Approche dirigée par les contrats de niveaux de service pour la gestion de l'élasticité du "nuage"

Yousri Kouki

▶ **To cite this version:**

Yousri Kouki. Approche dirigée par les contrats de niveaux de service pour la gestion de l'élasticité du "nuage". Génie logiciel [cs.SE]. Ecole des Mines de Nantes, 2013. Français. <NNT : 2013EMNA0134>. <tel-00919900>

HAL Id: tel-00919900

https://tel.archives-ouvertes.fr/tel-00919900

Submitted on 17 Dec 2013

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

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

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Yousri KOUKI

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure des mines de Nantes
Label européen
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 9 décembre 2013
Thèse n° : 2013 EMNA 0134

Approche dirigée par les contrats de niveaux de service pour la gestion de l'élasticité du "nuage"

JURY

Président :	**M. Claude Jard**, Professeur, Université de Nantes
Rapporteurs :	**M. Thierry Coupaye**, Responsable de recherche Cloud computing, Orange Labs
	M. François Taïani, Professeur, Université de Rennes 1
Examinateurs :	**Mme Fabienne Boyer**, Maître de conférences, Université Joseph Fourier, Grenoble
	M. Pierre Sens, Professeur, Université Paris 6
Directeur de thèse :	**M. Pierre Cointe**, Professeur, École des Mines de Nantes
Co-directeur de thèse :	**M. Thomas Ledoux**, Maître de conférences, École des Mines de Nantes

Remerciements

Je tiens à exprimer mes remerciements à M. Thierry Coupaye, responsable de recherche Cloud computing chez Orange Labs et M. François Taïani, Professeur à l'Université de Rennes 1, pour m'avoir fait l'honneur de rapporter les travaux de cette thèse. J'associe à ces remerciements M. Claude Jard, Professeur à l'Université de Nantes, d'avoir présidé mon jury de thèse. Mes remerciements s'adressent également à Mme Fabienne Boyer, Maître de conférences à l'Université Joseph Fourier et M. Pierre Sens, Professeur à l'Université Paris 6, pour avoir été examinateurs de thèse.

Je tiens à exprimer ma profonde gratitude et mes remerciements les plus sincères à M. Pierre Cointe, Professeur à l'École des Mines de Nantes et M. Thomas Ledoux, Maître de conférences à l'École des Mines de Nantes, 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 toute l'équipe du projet MyCloud, pour m'avoir permis de réaliser mes travaux dans un contexte professionnel et scientifique riche.

Mes remerciements vont également à tous les collègues du laboratoire LINA et en particulier les membres de l'équipe ASCOLA pour les moments agréables partagés durant cette période.

Je remercie de tout mon cœur mes chers parents, ma sœur et mes frères pour leur soutien inconditionnel. Je terminerai par une mention spéciale à ma femme pour tout l'amour qu'elle me porte et à mes anges Yasmine, Tasnime et ceux qui suivent...

Table des matières

1 Introduction **1**
1.1 Contexte et problématique . . . 2
1.2 Illustration . . . 3
1.3 Objectifs de la thèse . . . 4
1.4 Contributions . . . 6
1.5 Plan de la thèse . . . 7
1.6 Diffusion scientifique . . . 7

I État de l'art **9**

2 Informatique utilitaire **11**
2.1 Accord de niveau de service - SLA . . . 12
2.1.1 ABC du SLA . . . 12
2.1.2 Cycle de vie . . . 13
2.1.3 SLA @ informatique utilitaire . . . 15
2.2 Informatique en nuage, les concepts de base . . . 15
2.2.1 Notion de service . . . 16
2.2.2 Qu'est-ce que l'informatique en nuage ? . . . 17
2.2.3 Modèles de l'informatique en nuage . . . 19
2.3 Informatique en nuage, les ressources virtuelles . . . 21
2.3.1 Virtualisation . . . 21
2.3.2 Modélisation des ressources virtuelles . . . 23
2.3.3 Modèle économique sous-jacent . . . 25
2.4 Informatique autonomique . . . 27
2.4.1 Définition . . . 28
2.4.2 Propriétés autonomiques . . . 28
2.4.3 Boucle de contrôle autonomique . . . 29
2.5 Conclusion . . . 30

3 Gestion des contrats de niveaux de service **31**
3.1 Accord de niveau de service (SLA), la définition . . . 32
3.1.1 SLA @ service Web . . . 32
3.1.2 SLA @ Cloud . . . 36
3.1.3 SLA niveau IaaS . . . 39
3.1.4 Discussion . . . 41
3.2 Accord de niveau de service (SLA), les dépendances . . . 43
3.2.1 Les dépendances, les concepts . . . 43
3.2.2 Les dépendances, la pratique . . . 45

3.2.3 Discussion 47
3.3 Gestion de Capacité des ressources 48
3.3.1 Dimensionnement automatique, les concepts 48
3.3.2 Dimensionnement automatique, la pratique 54
3.3.3 Autres approches de gestion de capacité 61
3.3.4 Discussion 64
3.4 Synthèse 66

II Contributions scientifiques 69

4 CSLA : accord de niveau de service Cloud 71
4.1 Langage CSLA 72
4.1.1 Nouveautés de CSLA 72
4.1.2 Modèle conceptuel 72
4.1.3 Modèle économique 78
4.1.4 Exemple illustratif 84
4.1.5 Cycle de vie de CSLA 87
4.1.6 Expérimentations 88
4.2 Dépendances SLA 95
4.2.1 Calibrage de CSLA template, la méthode 96
4.2.2 Calibrage de CSLA template, la pratique 98
4.2.3 Expérimentations 101
4.3 Synthèse 104

5 HybridScale : dimensionnement automatique dirigé par SLA 105
5.1 *HybridScale*, vue d'ensemble 106
5.1.1 Nouveautés d'*HybridScale* 106
5.1.2 Modèle conceptuel 106
5.2 *BestPolicies* : politiques de gestion d'élasticité 107
5.2.1 Politiques des prix 107
5.2.2 Politiques de réaction 108
5.2.3 Politiques de dimensionnement 109
5.3 LoadForecasting : prédiction de charge de travail 111
5.3.1 Modélisation d'une série chronologique 111
5.3.2 Forecaster, la technique 112
5.4 *RightCapacity* : planification de capacité dirigée par SLA 114
5.4.1 Modèle Conceptuel 114
5.4.2 Modèle analytique 114
5.4.3 Planification des ressources, la méthode 120
5.4.4 Planification de l'architecture logicielle, la méthode 126
5.5 Synthèse 127

6 Prototype HybridScale et expérimentations 129
6.1 Prototype *HybridScale* 130
6.1.1 Boucle de contrôle MAPE-K 130
6.1.2 Cycle de vie du gestionnaire autonome 135
6.1.3 Implémentation 136
6.2 Expérimentations 139
6.2.1 Protocole d'expérimentation 139

6.2.2 Résultats . . . 141
6.3 Synthèse . . . 148

III Conclusion 149

7 Conclusion et perspectives 151
7.1 Conclusion . . . 152
7.1.1 Contexte et problématique . . . 152
7.1.2 Contributions . . . 153
7.2 Perspectives . . . 154
7.2.1 Amélioration . . . 155
7.2.2 Extensions . . . 155
7.2.3 Pistes exploratoires . . . 155

IV Annexes 171

A Syntaxe de CSLA 173

B Amazon Auto Scaling++ 181
B.1 Modèle conceptuel . . . 181
B.2 Elasticité de l'infrastructure . . . 182
B.3 Elasticité de l'architecture applicative . . . 182
B.4 Implémentation . . . 182

Liste des tableaux

2.1 Modèles des ressources 25

3.1 Bilan de SLA niveau IaaS 40
3.2 Solutions SLA 43
3.3 Bilan de repartiteurs de charge 63
3.4 Dimensionnement, les initiatives de recherche 65

4.1 SLA niveau SaaS (SLA$_S$) 89
4.2 SLA niveau IaaS (SLA$_R$) 89
4.3 SLA niveau SaaS (SLA_S) 92
4.4 SLA niveau IaaS (SLA_R) 93
4.5 Aggrégation des QoS 99
4.6 service-ressource 101
4.7 SLA_R 102
4.8 Traduction des préférences 103
4.9 Solutions pour préférence 1 103

6.1 SLA niveau SaaS (SLA$_S$) 140
6.2 SLA niveau IaaS (SLA$_R$) 140
6.3 Scaling Policies 141
6.4 Expérimentations 141

B.1 Règles à base des seuils niveau IaaS 182
B.2 Règles à base des seuils niveau SaaS 182

Table des figures

1.1 Scénario illustratif 4

2.1 Cycle de vie SLA (Ron et al. 2001) 14
2.2 Cycle de vie SLA (Sun 2002) 14
2.3 Service XaaS 20
2.4 Référence NIST 2011 [FJJ+11] 21
2.5 Machine virtuelle 22
2.6 Modèle de ressource Amazon EC2 [drA13] 23
2.7 Modèle de ressource Oracle [Ora10] 24
2.8 Modèle de ressource Deltacloud [drD13] 24
2.9 Modèle de ressource OCCI [OCC10] 25
2.10 Elasticité 26
2.11 Modèle économique Amazon EC2 [EC213a] 27
2.12 boucle de contrôle autonomique MAPE-K [KC03] 29

3.1 Méta-modèle WSLA 33
3.2 Méta-modèle WS-Agreement 34
3.3 Méta-modèle SLA@SOI 37
3.4 WSLA, WS-agreement et SLA* 41
3.5 inter-dépendances de SLA 45
3.6 inter-dépendances de QdS 45
3.7 Gestion du niveau de services 48
3.8 Application Web n-tiers 54

4.1 Méta-modèle CSLA 73
4.2 Validité 73
4.3 Les parties 74
4.4 Définition des services 74
4.5 Les paramètres 75
4.6 Le scope 76
4.7 Les exigences 76
4.8 Les termes de garanties 77
4.9 Les Pénalités 77
4.10 Le billing 78
4.11 La terminaison 78
4.12 Per-interval, Per-request 80
4.13 idéal, dégradation et inadéquat 80
4.14 pourcentage des requêtes 81
4.15 Coût d'une pénalite 83
4.16 Cycle de vie CSLA 87

