guo, Yike Guo, and Michelle Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems*, 32 :82–98, 2014. iv, 46, 72

[HGGG12] Rui Han, Li Guo, Moustafa M Ghanem, and Yike Guo. Lightweight resource scaling for cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 644–651. IEEE, 2012. 34, 44, 45, 72

[HHPS08] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4) :93–95, 2008. 205

[HKR13] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing : What it is, and what it is not. In *ICAC*, pages 23–27, 2013. 18, 30, 33, 71, 100

[HM08] Markus C Huebscher and Julie A McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3) :7, 2008. 26

[HN04] Simon Haykin and Neural Network. A comprehensive foundation. *Neural Networks*, 2(2004), 2004. 41

[Hor01] Paul Horn. Autonomic computing : Ibm\'s perspective on the state of information technology. 2001. 3, 26, 27, 104

[Htt12] Michael Httermann. *DevOps for developers*. Apress, 2012. 123

[HZPW09] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on*, pages 630–637. IEEE, 2009. 56

[IDCJ11] Waheed Iqbal, Matthew N Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6) :871–879, 2011. 34, 50, 72

[IKLL12] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1) :155–162, 2012. 22, 40, 41, 72

[inn15] Innovation frugale. `https://fr.wikipedia.org/wiki/Innovation_frugale`, 2015. 93

[int16] Intellij idea. `https://www.jetbrains.com/idea/`, 2016. 129

[JAP14] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Autonomic resource provisioning for cloud-based software. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 95–104. ACM, 2014. iv, 67, 68, 72

[jbo15] Jboss server. `http://jbossas.jboss.org/`, Août 2015. 57

[jdo16] Java document object model (jdom). `http://www.jdom.org/`, 2016. 150

[jet15] Jetty. `http://www.eclipse.org/jetty/`, 2015. 190

[JHJ+10] Gueyoung Jung, Matti Hiltunen, Kaustubh R Joshi, Richard D Schlichting, Calton Pu, et al. Mistral : Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 62–73. IEEE, 2010. 62, 71, 72

[JLS10] Z. Jaroucheh, X. Liu, and S. Smith. A model-driven approach to flexible multi-level customization of saas applications. In *Proc. 22nd Int. Conf. Software Engineering and Knowledge Engineering (SEKE ¡ 10), San Francisco*, pages 241–246, 2010. 205

[JLZL13] Jing Jiang, Jie Lu, Guangquan Zhang, and Guodong Long. Optimal cloud resource auto-scaling for web applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 58–65. IEEE, 2013. 40, 72

[jme15] Apache jmeter. `http://jmeter.apache.org/`, 2015. 81

[JMS02] Li-jie Jin, Vijay Machiraju, and Akhil Sahai. Analysis on service level agreement of web services. *HP June*, page 19, 2002. 15

[Jun13] Frederico Alvares De Oliveira Junior. *Multi Autonomic Management for Optimizing Energy Consumption in Cloud Infrastructures*. PhD thesis, Université de Nantes, 2013. 63

[JWW+14] Hye-Jin Jin, Xiongfei Wang, Shiqian Wu, Sheng Di, and Xinzhi Shi. Towards optimized fine-grained pricing of iaas cloud platform. 2014. 32

[KAMV12] Pavlos Kranas, Vasileios Anagnostopoulos, Andreas Menychtas, and Theodora Varvarigou. Elaas : An innovative elasticity as a service framework for dynamic management across the cloud stack layers. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2012 Sixth International Conference on*, pages 1042–1049. IEEE, 2012. iv, 59, 60, 72

[KC03] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003. 3, 26, 27, 51, 57, 79, 104, 106, 139

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990. 205

[KDH11] Steffen Kächele, Jörg Domaschka, and Franz J Hauck. Cosca : an easy-to-use component-based paas cloud system for common applications. In *Proceedings of the First International Workshop on Cloud Computing Platforms*, page 4. ACM, 2011. 53

[KdODL14] Yousri Kouki, Frederico Alvares de Oliveira, Simon Dupont, and Thomas Ledoux. A language support for cloud elasticity management. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 206–215. IEEE, 2014. v, 5, 87, 88

[KDR14] Kyriakos Kritikos, Jorg Domaschka, and Alessandro Rossini. Srl : A scalability rule language for multi-cloud environments. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 1–9. IEEE, 2014. 69, 72