4.17 Expérience 1 90
4.18 Expérience 2 91
4.19 Cas d'étude 2 92
4.20 Les performances 93
4.21 Les profits 93
4.22 Interface graphique 94
4.23 La méthode de calibrage 96
4.24 Processus métier vertical 97
4.25 Processus métier vertical utilisé 102
4.26 Scalabilité 104

5.1 HybridScale, modèle conceptuel 107
5.2 BestPolicies 108
5.3 Politiques de réaction 108
5.4 Politiques de dimensionnement 109
5.5 Prédiction des valeurs futures 111
5.6 Méta-modèle Forecaster 113
5.7 Technique de prédiction 113
5.8 Méta-modèle RightCapacity 115
5.9 Modélisation d'une architecture M-tiers avec des réseaux fermés 116
5.10 Modèle Analytique 117
5.11 Méta-modèle Infrastructure 117
5.12 Méta-modèle Application 118
5.13 Méta-modèle Workload 118
5.14 Méthode de planification de capacité 122

6.1 Méta-modèle MAPE-K 130
6.2 Méta-modèle Monitoring 132
6.3 Méta-modèle Analyze 133
6.4 Méta-modèle Plan 133
6.5 Méta-modèle Execute 134
6.6 Initialisation de la boucle 135
6.7 Cycle MAPE-K réactif/proactif 136
6.8 Notification d'un dépassement de seuil 136
6.9 Cycle de vie de l'application 137
6.10 Environnement d'évaluation 142
6.11 Un workload imprévisible 143
6.12 Un workload *On and Off* 145
6.13 Un workload prévisible 147

B.1 Méta-modèle AAS++ 181
B.2 Prototype *AAS++* 183

1

Introduction

Ce chapitre présente le contexte de nos travaux qui portent sur la gestion des niveaux de services pour l'informatique en nuage (Cloud computing). Nous présentons le compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients via un scénario illustratif. 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 dirigée par un contrat de niveau de service (SLA - Service Level Agreement).

Sommaire

1.1 Contexte et problématique 2
1.2 Illustration 3
1.3 Objectifs de la thèse 4
1.4 Contributions 6
1.5 Plan de la thèse 7
1.6 Diffusion scientifique 7

1.1 Contexte et problématique

Le contexte général de la thèse porte sur la gestion de la qualité de service (QdS) et l'accord de niveau de service (Service Level Agreement - SLA) pour l'informatique en nuage (Cloud computing).

A l'heure où la qualité de service (QdS) [SPU13] [ea13] présente un des éléments différenciateurs entre les offres de services du Cloud, la gestion des niveaux de services prend tout son sens. Pour cela est mis en place un accord de niveau de service (SLA), notamment pour les services payants, entre le fournisseur de service et son client [Bas12] [ADC11]. Il spécifie un ou plusieurs objectifs de niveau de service (Service Level Objective - SLO), afin de garantir que la QdS delivrée a satisfait les attentes du client. En cas de violation, des pénalités sont appliquées au fournisseur. Le défi principal d'un fournisseur de service de Cloud est d'assurer que la capacité des ressources utilisées augmente de façon continue durant les pics de demande pour maintenir la QdS en vue de garantir les SLAs, et diminue automatiquement durant la baisse de demande pour minimiser les coûts des ressources. Suivre de près la courbe de demande d'un service est une question de compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients.

Avec les infrastructures traditionnelles, la capacité des ressources d'un service informatique est définie lors de sa conception. Cependant sa charge de travail peut varier dans le temps, et parfois de manière imprévisible. Cela conduit généralement soit à une infrastructure sous-chargée ou à une infrastructure surchargée. Qu'il s'agisse d'une sous-charge ou d'une surcharge, ces derniers peuvent entraîner des conséquences négatives respectivement sur le niveau de qualité de service (QdS) ou le coût du service. Le sous-dimensionnement implique non seulement une diminution de la performance de service mais parfois l'indisponibilité de service. Une telle situation risque d'entrainer l'insatisfaction des utilisateurs. Alors que le sur-dimensionnement, même si il permet de répondre au besoin du service, fait référence à un gaspillage d'une partie de budget en ressources inutiles. Afin d'éviter une sous/sur-charge et absorber les variations, l'ajustement automatique et continu de la capacité de ressources est incontournable.

En réponse à ce problème, l'informatique en nuage [MGG11] (Cloud computing) assouplit considérablement la gestion des ressources. Ce nouveau modèle consiste à proposer les ressources informatiques sous forme des services à la demande. La formule (Pay-as-you-go) et l'élasticité [NRSR13] sont les principales caractéristiques de ce modèle. L'infrastructure dans le nuage, par exemple, offre l'accès à des services de calcul, de stockage et de réseau, tout en facturant uniquement les services réellement consommés. L'investissement initial en infrastructure sera pratiquement nul. En plus, c'est une infrastructure juste-à-temps avec un délai d'acquisition réduit à quelques minutes.

La migration vers l'informatique en nuage reflète le besoin de simplification de gestion des ressources. Ressources juste-à- temps, plus de disponibilité, et déploiement automatique sont les clés nécessaires mais pas suffisantes pour absorber les variations.

Des initiatives industrielles telles que Amazon Auto-Scaling[1], RightScale[2] et Scalr[3] ainsi que des initiatives de recherche comme CloudScale [SSGW11] ou SmartScale [DGVV12] proposent des services de dimensionnement automatique. Amazon Auto-Scaling, par exemple, permet d'augmenter ou diminuer automatiquement la capacité Amazon Elastic Compute Cloud (EC2) selon des conditions prédéfinies. Cependant, un effort important de l'utilisateur de service de dimensionnement est nécessaire pour la planification de capacité : définir le nombre de ressources à ajouter ou à supprimer pour ajuster la capacité automatiquement selon un modèle de charge de travail n'est pas si immédiat !

La solution de dimensionnement doit tenir compte d'une part, du modèle économique de l'informatique en nuage à savoir le modèle de facturation (à l'heure, par exemple), le temps non-négligeable d'initialisation de ressources et, d'autre part, des critères de QdS de haut niveau (temps de réponse, par exemple). En effet, l'utilisation des critères de QdS de bas niveau tel que CPU est un bon indicateur de l'utilisation du système, mais il ne reflète pas clairement si la QdS fournie répond aux besoins des utilisateurs. En plus, trouver la bonne translation (mapping) entre des critères de QdS de haut niveau et de bas niveau est un problème difficile à maîtriser.

1.2 Illustration

Aujourd'hui, selon le Gartner Group[4], 90% des entreprises utilisent des applications en mode SaaS ou envisagent de le faire. Ses principaux atouts sont le coût et la QdS.

Dans notre exemple fil conducteur, le service considéré est un service de type SaaS (e-commerce, informatique décisionnelle,...) accessible en ligne par plusieurs clients. La charge du service SaaS, en terme de nombre de clients concurrents accédant au service, peut être plus ou moins importante et peut varier au cours du temps. Le service SaaS est une application Web multi-tier hébergée par des machines virtuelles dans le Cloud. Ces dernières sont fournies par un fournisseur IaaS. Nous distinguons trois acteurs : i) le client final : c'est le consommateur de service SaaS, ii) le fournisseur SaaS : c'est le fournisseur de service SaaS et le consommateur de service IaaS et iii) le fournisseur IaaS : c'est le fournisseur des ressources informatiques (des machines virtuelles).

Les principaux critères de QdS perçus par un client final sont relatifs à la performance et à la disponibilité du service SaaS.

Disponibilité : La disponibilité d'un service est l'attribut le plus directement perceptible d'un point de vue client. Un service est disponible seulement si les utilisateurs peuvent y accéder comme prévu. La perception et les préférences varient d'un client à l'autre et d'un contexte à un autre. Ce critère de QdS est exprimé classiquement sous forme de pourcentage pour un intervalle de temps. C'est un critère à maximiser. En effet, le taux de rejet correspond au pourcentage de requêtes rejetées par le système et doit être très faible pour avoir une meilleure disponibilité de service.

[1] aws.amazon.com/autoscaling/
[2] www.rightscale.com
[3] http ://wiki.scalr.com/display/docs/Scaling
[4] www.gartner.com

Performance : Un service est caractérisé par sa performance qui indique la rapidité du traitement des requêtes. L'image d'un service est représentée par le temps de réponse (ou encore la latence). Ce dernier est le temps utile pour le traitement d'une requête, il est exprimé en secondes (ou millisecondes) et représente un critère de QdS à minimiser. Plusieurs études ont démontré l'influence de la latence sur le comportement des utilisateurs face à un système [Mil68]. "0,1 seconde" est la latence en dessous de laquelle le système est vu comme répondant instantanément. L'utilisateur remarque le délai de la latence même s'il est de quelques millisecondes et au delà de "10 secondes", et il peut interrompre son interaction avec le système.

Afin de formaliser la QdS entre les différents acteurs, nous distinguons deux niveaux de SLA (voir Figure 1.1) :

SLA$_S$: c'est le contrat entre le client final et le fournisseur de SaaS. Il spécifie un ensemble d'objectifs de niveau de service SaaS en terme de performance et disponibilité.

SLA$_R$: c'est le contrat entre le fournisseur SaaS et le fournisseur IaaS. Il indique un objectif de niveau de service IaaS en terme de disponibilité.