[KF11] Thomas Knauth and Christof Fetzer. Scaling non-elastic applications using virtual machines. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 468–475. IEEE, 2011. 22

[KL12] Yousri Kouki and Thomas Ledoux. Sla-driven capacity planning for cloud applications. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 135–140. IEEE, 2012. 15

[KMÅHR14] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. Brownout : building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 700–711. ACM, 2014. 53, 72, 103, 173, 174

[KML99] Nilesh N Karnik, Jerry M Mendel, and Qilian Liang. Type-2 fuzzy logic systems. *Fuzzy Systems, IEEE Transactions on*, 7(6) :643–658, 1999. 67

[Koo11] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times*, page 9, 2011. 16

[Kou13] Yousri Kouki. *Approche dirigée par les contrats de niveaux de service pour la gestion de l'élasticité du" nuage"*. PhD thesis, Nantes, Ecole des Mines, 2013. 65, 87, 88

[KP06] Kyriakos Kritikos and Dimitris Plexousakis. Semantic qos metric matching. In *Web Services, 2006. ECOWS'06. 4th European Conference on*, pages 265–274. IEEE, 2006. 69

[kvm15] Linux kvm. http://www.linux-kvm.org/, Août 2015. 43

[KY95] George Klir and Bo Yuan. *Fuzzy sets and fuzzy logic*, volume 4. Prentice Hall New Jersey, 1995. 38

[KYTA12] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. Workload characterization and prediction in the cloud : A multiple time series approach. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1287–1294. IEEE, 2012. 192

[LBCP09] Harold C Lim, Shivnath Babu, Jeffrey S Chase, and Sujay S Parekh. Automated control in cloud computing : challenges and opportunities. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 13–18. ACM, 2009. 46, 72

[LBMAL12] Tania Lorido-Botrán, José Miguel-Alonso, and Jose Antonio Lozano. Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, 12 :2012, 2012. 25, 80

[LBMAL14] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4) :559–592, 2014. 33, 35

[Let14] Loic Letondeur. *Planification pour la gestion autonomique de l'élasticité d'applications dans le cloud*. PhD thesis, Universite Joseph Fourier, 2014. 34, 66

[log15] Logstash. https://www.elastic.co/products/logstash, 2015. 85

[LRME13] Kathryn Bean Laura R. Moore and Tariq Ellahi. A coordinated reactive and predictive approach to cloud elasticity. *CLOUD COMPUTING 2013 : The Fourth International Conference on Cloud Computing, GRIDs, and Virtualization*, 2013. iv, 47, 48, 72

[LSD+03] Xue Liu, Lui Sha, Yixin Diao, Steven Froehlich, Joseph L Hellerstein, and Sujay Parekh. Online response time optimization of apache web server. In *Quality of Service—IWQoS 2003*, pages 461–478. Springer, 2003. 56

[LTM+11] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, and Dawn Leaf. Nist cloud computing reference architecture. *NIST special publication*, 500 :292, 2011. 9

[LXC15] Linux Container (LXC). https://linuxcontainers.org/fr/, 2015. 13

[MA04] R Timothy Marler and Jasbir S Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6) :369–395, 2004. 105, 152

[MADD04] Daniel A Menasce, Virgilio AF Almeida, Lawrence W Dowdy, and Larry Dowdy. *Performance by design : computer capacity planning by example*. Prentice Hall Professional, 2004. 23

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *ACM Sigplan Notices*, volume 22, pages 147–155. ACM, 1987. 107

[MBS11] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Enacting slas in clouds using rules. In *Euro-Par 2011 Parallel Processing*, pages 455–466. Springer, 2011. 22, 45, 72, 188

[MBS12] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Self-adaptive and resource-efficient sla enactment for cloud computing infrastructures. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 368–375. IEEE, 2012. 63, 72, 188

[MCTD13] Daniel Moldovan, Georgiana Copil, Hong-Linh Truong, and Schahram Dustdar. Mela : Monitoring and analyzing elasticity of cloud services. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 80–87. IEEE, 2013. iv, 61, 63, 64, 67, 72, 100

[MDT13] Christoph Meinel, Wesam Dawoud, and Ibrahim Takouna. Elastic vm for dynamic virtualized resources provisioning and optimization. *HPI Future SOC Lab : proceedings 2011*, 70 :13, 2013. 56, 72

[mdt15] Eclipse model development tools. `http://www.eclipse.org/modeling/mdt/`, 2015. 69