FIGURE 1.1 – Scénario illustratif

Le fournisseur SaaS est un producteur-consommateur : il fournit un service SaaS à ses clients et il est un client d'un ou plusieurs IaaS. Le nombre des ressources allouées par le fournisseur SaaS peut être plus ou moins important, pour répondre à une charge de ses clients plus ou moins importante. Intuitivement, plus le nombre d'instances est important et meilleures sont la performance et la disponibilité de service SaaS rendu, mais plus élevé est le coût du service SaaS. De plus, une configuration unique ne peut répondre à toutes les variations de charge. L'objectif du fournisseur SaaS est d'améliorer son profit. Ceci revient à minimiser le coût du service fourni tout en minimisant les violations de SLA et d'une manière indirecte les pénalités. Minimiser le côut de service en respectant le SLA est une question de compromis.

1.3 Objectifs de la thèse

Cette thèse adresse compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients. Elle vise à fournir une solution permettant d'implémenter l'élasticité de

l'informatique en nuage pour satisfaire ce compromis. Les principaux verrous à lever par ce travail sont les suivants : la définition de SLA, les dépendances du SLA entre couches XaaS et la gestion de capacité des ressources.

La définition de SLA : (Fournisseurs SaaS/IaaS) Comment définit-on nos SLAs ?

WSLA [LF03] et WS-Agreement [Aa07] ont contribué de manière significative à la standardisation de SLA. Cependant, aucun ne propose une solution qui supporte les caractéristiques des services Cloud. Des initiatives, comme SLA@SOI [KTK10], ont été développées pour répondre à ces limites mais ne tiennent pas compte de l'instabilité de la QdS dans un Cloud hautement dynamique.

Un des buts de ce travail est de proposer un langage qui facilite la définition de SLA pour n'importe quel service Cloud XaaS (*anything as a Service*) et améliore la gestion de SLA en tenant compte de l'instabilité de la QdS.

Les dépendances du SLA : (Fournisseur SaaS) Comment puis-je définir, calibrer et négocier un SLA avec mon client dans le cadre de mon SLA avec mon fournisseur ?

Il est clair que SLA_S et SLA_R présentent des dépendances fortes. En effet, les couches XaaS sont connectées de manière client-fournisseur où une couche bénéfice des services/ressources des niveaux inférieurs et fournit un service/ressource aux niveaux supérieurs. Il s'agit d'une dépendance verticale entre les différentes couches XaaS. Dans notre contexte, une violation de SLA_R peut provoquer une violation de SLA_S. En se basant sur SLA_R avec le fournisseur IaaS, le fournisseur SaaS doit définir, calibrer et négocier le SLA_S avec ses clients.

Le deuxième but de cette thèse est la maîtrise des dépendances du SLA.

La gestion de capacité : (Fournisseur SaaS) Comment puis-je respecter les SLAs tout en maximisant mon profit ?

L'implémentation de l'élasticité de l'informatique en nuage fait apparaître un certain nombre de défis en induisant diverses contraintes comme le modèle économique, plusieurs types de ressources et le temps d'initialisation des ressources non négligeable [MH12]. La gestion de capacité est l'assurance qu'un service répondra à un niveau spécifié de demandes avec un niveau spécifié de qualité. Une gestion efficace de la capacité de service peut ainsi avoir des effets directs sur la garantie de service. Le fournisseur SaaS doit gérer la capacité de son service de telle sorte à allouer le nombre d'instances nécessaires pour garantir le contrat signé avec ses clients mais aussi le minimum possible pour réduire le coût associé.

L'objectif de cette thèse est de veiller à ce que la capacité puisse répondre aux objectifs de niveau de service d'une manière rentable (coût), ponctuelle (juste à temps) et autonome.

1.4 Contributions

L'apport principal de cette thèse est la résolution du compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients par un dimensionnement automatique des ressources dirigé par les SLAs. Les contributions majeures sont le langage *CSLA* et le framework de dimensionnement automatique *HybridScale* qui englobe d'autres contributions : *RightCapacity* et *BestPolicies*. L'ensmble des contributions est validé via des expérimentations dans l'environnement IaaS Amazon EC2.

Définition de SLA

CSLA : accord de niveau de service pour l'informatique en nuage CSLA est un langage pour définir des contrats SLA. L'originalité de *CSLA* est double i) grâce à certaines propriétés, il améliore la gestion de SLA, en particulier la gestion des violations via la dégradation de fonctionnalité, la dégradation de QdS et un modèle avancé de pénalités et ii) il suit l'architecture de référence de l'informatique en nuage de l'institut national des standards et de la technologie (NIST)[5] et supporte l'interface OCCI (Open Cloud Computing Interface)[6]. Un aspect important de *CSLA* est sa capacité à s'adapter à n'importe quel service du nuage informatique.

Dépendances SLA. Nous traitons partiellement le problème de dépendances de SLA via la programmation par contraintes [RBW06]. En se basant sur les préférences du client, notre solution fournit au "*design-time*" un SLA possible au client et une configuration optimale au fournisseur SaaS en terme de type et nombre de ressources IaaS.

Gestion de SLA

BestPolicies : politiques de gestion de l'élasticité de l'informatique en nuage. Afin de tenir compte du contexte de l'informatique en nuage, *BestPolicie*s propose des politiques adaptées pour gérer les services en nuage. L'originalité de BestPolicies repose principalement sur trois types de politiques : i) les politiques de prix , ii) les politiques de réaction et iii) les politiques de dimensionnement. Pour les dernières, nous avons proposé la liste de politiques suivantes : *Infrastructure Elasticity* qui consiste à contrôler le dimensionnement des ressources et *Application Elasticity* qui gère l'élasticité applicative (la dégradation de fonctionnalité).

RightCapacity : planification de capacité dirigée par SLA. *RightCapacity* est une méthode qui calcule la capacité optimale pour une charge de travail donnée selon les SLAs associés. L'originalité de RightCapacity réside dans la planification multi-couches (*cross-layer*). Il prend compte deux niveaux à savoir : l'application (SaaS) et l'infrastructure (IaaS).

HybridScale : dimensionnement automatique dirigé par SLA. HybridScale est une implémentation avancée de l'élasticité de l'informatique en nuage. Elle englobe les autres contibutions. De plus, elle contient un algorithme de prédiction basé sur des méthodes de série chronologique [BJ94]. L'originalité de *HybridScale* est le triple hybride : gestion réactive-proactive, dimensionnement vertical-horizontal et multi-

[5] http ://www.nist.gov/index.html
[6] occi-wg.org/

couches (application-infrastructure). *HybridScale* est implémentée via une boucle de contrôle autonomique MAPE-K [Hor01].

1.5 Plan de la thèse

Ce mémoire de thèse est découpé en trois grandes parties qui sont organisées comme suit :

1. la première partie concerne la problématique de la thèse. Elle inclut le chapitre 1 d'introduction déja présenté, le chapitre 2 qui donne un aperçu sur l'informatique utilitaire et le chapitre 3 qui présente un état de l'art sur la définition d'un SLA, la dépendances SLA et la gestion de capacité. Le chapitre 3 se termine par une mise en perspective de notre approche avec les solutions existantes.

2. la deuxième partie présente les contributions scientifiques de cette thèse. Elle inclut le chapitre 4 qui illustre le langage CSLA, le chapitre 5 qui présente HybridScale, un framework de dimensionnement automatique dirigé par CSLA, et le chapitre 6 qui qui expose nos prototypes de recherche ainsi que les exprémentations.

3. la troisième partie conclut ce mémoire de thèse en récapitulant les contributions apportées et annonçant les perspectives de travaux de recherche futurs dans le chapitre 7.

1.6 Diffusion scientifique

Les différents travaux présentés dans ce document ont fait l'objet de diverses publications listées ci-dessous.

Chapitre de livre :

1. **Y. Kouki** , F. Alavares and T. Ledoux. Cloud Capacity Planning and Management. Encyclopedia on Cloud Computing, Ch-35. Editors : San Murugesan, Irena Bojanova. Wiley. 2014.

Journal international avec comité de lecture :

1. **Y.Kouki** and T.Ledoux. RightCapacity : SLA-driven Cross-Layer Cloud Elasticity Management. International Journal of Next-Generation Computing (IJNGC), Vol. 4, No. 3, 2013.

Conférence internationale avec comité de lecture :

1. D. Serrano, S. Bouchenak, **Y.Kouki**, T.Ledoux, J. Lejeune, J. Sopena, L. Arantes and P. Sens. Towards QoS-Oriented SLA Guarantees for Online Cloud Services. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2013), Delft, The Netherlands, 2013. (taux d'acceptation 22.18%).

2. **Y. Kouki** and T. Ledoux. SCAling : SLA-driven Cloud Auto-scaling. ACM Symposium on Applied Computing (SAC 2013), Coimbra, Portugal, 2013.

3. **Y. Kouki** and T. Ledoux. SLA-driven Capacity Planning for Cloud applications. IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2012), Taipei, Taiwan, 2012. (taux d'acceptation 17%).

4. **Y.Kouki** and T.Ledoux. CSLA : a language for improving Cloud SLA Management. International Conference on Cloud Computing and Services Science (CLOSER 2012), Porto, Portugal, 2012. (taux d'acceptation 19%).

5. **Y. Kouki**, T. Ledoux and R. Sharrock. Cross-layer SLA selection for Cloud services. IEEE International Symposium on Network Cloud Computing and Applications (NCCA 2011), Toulouse, France, 2011.

Conférence nationale avec comité de lecture :

1. **Y.Kouki**, T.Ledoux, D. Serrano, S. Bouchenak, J. Lejeune, J. Sopena, L. Arantes and P. Sens. SLA et qualité de service pour le Cloud Computing. Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS 2013), Grenoble, France, 2013.

Posters :

1. **Y. Kouki** and T. Ledoux. SCAling : SLA-driven Cloud Auto-scaling. ACM Symposium on Applied Computing (SAC 2013), Coimbra, Portugal, 2013.

2. OpenCloudware : The Open Source Cloud Application Lifecycle Management Project. Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS 2013), Grenoble, France, 2013.

3. MyCloud : SLA and Quality-of-Service for Cloud Computing. Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS 2013), Grenoble, France, 2013.

Soumissions :

1. **Y. Kouki** , F. Alvares and T. Ledoux. Language support for Cloud Elasticity Management. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014).

2. D. Serrano, S. Bouchenak, **Y.Kouki**, F. Alvares , T.Ledoux, J. Lejeune, J. Sopena, L. Arantes and P. Sens. SLA Guarantees for Cloud Services. IEEE Transactions on Cloud Computing. 2014.

De plus, j'ai contribué à la rédaction des livrables des projets ANR MyCloud [MyC13] (2010-2013) et FSN OpenCloudware [Ope13](2012-2014).

État de l'art

2

Informatique utilitaire

L'informatique utilitaire est un concept qui a été proposé en 1961 par John McCarthy lors d'une conférence au MIT (Massachusetts Institute of Technology) [GA99]. Il a expliqué que les ressources informatiques pourraient un jour être organisées comme un service public tout comme le téléphone, l'électricité ou le gaz. Ce concept a l'avantage d'avoir un coût d'investissement initial très faible pour un accès immédiat aux ressources informatiques.

Telle que l'imaginait John McCarthy, l'émergence du Cloud computing représente une suite logique dans l'histoire de l'informatique. Ce modèle doit son existence à l'uniformisation des technologies sous-jacentes telles que le Web 2.0, l'architecture orientée services (SOA) et la virtualisation. La garantie de la QdS de service délivrée par le Cloud devient aujourd'hui une question cruciale. Dans ce que suit, nous présentons d'abord, le contrat de niveau de service (Accord de niveau de service - SLA). Puis, nous détaillons le paradigme du Cloud computing. Enfin, nous présentons l'informatique autonomique.

Sommaire

2.1 Accord de niveau de service - SLA . . . 12
- 2.1.1 ABC du SLA . . . 12
- 2.1.2 Cycle de vie . . . 13
- 2.1.3 SLA @ informatique utilitaire . . . 15

2.2 Informatique en nuage, les concepts de base . . . 15
- 2.2.1 Notion de service . . . 16
- 2.2.2 Qu'est-ce que l'informatique en nuage ? . . . 17
- 2.2.3 Modèles de l'informatique en nuage . . . 19

2.3 Informatique en nuage, les ressources virtuelles . . . 21
- 2.3.1 Virtualisation . . . 21
- 2.3.2 Modélisation des ressources virtuelles . . . 23
- 2.3.3 Modèle économique sous-jacent . . . 25

2.4 Informatique autonomique 27
2.4.1 Définition 28
2.4.2 Propriétés autonomiques 28
2.4.3 Boucle de contrôle autonomique 29
2.5 Conclusion 30

2.1 Accord de niveau de service - SLA

Le Service Level Agreement (SLA) est un concept qui a été introduit dès les années 80 pour gérer la qualité de service (QdS) dans le domaine des télécommunications [WB10]. Le Service Level Agreement, que l'on pourrait traduire en français par **accord de niveau de service** ou **contrat de niveau de service**, est donc un contrat dans lequel on formalise la QdS prescrite entre un prestataire et un client.

Dans ce qui suit, nous détaillons en premier lieu les notions de base du SLA. Ensuite, nous expliquons le cycle de vie de SLA. Enfin, nous illustrons le SLA dans l'informatique utilitaire.

2.1.1 ABC du SLA

Cette section illustre les notions de base de SLA à savoir la terminologie, la définition et la structure de SLA.

i) Terminologie

Qualité de service (QdS) La qualité de service QoS (Quality of Service) désigne la capacité d'un service à répondre aux différentes exigences de ses utilisateurs en termes par exemple de disponibilité (e.g., taux de rejet), performance (e.g., temps de réponse), fiabilité (e.g., temps moyen entre deux pannes) ou coût (e.g., économique, énergétique). Les principales composantes de la qualité de service seront fournies par des métriques caractérisées par un type, une unité et une fonction de calcul.

Qualité de protection (QdP) Un attribut de qualité de protection QdP (Quality of Protection) décrit un aspect spécifique lié à la sécurité d'un service. A titre d'exemple, un attribut de QdP peut être : l'authentification, la réputation, l'emplacement des ressources.

Indicateurs clés de performance (PKI) Un indicateur clé de performance est une métrique permettant le suivi de variables de performance cruciales sur une durée.

Objectif de niveau de service (SLO) Un objectif de niveau de service SLO (Service Level Objective) est une clause de contrat. Elle exprime un engagement à maintenir un état particulier du service pendant une période donnée. L'état est défini comme une expression basée sur une QdS, QdP ou PKI.

ii) Définition

Contrat de niveau de service (SLA) Le SLA est un document qui définit la QdS et la QdP prescrite entre un prestataire et un client. Il spécifie un ou plusieurs **objectifs de niveau de service** SLO (Service Level Objective). En cas de violation, des pénalités sont appliquées. La pénalité non seulement pénalise le prestataire de services en réduisant leur profit, mais permet aussi aux utilisateurs de tolérer la défaillance du service.

Nous avons identifié plusieurs définitions de SLA par domaine. Nous citons en particulier la définition respectivement pour les domaines réseau [WB10], internet [RA01], service Web [jJMM+02] et centre de données [Mic02].

"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." [WB10]

"SLA constructed the legal foundation for the service delivery. All parties involved are users of SLA. Service consumer uses SLA as a legally binding description of what provider promised to provide. The service provider uses it to have a definite, binding record of what is to be delivered." [RA01]

"SLA is an agreement used to guarantee web service delivery. It defines the understanding and expectations from service provider and service consumer." [jJMM+02]

"SLA is a formal agreement to promise what is possible to provide and provide what is promised." [Mic02]

iii) Structure

Un contrat de niveau de service, comme tout contrat, contient un ensemble d'éléments (obligatoires et optionnels) à savoir la période de validité, les parties impliquées, les services, les garanties en terme d'objectifs, les pénalités et la suspension ou la résiliation du contrat et les sanctions [jJMM+02].

2.1.2 Cycle de vie

En 2001, Ron et al. [RA01] ont défini le cycle de vie de SLA par trois phases de haut niveau alors que Sun Microsystems Internet Data Center Group (2002) [Mic02] ont détaillé à grain plus fin le cycle de vie de SLA via six étapes.

i) Haut niveau : trois phases

Ron et. al. 2001 Ron et al [RA01] ont défini le cycle de vie de SLA par trois phases (création, fonctionnement, suppression) comme illustré dans la Figure 2.1. La phase de création consiste à trouver le fournisseur de service qui correspond aux besoins du client. Pendant la phase de fonctionnement, le client a un accès en lecture seule du SLA. Enfin, la phase de suppression, dans laquelle le SLA est résilié.

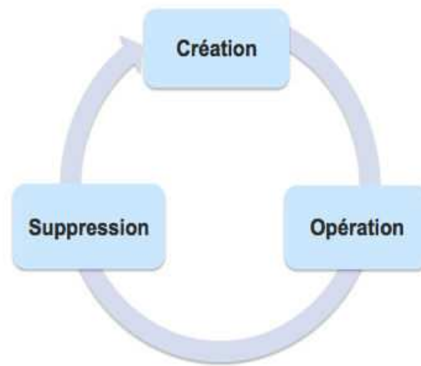


FIGURE 2.1 – Cycle de vie SLA (Ron et al. 2001)

ii) Détaillé : six phases

Sun Microsystems, 2002 : Un cycle de vie plus détaillé a été proposé par le groupe de Sun [Mic02], qui en comprend six phases à savoir : la découverte et la sélection du fournisseur, la définition du contrat (négociation), l'établissement du contrat, le monitoring des SLOs, la terminaison du contrat et les pénalités en cas de violation.

Les services sont sélectionnés selon les préférences du consommateur lors de la première étape. Une fois les services séléctionnés, il est nécessaire de définir les différents éléments du contrat. Ainsi, il est possible que les parties impliquées négocient quelques éléments du contrat. L'accord signé comprend la définition des services fournis, les garanties et les pénalités. Les différents paramètres de QdS sont mesurés par un système de monitoring. En cas de violations, les clauses de pénalité SLA correspondantes sont exécutées.

Par comparaison avec la définition de Rol et al. la découverte et la sélection du fournisseur, la définition du contrat (négociation) et l'établissement du contrat sont équivalents à la phase de création. Le monitoring des violations est comparable à la phase de fonctionnement. La phase de suppression regroupe la terminaison du contrat et les pénalités en cas de violation.

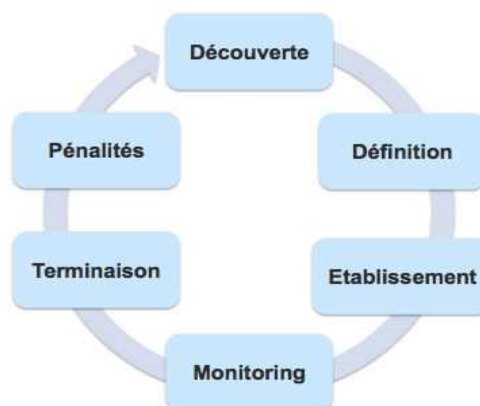


FIGURE 2.2 – Cycle de vie SLA (Sun 2002)

2.1.3 SLA @ informatique utilitaire

L'utilisation de SLA a commencé dès les années 80 pour gérer la QdS dans le domaine des télécommunications [WB10]. Cette section illustre l'utilisation de SLA dans l'informatique utilitaire en particulier la grille informatique, les fournisseurs de service d'application et l'informatique en nuage [Bas12].

i) Grille informatique - Grid computing

Le Grid computing [YB06] a contribué à l'émergence du SLA et en particulier avec ses services commerciaux sous-jacents. La QdS, en terme de temps de réponse, débit, disponibilité, ou encore la sécurité, est liée aux services de la grille, et pas seulement au réseau qui relie ces services. Les ressources peuvent être annoncées et commercialisées sous forme de services basés sur un SLA comme pour Globus [1]. Un tel SLA doit inclure des spécifications générales et techniques, y compris les politiques de prix et les propriétés des ressources nécessaires à l'exécution du service [SW07] [MC04].