[Mey92] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10) :40–51, 1992. 152, 204

[MG11] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011. 9, 17, 18, 58

[MH12] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430. IEEE, 2012. 32

[MH13] Ming Mao and Marty Humphrey. Scaling and scheduling to maximize application performance within budget constraints in cloud workflows. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 67–78. IEEE, 2013. 47, 72

[MHS05] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4) :316–344, 2005. 109, 111

[Mic15] Microsoft Autoscaling Application Block. `https://msdn.microsoft.com/en-us/library/hh680892`, Août 2015. 24, 69

[MOC+14] Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, and Athanasios V Vasilakos. Cloud computing : Survey on energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2) :33, 2014. 16

[MR09] Jim Marino and Michael Rowley. *Understanding sca (service component architecture)*. Pearson Education, 2009. 65

[MRS11] Philippe Merle, Romain Rouvoy, and Lionel Seinturier. Frascati : Adaptive and reflective middleware of middleware. In *12th ACM/IFIP/USENIX International Middleware Conference*, 2011. 71

[MWM+14] Clarissa Cassales Marquezan, Florian Wessling, Andreas Metzger, Klaus Pohl, Chris Woods, and Karl Wallbom. Towards exploiting the full adaptation potential of cloud applications. In *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, pages 48–57. ACM, 2014. iv, 60, 61, 72

[ngi15] Nginx. https://www.nginx.com/, 2015. 174, 190

[NKHR14] Vladimir Nikolov, Steffen Kachele, Franz J Hauck, and Dieter Rautenbach. Cloudfarm : An elastic cloud platform with flexible and adaptive resource management. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 547–553. IEEE, 2014. 53, 54, 72, 103

[NMR+] Vladimir Nikolov, Matthias Matousek, Dieter Rautenbach, Lucia Draque Penso, and Franz J Hauck. Artos : System model and optimization algorithm. 53

[NRTV07] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V Vazirani. *Algorithmic game theory*, volume 1. Cambridge University Press Cambridge, 2007. 24

[ocl15] Langage ocl. http://www.omg.org/spec/OCL/, 2015. 69, 204

[ope15a] Open data nantes : ouverture des données publiques. http://data.nantes.fr/, 2015. 77, 78

[ope15b] Openstack. https://www.openstack.org/, 2015. 67, 174, 190

[osg15] Osgi. http://www.osgi.org/Specifications/HomePage, 2015. 53

[otp15] Opentripplanner. http://www.opentripplanner.org/, 2015. 78

[Par14] Fawaz Paraiso. *soCloud : une plateforme multi-nuages distribuée pour la conception, le déploiement et l'exécution d'applications distribuées à large échelle*. PhD thesis, Université des Sciences et Technologie de Lille-Lille I, 2014. vii, 65, 66

[PBvDL05] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering : foundations, principles and techniques*. Springer Science & Business Media, 2005. 205

[PHS+09] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26. ACM, 2009. iv, 22, 39, 72

[Pin12] Michael L Pinedo. *Scheduling : theory, algorithms, and systems*. Springer Science & Business Media, 2012. 161

[Pro15] ProfitBricks. https://www.profitbricks.com/, Août 2015. 22

[PSG06] Adrian Paschke and Elisabeth Schnappinger-Gerull. A categorization scheme for sla metrics. *Service Oriented Electronic Commerce*, 80 :25–40, 2006. 14

[Rac15] Rackspace Auto Scale. http://www.rackspace.com/cloud/auto-scale, Août 2015. 24

[RBX+09] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. Vconf : a reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th international conference on Autonomic computing*, pages 137–146. ACM, 2009. iv, 22, 38, 54, 72

[RDG11] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507. IEEE, 2011. 43, 62, 72

[rec12] Algorithmes de recommandation. http://www.podcastscience.fm/dossiers/2012/04/25/les-algorithmes-de-recommandation/, 2012. 172, 174, 188

[red15] Redis. http://redis.io/, 2015. 86

[RL80] Martin Reiser and Stephen S Lavenberg. Mean-value analysis of closed multichain queuing networks. *Journal of the ACM (JACM)*, 27(2) :313–322, 1980. 23

[rsa15] Chiffremment rsa. https://fr.wikipedia.org/wiki/Chiffrement_RSA, 2015. 97

[RTL96] Mark J Rentmeesters, Wei K Tsai, and Kwei-Jay Lin. A theory of lexicographic multi-criteria optimization. In *Engineering of Complex Computer Systems, 1996. Proceedings., Second IEEE International Conference on*, pages 76–79. IEEE, 1996. 105, 152

[rub15a] Prototype rubbos. http://forge.ow2.org/projects/rubbos/rubbos/, 2015. 53

[rub15b] Prototype rubis. http://rubis.ow2.org/, 2015. 53

[rub15c] Rubis benchmark. http://rubis.ow2.org/, 2015. 173

[RV89] Bernard Préfacier ROY and Philippe VINCKE. *L'aide multicritère à la décision*. Editions Ellipses, 1989. 105

[RVBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006. 24

[SAM13] Manuj Sabharwal, Abhishek Agrawal, and Grace Metri. Enabling green it through energy-aware software. *IT Professional*, (1) :19–27, 2013. 16

[Sax10] Eric Saxe. Power-efficient software. *Commun. ACM*, 53(2) :44–48, 2010. 16

[sax16] Simple api for xml (sax). http://www.saxproject.org/, 2016. 150

[SB98] Richard S Sutton and Andrew G Barto. *Reinforcement learning : An introduction*, volume 1. MIT press Cambridge, 1998. 24, 37

[Sch06] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2) :25, 2006. 205

[Sch09] Henry E Schaffer. X as a service, cloud computing, and the need for good judgment. *IT professional*, 11(5) :4–5, 2009. 11

[SGB+13] Omran Saleh, Francis Gropengieβer, Heiko Betz, Waseem Mandarawi, and Kai-Uwe Sattler. Monitoring and autoscaling iaas clouds : a case for complex event processing on data streams. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, pages 387–392. IEEE Computer Society, 2013. vii, 66, 72, 113

[SH10] Weiming Shi and Bo Hong. Resource allocation with a budget constraint for computing independent tasks in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 327–334. IEEE, 2010. 24

[SHRE13] Mina Sedaghat, Francisco Hernandez-Rodriguez, and Erik Elmroth. A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, page 6. ACM, 2013. 34, 103

[sla15] Slashdot effect (wikipedia). https://en.wikipedia.org/wiki/Slashdot_effect, 2015. 58

[SMM+12] Julia Schroeter, Peter Mucha, Marcel Muth, Kay Jugel, and Malte Lochau. Dynamic configuration management of cloud-based applications. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*, pages 171–178. ACM, 2012. 205

[SN05] James E Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5) :32–38, 2005. 12

[SSGW11] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale : elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 5. ACM, 2011. iv, 25, 40, 50, 72, 188

[SSJL12] Basem Suleiman, Sherif Sakr, Ross Jeffery, and Anna Liu. On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure. *Journal of Internet Services and Applications*, 3(2) :173–193, 2012. 30

[SSS13] Upendra Sharma, Prashant Shenoy, and Sambit Sahu. A flexible elastic control plane for private clouds. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, page 4. ACM, 2013. 66, 72

[SSSS11] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 559–570. IEEE, 2011. 22, 44, 72

[syn15] Définition synergie wikipedia. https://fr.wikipedia.org/wiki/Synergie, 2015. 94, 104

[TDC+14] Hong Linh Truong, Schahram Dustdar, Georgiana Copil, Alessio Gambi, Waldemar Hummer, Duc Hung Le, and Daniel Moldovan. Comot–a platform-as-a-service for elasticity in the cloud. In *2014 IEEE International Conference on Cloud Engineering (IC2E)*, pages 619–622. IEEE, 2014. iv, 61, 72

[TL14] Marian Turowski and Alexander Lenk. Vertical scaling capability of openstack. 2014. 22, 32

[tom15] Apache tomcat. http://tomcat.apache.org/, Août 2015. 35, 54, 57, 59, 78

[uml15] Unified modeling language (uml). http://www.uml.org/, 2015. 106

[USC+08] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(1) :1, 2008. iv, 48, 49, 72

[use15] Expérience utilisateur. https://fr.wikipedia.org/wiki/Expérience_utilisateur, 2015. 97

[VAN08a] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pmapper : power and migration cost aware application placement in virtualized systems. In *Middleware 2008*, pages 243–264. Springer, 2008. 16, 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*, pages 175–184. ACM, 2008. 16, 22

[Vin06] Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, (6) :87–89, 2006. 67