Comme réponse au besoin de "Haute Disponibilité", les fermes de serveurs sont apparues. Une telle solution gère mieux la montée en charge et le passage à l'échelle.

ii) Fournisseurs de services d'application - ASP

En plus des services utilitaires qui allouent des ressources (Grid, ferme de serveurs), nous pouvons citer les fournisseurs de services d'application (ASP Application Service Provider) [DC03] qui allouent des applications métiers en ligne à savoir . Dans ce contexte, la nécessité du SLA est encore consolidée. Une large gamme d'applications est devenue disponible au grand public grâce au modèle ASP comme le service de paiement par carte de crédit.

iii) Cloud computing

Telle que l'imaginait John McCarthy, l'émergence du Cloud computing représente une suite logique dans l'histoire de l'informatique [GA99]. Ce modèle d'informatique utilitaire propose un nombre important d'offres. Par exemple plusieurs fournisseurs de Cloud proposent des services de calcul comme Amazon EC2, Microsoft Azure, Google App Engine, Go Grid, Rackspace... Un des éléments différenciateurs entre ces offres est le SLA proposé.

2.2 Informatique en nuage, les concepts de base

L'informatique en nuage, informatique dématérialisée, ou encore infonuagique (Cloud computing) est un nouveau modèle informatique qui consiste à proposer les services informatiques sous forme des services à la demande accessibles de n'importe où, n'importe quand et par n'importe qui. L'informatique en nuage consiste à dématérialiser les ressources informatiques et d'y accéder à l'aide d'un service, un concept qui est emprunté à l'architecture orientée services (SOA).

[1] http ://www.globus.org/

Dans ce qui suit, nous présentons en premier lieu la notion de service. Nous détaillons, ensuite, la définition de l'informatique en nuage. A la fin de la section, nous développons les modèles de l'informatique en nuage.

2.2.1 Notion de service

Les Web services constituent une approche pour mettre en œuvre le paradigme de service. Au début, ils ont été proposés par IBM, Microsoft, et par la suite par le consortium W3C (World Wide Web). Dans ce qui suit, nous détaillons la notion de service, l'architecture orientée services et l'architecture de composants de service.

i) Service

Un service doit être "abstrait" et n'est lié à aucune implémentation comme l'exemple d'un service de réservation de train. Il est encore vu comme étant un composant logiciel qui expose un nombre d'opérations offrant un traitement de bout en bout. Chaque service doit être indépendant des autres afin de garantir sa ré-utilisabilité et son interopérabilité. Le service ou composant peut être publié, découvert et invoqué à travers le Web grâce à l'utilisation d'Internet comme infrastructure de communication et d'XML en tant que format de données. Un service répond à un besoin métier identifié et représente une fonction contractualisée par une interface publiée via une infrastructure et rendue accessible aux applications. Il est défini par un contrat simple qui décrit ses entrées, son mode de fonctionnement et ses résultats. Un service est défini par quatre propriétés qui sont : l'autonomie, l'exposition d'un contrat, la communication par messages entre services et des frontières explicites entre les services [PH07].

ii) Architecture orientée services -SOA

"*SOA is an approach to designing software that dissolves business applications into separate 'services' that can be used independent of the applications of which they're a part and computing platforms on which they run*". Jay DiMare, IBM Global Services, 2006.

SOA (Services Oriented Architecture) est une architecture logicielle ouverte et extensible qui se base sur un ensemble de services simples dits aussi composants logiciels [ST05]. Ces services sont mis en œuvre avec une forte cohérence interne ou externe. L'objectif de l'architecture orientée services est : i) de décomposer une fonctionnalité en plusieurs tâches élémentaires appelées services ; ii) de décrire l'échange entre ces services. Lorsque l'architecture SOA s'appuie sur des web services, on parle alors de WSOA (Web Services Oriented Architecture). Pour fonctionner correctement, un service Web repose sur des normes basées sur XML :

- SOAP (Simple Object Access Protocol) : simple protocole d'échange des données pour le transport d'informations et l'infrastructure de communication,

- WSDL (Web Service Description Language) : permet de décrire un service Web. Il définit la grammaire XML pour décrire des services comme des ensembles de paramètres de communication capables d'échanger des messages, et est normalisé par le W3C,

- UDDI (Universal Discovery, Description, and Integration) : annuaires pour le référencement des services par les fournisseurs et leur découverte par les utilisateurs. UDDI est un registre ouvert à tout le monde et est normalisé par l'OASIS.

- BPEL (Business Process Execution Language) : dit aussi le chef d'orchestre, il décrit le processus métier par combinaison de plusieurs Web services.

iii) Architecture de composants de service -SCA

SCA (Service Componant Architecture) [cas13] est une spécification d'implémentation des composants d'une architecture orientée services (SOA) visant à simplifier la création et la composition des services indépendamment de leur implémentation. L'originalité de SCA vise à étendre l'approche orientée service et à découpler les problématiques liés à la conception d'application, l'association de services implémentés dans des langages différents et à l'assemblage et de déploiement de composants.

SCA repose sur la notion de composant comme étant un élément de base. Le composant représente alors l'unité la plus petite. Il possède des fonctionnalités qui seront exposées pour qu'elles soient utilisées par d'autres composants. Ces fonctionnalités sont appelées services et sont décrites via des contrats de service implémentés par le composant même. La SCA repose aussi sur la notion de composé comme étant un autre élément de base. Le composé est un regroupement de composants, services, références, propriétés et des liens qui relient ces éléments entre eux. Un composé est alors un composant de haut niveau car il fournit des services, dépend de références et possède des propriétés, et il peut être référencé par d'autres composants et utilisé au sein d'autres composés.

2.2.2 Qu'est-ce que l'informatique en nuage ?

Cette section est dédiée à la présentation de l'informatique en nuage. Nous commençons par la chronologie du Cloud. Ensuite, nous illustrons la définition du Cloud. Enfin, nous clarifions les caractéristiques principales du Cloud.

i) Chronologie

Le concept remonte à 1960, lorsque John McCarthy a estimé que le calcul peut un jour être organisé comme un service d'utilité publique [GA99].

"*If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility... The computer utility could become the basis of a new and important industry.*" John McCarthy, speaking at the MIT Centennial in 1961.

Le Cloud est la 5e génération d'infrastructures informatiques [BYV+09] : mainframe (1970), client-serveur (1980), web (1990), SOA (2000) et Cloud (20 ? ?). Il n'y a pas de date-clé à laquelle nous puissions dire que le Cloud computing est né. Amazon a dévoilé sa version d'essai d'Elastic Computing Cloud (EC2) le 24 août 2006. Une telle date peut être considérée comme date de naissance. L'expression est devenue populaire en 2007.

La notion de Cloud fait référence à un nuage, tel que l'on a l'habitude de l'utiliser dans des schémas techniques lorsque l'on veut représenter Internet. Le Cloud computing met en œuvre l'idée d'informatique utilitaire du type service public, proposée par John McCarthy. Un tel concept peut être comparé au cluster de calcul ou encore à l'informatique en grille.

ii) Définition

L'informatique en nuage est une métaphore désignant un réseau de ressources informatiques (logicielles et/ou matérielles) accessibles à distance par le biais des technologies Internet [MGG11] [AFG+10] [VRMCL08] [ZCB10] [BYV+09] [AFG+09]. C'est un modèle informatique selon lequel des ressources informatiques sont fournies sous la forme d'un service à la demande. Ainsi les services de l'informatique en nuage reposent sur un concept de paiement à l'utilisation. C'est un concept qui est comparable à celui de la distribution de l'énergie électrique. La puissance de calcul et de stockage de l'information est proposée à la consommation par des fournisseurs de l'informatique en nuage. De ce fait, l'utilisateur n'a plus besoin des serveurs propres, mais confie cette ressource à un fournisseur qui lui garantit une puissance de calcul et de stockage à la demande. L'idée, donc, est de louer les technologies de l'information. L'utilisateur peut bénéficier d'une flexibilité importante avec un effort minimal de gestion.

De nombreuses définitions ont été données du Cloud Computing. Nous citons deux définitions : la première de NIST [MGG11] et la deuxième de Buyya [Buy09] :

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. NIST 2011 [MGG11].

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. Buyya 2009 [Buy09].

iii) Caractéristiques principales

L'informatique en nuage est une nouvelle façon de délivrer les ressources informatiques, et non une nouvelle technologie. Il se caractérise par [MGG11] :

Un accès en libre service à la demande : un client pourra commander des ressources informatiques en fonction de ses besoins. Les ressources informatiques sont fournies d'une manière entièrement automatisée et c'est le client, au moyen d'une interface, qui met en place et gère la configuration à distance.

Un accès ubiquitaire au réseau : les capacités sont disponibles sur le réseau et accessibles par des mécanismes standards, qui favorisent l'accès au service par des clients lourds ou légers via des plates-formes hétérogènes.

Une mise en commun des ressources : les ressources informatiques sont mises à la disposition des clients sur un modèle multi-locataires, avec une attribution dynamique des ressources physiques et virtuelles en fonction de la demande. Le client n'a généralement ni contrôle, ni connaissance sur l'emplacement des ressources, mais est en mesure de le spécifier à un plus haut niveau d'abstraction (pays, état ou centre de données). Ce partage des ressources est la caractéristique qui différencie le Cloud Computing de l'infogérance.

Une élasticité rapide : les capacités informatiques mises à disposition du client peuvent être ajustées (augmenter ou diminuer) rapidement (quelques minutes voire quelques secondes) en fonction des besoins et/ou de la charge et de façon automatique dans certains cas. L'utilisateur a l'illusion d'avoir accès à des ressources illimitées bien que le fournisseur définit toujours un seuil (par exemple : 20 instances par zone est le maximum possible pour Amazon EC2).

Un service mesuré en permanence : Les ressources consommées sont contrôlées et communiquées au client et au fournisseur de service de façon transparente. Cela garantit un niveau de disponibilité adapté aux besoins spécifiques des clients.

Pour conclure, la somme de ces cinq caractéristiques permet aujourd'hui de dire si un service proposé est vraiment de type Cloud computing ou non. Maintenant que nous avons cerné les caractéristiques d'un service Cloud, nous détaillons dans ce qui suit les acteurs, les modèles de services existants et leur mode de déploiement.

2.2.3 Modèles de l'informatique en nuage

Cette section expose les modèles du Cloud. D'abord, nous étudions les modèles de service. Ensuite, nous expliquons les modèles de déploiement. Enfin, nous introduisons les acteurs du Cloud.

i) Modèles de service

XaaS (X as a Service) est à la base du paradigme de l'informatique en nuage. Nous dénombrons 3 modèles de service [MGG11] comme illustré dans la Figure 2.3 : i) Logiciels en tant que service SaaS (Software as a Service) où le matériel, l'hébergement, le framework d'application et le logiciel sont dématérialisés, ii) Plateforme en tant que service PaaS (Platform as a Service) où le matériel, l'hébergement et le framework d'application sont dématérialisés, et iii) Infrastructure en tant que service IaaS (Infrastructure as a Service) où seul le matériel (serveurs) est dématérialisé.

Le SaaS est l'ensemble des logiciels mis à disposition via une interface Web ou via des API de type Services Web. Parmi les exemples les plus utilisés, on trouve des messageries comme Gmail, ou des applications telles que la CRM (Customer Relationship Management), voire les réseaux sociaux tels que le Facebook. Le SaaS représente le niveau optimal de délégation des responsabilités. Le fournisseur délivre un produit fini. L'utilisateur final ne requiert alors aucune connaissance informatique particulière. Le PaaS désigne une plateforme d'exécution hébergée par un fournisseur. Cette plateforme peut être utilisée pour gérer le cycle de vie du SaaS, et peut aussi être mise à la disposition des entreprises qui souhaitent faire héberger leurs applications issues de

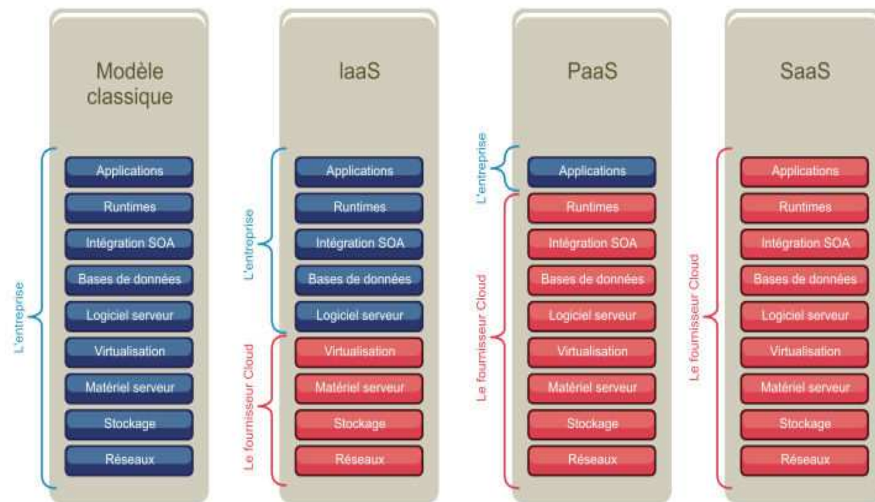


FIGURE 2.3 – Service XaaS

développements spécifiques. Les principaux acteurs sont Google (Google App Engine) [Eng13a] et Microsoft (Windows Azure) [Azu13b]. Le IaaS s'adresse aux ingénieurs systèmes. Il désigne l'ensemble des composants d'une infrastructure technique déportée dans les nuages. Le IaaS se traduit principalement par des environnements d'exécution virtualisés. Le principle acteur de IaaS est actuellement Amazon EC2 [EC213a].

ii) Modèles de déploiement

L'informatique en nuage marque une nouvelle avancée vers l'infrastructure informatique élastique. Il existe quatre principaux modèles de déploiement des services en nuage : nuage privé, nuage communautaire, nuage public et nuage hybride [MGG11].

Cloud privé : nuage réservé à l'usage exclusif d'une seule organisation. Il peut être possédé, géré et opéré par cette organisation, un intervenant extérieur ou une combinaison des deux. Généralement il est situé dans les locaux de l'organisation.

Cloud communautaire : nuage réservé à l'usage d'une communauté spécifique de consommateurs partageant des intérêts communs. Il peut être possédé, géré et opéré par un ou plusieurs organismes participant à la communauté, un intervenant extérieur ou une combinaison d'entre eux. Il est situé dans les locaux des organismes participant ou dans ceux d'un hébergeur externe.

Cloud public : nuage externe à l'entreprise, accessible via Internet ou un réseau privé, géré par un opérateur externe propriétaire des infrastructures, avec des ressources totalement partagées entre tous ses clients.

Cloud hybride : nuage résultat d'une conjonction de deux ou plusieurs Clouds différents (public, privé ou communautaire) amenés à «coopérer», à partager entre eux les applications et les données.

iii) Acteurs

La Figure 2.4 présente un aperçu de l'architecture de référence de l'informatique en nuage selon NIST [FJJ+11]. Il identifie les principaux acteurs et leurs activités. Il y a cinq acteurs :

- consommateur (Cloud Consumer) : une entité qui utilise un service fourni par un fournisseur,
- fournisseur (Cloud Provider) : une entité chargée de rendre un service à la disposition des parties intéressées,
- auditor (Cloud Auditor) : une entité qui peut procéder à une évaluation indépendante des services : la performance et la sécurité du nuage,
- courtier (Cloud Broker) : une entité qui gère l'usage, la performance et le provisionnement de services, et qui négocie la relation entre les fournisseurs et les consommateurs,
- carrier (Cloud Carrier) : un intermédiaire qui fournit la connectivité et le transport des services des fournisseurs aux consommateurs.

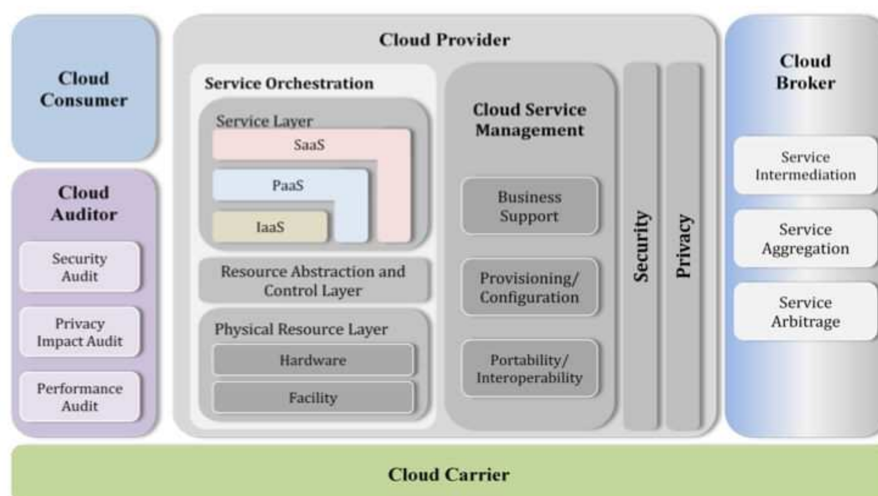


FIGURE 2.4 – Référence NIST 2011 [FJJ+11]

2.3 Informatique en nuage, les ressources virtuelles

Dans cette section, nous étudions les ressources virtuelles du Cloud. En premier lieu, nous introduisons le concept de virtualisation. Ensuite, la modélisation des ressources virtuelles est developpée. Enfin, nous détaillons le modèle économique sous-jacent.

2.3.1 Virtualisation

Le concept de virtualisation [SN05] [Gol73] a vu le jour pendant les années 1960 et permet de faire fonctionner nombreux systèmes d'exploitation et/ou plusieurs applications sur un seul serveur. Ces derniers doivent être indépendants et autonomes. La

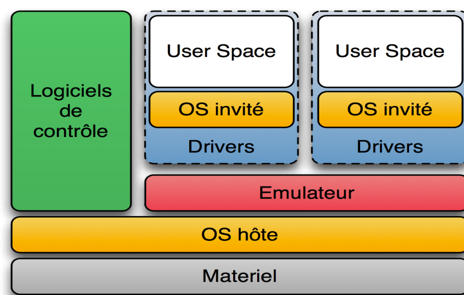


FIGURE 2.5 – Machine virtuelle

virtualisation apporte de très nombreux avantages tels que : i) l'utilisation optimale des ressources existantes et ii) l'économie sur le matériel par mutualisation. Une infrastructure virtuelle permet de réduire les coûts informatiques tout en augmentant l'efficacité, le taux d'utilisation et la flexibilité des actifs existants.

Dans ce qui suit, nous présentons dans l'ordre : la définition d'une machine virtuelle (VM) et les types de VMs proposés par les fournisseurs IaaS.

i) Machine virtuelle - VM

Une VM (voir Figure 2.5)[2] est un conteneur de logiciels totalement isolé, capable d'exécuter ses propres systèmes d'exploitation et applications, à l'instar d'un ordinateur physique. Une machine virtuelle se comporte exactement comme un ordinateur physique. Elle contient un processeur, une mémoire RAM, un disque dur et une carte d'interface réseau virtuels (autrement dit, basés sur des logiciels) qui lui sont propres [SN05].

ii) Plusieurs types de machines virtuelles

Amazon est un pionnier dans le domaine du Cloud computing et le leader actuel du marché. Dans ce travail, Amazon est utilisé comme une référence ainsi qu'une infrastructure de test et de validation.