[VLWYH09] Gregor Von Laszewski, Lizhe Wang, Andrew J Younge, and Xi He. Power-aware scheduling of virtual machines in dvfs-enabled clusters. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009. 50, 205

[vMo15] Module vMotion de VMware vSphere. https://www.vmware.com/fr/products/vsphere/features/vmotion, Août 2015. 22

[VNM+12] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu : accelerating resource allocation in virtualized environments. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 423–436. ACM, 2012. 37, 72

[VRMCL08] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds : towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1) :50–55, 2008. 9

[WB12] Linlin Wu and Rajkumar Buyya. Service level agreement (sla) in utility computing systems. *IGI Global*, 2012. 15

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4) :279–292, 1992. 37

[Wei05] Sanford Weisberg. *Applied linear regression*, volume 528. John Wiley & Sons, 2005. 41

[WHGK14] Andreas Weber, Nikolas Herbst, Henning Groenda, and Samuel Kounev. Towards a resource elasticity benchmark for cloud environments. In *Proceedings of the 2nd International Workshop on Hot Topics in Cloud service Scalability*, page 5. ACM, 2014. 17, 18, 19, 30, 71

[wik16] Workload trace : Wikipedia. http://nbviewer.jupyter.org/urls/gitlab.com/soodeh/workloads/raw/master/wikipedia.ipynb, 2016. 176

[wil15] Wildcat. http://wildcat.ow2.org/backup2/introduction.html, 2015. 86

[WK98] Jos B Warmer and Anneke G Kleppe. The object constraint language : Precise modeling with uml (addison-wesley object technology series). 1998. 152

[xen15a] Xen credit scheduler. http://wiki.xen.org/wiki/Credit_Scheduler, Août 2015. 56

[xen15b] Xen server. http://xenserver.org/, Août 2015. 42, 43

[xpa16] Xml path language (xpath). https://www.w3.org/TR/1999/REC-xpath-19991116/, 2016. 144

[XRB12] Cheng-Zhong Xu, Jia Rao, and Xiangping Bu. Url : A unified reinforcement learning approach for autonomic cloud management. *Journal of Parallel and Distributed Computing*, 72(2) :95–105, 2012. iv, 54, 55, 72

[xte15a] Xtend. https://eclipse.org/xtend/documentation/, 2015. 128

[xte15b] Xtext. https://eclipse.org/Xtext/, 2015. 126, 128

[XZF+07] Jing Xu, Ming Zhao, Jose Fortes, Robert Carpenter, and Mazin Yousif. On the use of fuzzy modeling in virtualized data center management. In *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*, pages 25–25. IEEE, 2007. 38, 72

[Yah11] Interview de Todd Papaioannou par Julie Bort. http://www.networkworld.com/article/2179359/cloud-computing/yahoo-builds-ultimate-private-cloud.html, July 2011. 32

[YQL09] Jie Yang, Jie Qiu, and Ying Li. A profile-based approach to just-in-time scalability for cloud applications. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 9–16. IEEE, 2009. 20, 30, 32

[ZA10] Qian Zhu and Gagan Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 304–307. ACM, 2010. 58, 72

[ZBN+04] Liangzhao Zeng, Boualem Benatallah, Anne HH Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *Software Engineering, IEEE Transactions on*, 30(5) :311–327, 2004. 161

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing : state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1) :7–18, 2010. 9

[ZHLM10] Ying Zhang, Gang Huang, Xuanzhe Liu, and Hong Mei. Integrating resource consumption and allocation for infrastructure resources on-demand. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 75–82. IEEE, 2010. iv, 57, 72

[znn15] Znn.com. https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/model-problem-znn-com/, 2015. 57

[ZPL+12] Han Zhao, Miao Pan, Xinxin Liu, Xiaolin Li, and Yuguang Fang. Optimal resource rental planning for elastic applications in cloud market. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 808–819. IEEE, 2012. 24