Amazon Elastic Compute Cloud ou Amazon EC2 [EC213a] est un Cloud public fondé sur les produits de virtualisation XEN et qui permet l'hébergement de machines virtuelles nommées instances. On y repère 12 types d'instances (VM) allant de la « Micro Instance » à la « Cluster Compute Quadruple Extra Large », termes correspondants à leurs capacités techniques (puissance de calcul, mémoire vive, mémoire interne, etc.). Amazon met à disposition un catalogue de machines virtuelles prêtes à l'emploi, nommées les « Amazon Machine Images » (AMI), et parmi lesquelles nous trouvons une version de Windows ou une distribution de Linux. Les AMIs peuvent également contenir des versions pré-packagées avec la couche middleware déjà installée comme MySQL ou Apache.

Également Microsoft Azure [Azu13b] offre 7 types d'ordinateur virtuel (XS, S, M, L, XL, A6, A7) combinés avec deux systèmes d'exploitation Microsoft et Linux.

[2]http ://fr.wikipedia.org/wiki/Fichier :DiagrammeArchiEmulateur.png

2.3.2 Modélisation des ressources virtuelles

Suite à l'étude des travaux courants, nous avons remarqué l'absence de norme ou standard du Cloud Computing malgré l'effort de quelques acteurs [Clo13a]. Nous citons en particulier NIST qui aborde la définition du Cloud, DMTF (Distributed Management Task Force)[3] qui se focalise sur le format de virtualisation, OCCI (Open Cloud Computing Interface)[4] qui se concentre sur la modélisation d'une interface pour le Cloud ou encore SNIA (Storage Networking Industry Association)[5] qui traite le stockage.

Nous remarquons le manque d'interopérabilité entre solutions. Des nombreux travaux abordent sous différents angles la standardisation du Cloud computing. Cependant, ces standards ne touchent pas tous les couches XaaS de manière équitable. La plupart des travaux se concentrent sur le IaaS mais très peu voire aucune tentative pour la standardisation de PaaS. Cette couche supporte jusqu'à aujourd'hui plusieurs définitions. Alors que le niveau SaaS est traité partiellement avec des standards liés au SOA.

i) Amazon EC2

Le modèle de ressources [drA13] proposé par Amazon est présenté par la Figure 2.6. L'image est une template prédéfinie pour instancier une VM. Le type d'une instance définit les caractéristiques d'instance à savoir : CPU, RAM, disque dur. Une instance est une image déployée sur un type d'instance. Le stockage est la composante où les données sont stockées. Le réseau relie de nombreuses instances. L'adresse IP est une adresse statique qui peut être reliée à n'importe quelle instance. La zone de disponibilité est l'endroit où l'infrastructure est localisée. Chaque zone de disponibilité est isolée des pannes d'autres zones.

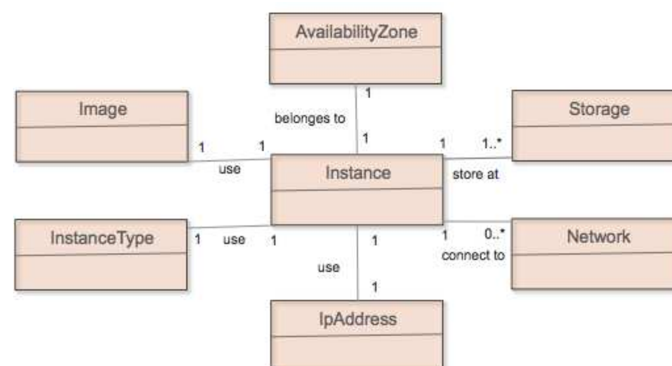


FIGURE 2.6 – Modèle de ressource Amazon EC2 [drA13]

ii) Oracle

Le modèle proposé par Oracle (voir Figure 2.7) présente les hiérarchies entre les ressources ainsi que les relations [Ora10]. Par exemple la ressource "Cloud" est un ensemble des centres des données virtuelles "VDC". Chaque VDC possède plusieurs réseaux virtuels "VNet". Ce dernier a plusieurs interfaces réseaux.

[3]http ://www.dmtf.org/
[4]http ://occi-wg.org/
[5]http ://www.snia.org/

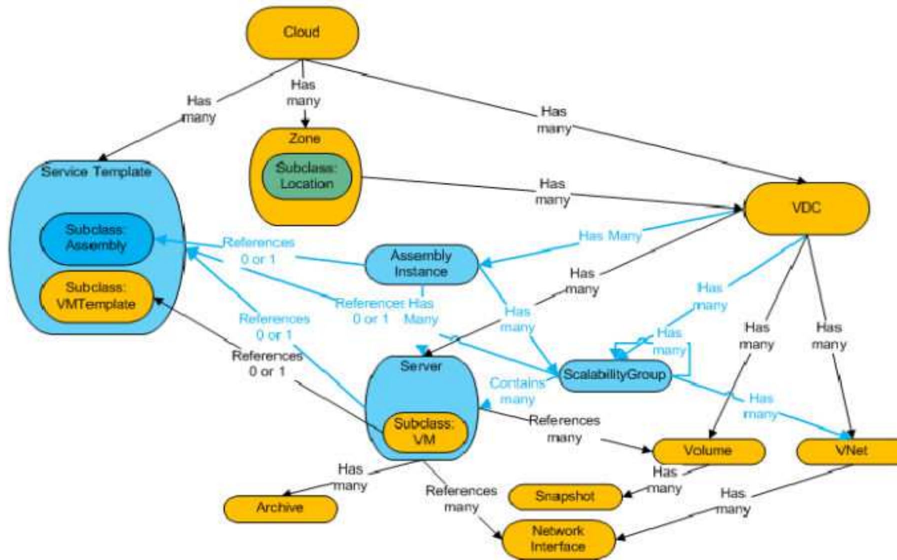


FIGURE 2.7 – Modèle de ressource Oracle [Ora10]

iii) DeltaCloud

La Figure 2.8 illustre le modèle de ressource de DeltaCloud [drD13]. Le profil matériel définit des aspects tels que le stockage, RAM et l'architecture disponible. L'image, n'est pas directement exécutable, mais représente une template pour la création des instances. Une instance est une machine concrète réalisée à partir d'une image. Le "Realms" présente le domaine (centre de données, pools) ou la localisation géographique des ressources.

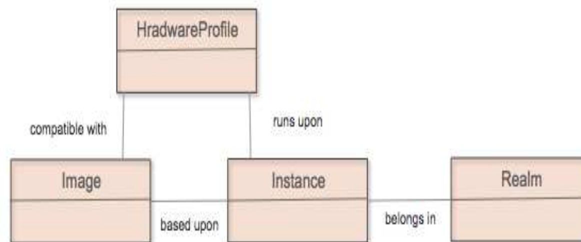


FIGURE 2.8 – Modèle de ressource Deltacloud [drD13]

iv) OCCI

OCCI est une spécification pour la gestion à distance des infrastructures de Cloud, permettant le développement d'outils interopérables pour le déploiement, le dimensionnement et la surveillance. Les concepts du modèle sont le calcul, le stockage et le réseau [OCC10] . Ces concepts ainsi que leurs relations sont représentées dans la Figure 2.9.

Le Tableau 2.1 résume les modélisations des ressources virtuelles issues de différents acteurs : fournisseur IaaS comme Amazon EC2 et Oracle, open-source comme DeltaCloud et des groupes comme OCCI.

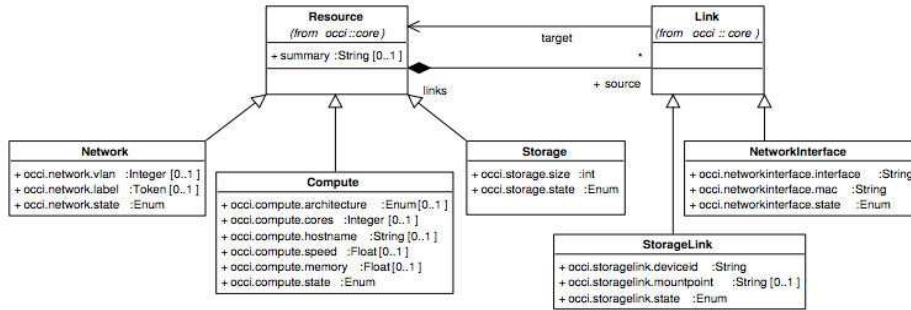


FIGURE 2.9 – Modèle de ressource OCCI [OCC10]

TABLE 2.1 – Modèles des ressources

	instance	calcul	stockage	réseau	image	cloud	data center virtuel	zone	type d'instance
Oracle	x	x	x	x	x	x	x	x	x
EC2	x	x	x	x	x			x	x
OCCI	x	x	x	x					
δ Cloud	x				x				

2.3.3 Modèle économique sous-jacent

Le modèle économique de Cloud est basé sur l'économie d'échelle. Dans ce qui suit, nous étudions l'économie d'échelle, la facturation et le temps d'initialisation d'une VM.

i) Economique d'échelle

Nous avons vu précédemment que l'élasticité rapide [NRSR13] est une des caractéristiques principales du Cloud. La Figure 2.10[6] clarifie l'avantage économique de l'élasticité. Proposée par Amazon EC2, cette figure est devenue une icône du Cloud. L'élasticité permet de suivre de près la courbe de demande. Les capacités informatiques mises à disposition du client peuvent être provisionnées et libérées rapidement de façon automatique en fonction des besoins et/ou de la charge afin d'éviter une sous/sur-charge et absorber les variations.

La mutualisation des capacités informatiques, leur lissage et leur meilleure utilisation autorisent des économies d'échelles. En fournissant des services en nuage simultanément à de multiples utilisateurs, le fournisseur peut offrir ses services à un prix moins élevé que si les utilisateurs s'organisaient eux-mêmes. En outre, le Cloud offre aux clients un accès facile et une disponibilité immédiate pour répondre à leurs besoins. Il permet l'absorption des pics, la possibilité de croissance rapide, mais aussi la professionnalisation de l'exploitation.