[ZWS06] Xiaoyun Zhu, Zhikui Wang, and Sharad Singhal. Utility-driven workload management using nested control design. In *American Control Conference, 2006*, pages 6–pp. IEEE, 2006. 56

l'université nantes angers le mans
COMMUNAUTÉ D'UNIVERSITÉS ET ÉTABLISSEMENTS

MINES Nantes

Thèse de Doctorat

Simon DUPONT

Gestion autonomique de l'élasticité multi-couche des applications dans le Cloud

Vers une utilisation efficiente des ressources et des services du Cloud

Crosslayer elasticity management for Cloud

Towards an efficient usage of Cloud resources and services

Résumé

L'informatique en nuage, au travers de son modèle en couche et de l'accès à ses services à la demande, a bouleversé la façon de gérer les infrastructures (IaaS) et la manière de produire les logiciels (SaaS). Grâce à l'élasticité de l'infrastructure, la quantité de ressource peut être ajustée automatiquement en fonction de la demande afin de satisfaire un certain niveau de qualité de service (QoS) aux clients tout en minimisant les coûts d'exploitation sous-jacents. Le modèle d'élasticité actuel qui consiste à ajuster les ressources IaaS au travers de services de dimensionnement automatique basiques montre ses limites en termes de réactivité et de granularité d'adaptation. De plus, bien qu'étant une caractéristique cruciale de l'informatique en nuage, l'élasticité est à ce jour pauvrement outillée empêchant ainsi les différents acteurs du Cloud de jouir pleinement de ses bienfaits. Dans ce travail de thèse, nous proposons d'étendre le concept d'élasticité aux couches hautes du nuage, et plus précisément au niveau du SaaS. Nous présentons ainsi le nouveau concept d'*élasticité logicielle* que nous définissons comme la capacité d'un logiciel à s'adapter, idéalement de manière autonome, pour répondre aux changements de la demande et/ou aux limitations de l'élasticité des ressources de l'infrastructure. Il s'agit alors d'envisager l'élasticité de manière transverse et multi-couche en considérant l'adaptation des ressources Cloud au sens large. Pour ce faire, nous présentons un modèle pour la gestion autonome de l'élasticité multi-couche et le framework *ElaStuff* associé. Dans le but d'outiller et d'industrialiser le processus de gestion de l'élasticité, nous proposons l'outil de surveillance *perCEPtion* basé sur le traitement des événements complexes et permettant à l'administrateur de mettre en place une observation avancée du système Cloud. De plus, un langage dédié à l'élasticité multi-couche nommé *ElaScript* est proposé pour exprimer simplement et efficacement des plans de reconfiguration orchestrant les actions d'élasticité de différents niveaux. Enfin, notre proposition d'étendre l'élasticité aux couches hautes du Cloud, et plus particulièrement au niveau SaaS, est validée expérimentalement selon plusieurs points de vue (QoS, énergie, réactivité et précision du passage à l'échelle, etc.).

Mots clés

Informatique en Nuage, Élasticité, Informatique autonome, Dimensionnement automatique, Reconfiguration dynamique, Langage dédié.

Abstract

Cloud computing, through its layered model and access to its on-demand services, has changed the way of managing the infrastructures (IaaS) and how to produce software (SaaS). With the advent of IaaS elasticity, the amount of resources can be automatically adjusted according to the demand to satisfy a certain level of quality of service (QoS) to customers while minimizing underlying operating costs. The current elasticity model is based on adjusting the IaaS resources through basic autoscaling services, which reaches to its limit in terms of responsiveness and adaptation granularity. Although it is an essential feature for Cloud computing, elasticity remains poorly equipped which prevents the various actors of the Cloud to really enjoy its benefits. In this thesis, we propose to extend the concept of elasticity to higher layers of the cloud, and more precisely to the SaaS level. Then, we present the new concept of *software elasticity* by defining the ability of the software to adapt, ideally in an autonomous way, to cope with workload changes and/or limitations of IaaS elasticity. This predicament brings the consideration of Cloud elasticity in a multi-layer way through the adaptation of all kind of Cloud resources. To this end, we present a model for the autonomic management of multi-layer elasticity and the associated framework *ElaStuff*. In order to equip and industrialize the elasticity management process, we propose the *perCEPtion* monitoring tool, based on complex event processing, which enables the administrators to set up an advanced observation of the Cloud system. In addition, we propose a domain specific language (DSL) for the multi-layer elasticity, called *ElaScript*, which allows to simply and effectively express reconfiguration plans orchestrating the different levels of elasticity actions. Finally, our proposal to extend the Cloud elasticity to higher layers, particularly to SaaS, is validated experimentally from several perspectives (QoS, energy, responsiveness and accuracy of the scaling, etc.).

Key Words

Cloud computing, Elasticity, Autonomic computing, Autoscaling, Dynamic reconfiguration, Domain Specific Language.