[6]http ://aws.amazon.com/fr/economics/

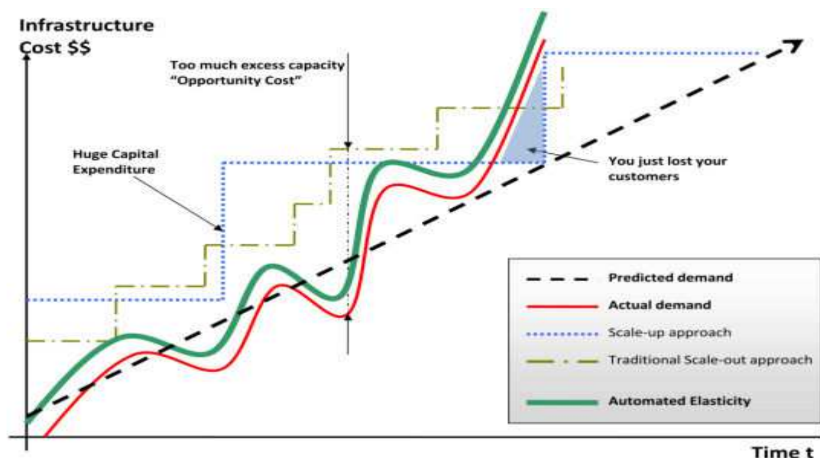


FIGURE 2.10 – Elasticité

ii) Facturation

Le modèle de facturation, selon la formule « pay as you go », est l'une des clés du Cloud Computing. Les offres Cloud tendent à multiplier, à l'instar des pratiques de la téléphonie mobile, les options, forfaits, extensions, etc. 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. Pour Amazon, la tarification commence à partir du moment où une instance est lancée jusqu'à ce qu'elle soit terminée ou arrêtée (voir Figure 2.11). Chaque heure d'instance partielle consommée sera facturée comme une heure pleine [EC213a]. L'instance arrêtée est facturée selon le service de stockage (Elastic Block Store (EBS)). Quelques cas particuliers tels que Windows Azure qui facture l'utilisation par l'heure naturelle (fixe) [Azu13b], c'est à dire, si une instance est démarrée à 08h55 et s'est arrêtée à 09h05, l'utilisateur devra payer 2 heures. En plus du temps de calcul mesuré en heure d'instance, le prix évolue en fonction de type des instances, la quantité de stockage mesuré en Gigaoctet, le nombre de transactions représentant le nombre d'accès aux services de stockage et la bande passante en entrée comme en sortie.

En plus de modèle « pay as you go », nous distinguons différents modèles de prix. Amazon EC2 [EC213a], par exemple, propose à la fois un modèle de prix statique (instances à la demande et les instances réservées) et un modèle de prix dynamique (instances ponctuelles). Avec les instances à la demande, les clients paient la capacité de calcul à l'heure, sans engagement à long terme. Les instances réservées permettent d'effectuer un seul paiement, peu élevé, pour chaque instance à réserver, et de recevoir, en retour, une réduction significative sur les frais horaires d'utilisation de cette instance. Les instances ponctuelles (spot instances) permettent à un client de faire une offre sur la capacité Amazon EC2 non utilisée et d'exécuter ces instances aussi longtemps que leur offre dépasse le prix ponctuel actuel. Le prix ponctuel change périodiquement en fonction de l'offre et de la demande, et les clients, dont les offres répondent ou dépassent ce prix, ont accès aux instances ponctuelles disponibles. L'application tolérante aux pannes, qui peut surmonter une interruption potentielle si leur demande Spot est surenchérie, est bien adaptées pour tirer avantage du bas prix et de l'élasticité des instances ponctuelles.

Microsoft Azure [Azu13b] propose également le modèle de paiement à l'utilisation

(instances à la demande d'Amazon). De plus, Microsoft propose des abonnements (instances réservés d'Amazon) de six et douze mois qui offrent jusqu'à 29,5 % d'économie.

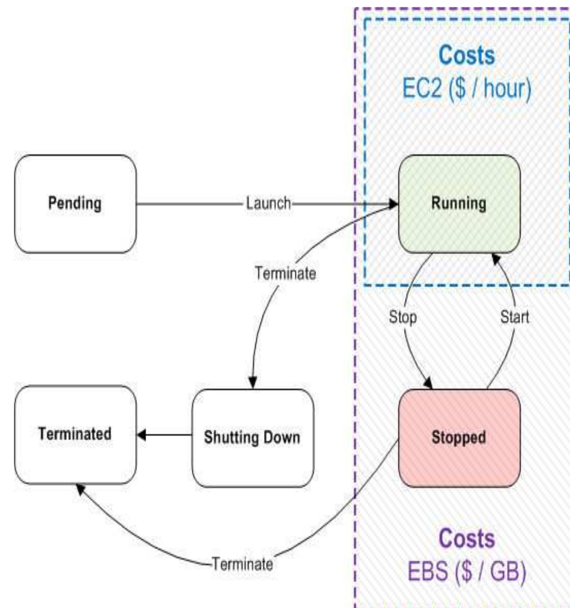


FIGURE 2.11 – Modèle économique Amazon EC2 [EC213a]

iii) Temps d'initialisation d'une instance

Le temps d'initialisation d'une instance est généralement de quelques minutes. Cette période dépend d'une certaine quantité de facteurs incluant la taille de la AMI, le nombre d'instances lancées et le temps écoulé depuis la dernière fois que la AMI a été utilisée. L'image lancée pour la première fois peut prendre légèrement plus longtemps pour démarrer. Une heure-instance est facturée dès que l'instance est en état de "fonctionnement".

Une étude comparative [MH12] montre que les instances Windows de Amazon EC2 prennent environ 9 fois plus longtemps que les instances Linux, alors que les instances Azure présentent des performances similaires. Le temps d'initialisation des instances Azure varie avec le type de l'instance. Cependant, ce n'est pas le cas avec Amazon EC2.

2.4 Informatique autonomique

Afin de faire face à un environnement hautement dynamique comme le Cloud, il est indispensable de s'appuyer sur des modèles qui permettent de réagir au contexte d'exécution afin de garantir la performance et la fiablité du Cloud. Un système autonomique est un système qui s'administre lui-même et n'aura pas besoin d'une intervention humaine. Le fonctionnement d'un système autonomique imite celui d'un système nerveux dans la reproduction de quelques comportements lui permettant de rester en bon état.

Dans ce qui suit, nous illustrons la définition de l'informatique autonomique. Ensuite, nous détaillons les propriétés autonomiques. Enfin, nous introduisons la boucle de contrôle autonomique.

2.4.1 Définition

L'informatique autonomique est une tentative initiée en 2001 par IBM comme réponse à la difficulté d'administration des systèmes complèxes. L'objectif est de soulager l'opérateur humain en l'assistant dans les tâches d'administration, et, ultimement, de le remplacer partout ou cela est possible [Hor01].

"Autonomic computing is the ability of an IT infrastructure to adapt to change in accordance with business policies and objectives." IBM 2003 [KC03].

A l'instar du corps humain qui intègre différents systèmes de régularisation comme le système d'optimisation qui contrôle le rythme cardiaque en fonction de l'effort auquel il est soumis, un système autonome doit comporter des propriétés autonomiques telles que définies par kephart et Chess [KC03] ainsi que Brantz [BBC+03] en 2003. Un système auto-géré ou "self-managed" doit implémenter ces quatres paradigmes : "self-healing", "self-protection", "self-configuration" et "self-optimization".

2.4.2 Propriétés autonomiques

Les quatres propriétés autonomiques sont présentées dans cette section [KC03] [BBC+03] [HM08].

i) Auto-guérison - "self-healing"

Auto-guérison réfère à la capacité à détecter, diagnostiquer puis compenser ou réparer des pannes survenant dans le système. Ce dernier peut diagnostiquer les erreurs ou les situations qui peuvent se produire au cours du fonctionnement. Il peut aussi faire une réparation et ce de manière à ce que son activité soit la moins perturbée possible. Un système est dit alors "self-healing" quand il est capable de garder son activité dans un état stable en dépassant les événements gênants dus à la défaillance de ses éléments.

ii) Auto-protection - "self-protection"

Auto-protection est l'aptitude à protéger le système contre des actions qui peuvent le déstabiliser et le rendre inactif. La "self-protection" permet à un système d'assurer la confidentialité et la protection de ses données et ajuster son comportement dont le but est sa survie. Cette propriété doit lui permettre de réagir à tous types de menaces et anticiper les attaques dans le but de prendre les précautions adéquates.

iii) Auto-configuration - "self-configuration"

Auto-configuration est le pouvoir de déterminer et appliquer un paramétrage ou une configuration permettant au système de fonctionner correctement en répondant à un objectif particulier. Cette propriété permet à un système de se (re)paramétrer et de se mettre à jour tout seul ou via des paramètres pré-définis par l'être humain.

iv) Auto-optimisation - "self-optimisation"

Auto-optimisation est la force d'assurer des niveaux de performances face à l'évolution de l'état du système. Cette propriété permet au système d'adapter un paramétrage

et/ou une configuration optimale afin d'éviter des états comme la surcharges ou la sous-charges. L'optimisation se fait en respectant des critères définis généralement par l'être humain.

2.4.3 Boucle de contrôle autonomique

IBM a proposé un modèle de référence [Hor01] appelée la boucle MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) début des années 2000. Ce modèle est inspiré des boucles de contrôles utilisées en automatique dans le but d'atteindre les objectifs d'une informatique autonomique. Il est illustré par la Figure 2.12 [KC03].

Dans ce qui suit, nous présentons le gestionnaire autonomique qui gère la boucle MAPE-K. Ensuite, nous détaillons les éléments de la boucle de contôle.

i) Gestionnaire autonomique

Le gestionnaire autonomique est un logiciel configuré manuellement en utilisant les politiques qui forment une description de haut niveau de la stratégie posée par l'administrateur. Le gestionnaire autonomique fonctionne en utilisant les informations collectées par les capteurs. Il observe les éléments gérés et par la suite, il exécute les changements à l'aide des actionneurs. Ce gestionnaire utilise, en plus des informations collectées, sa connaissance interne pour exécuter toutes les actions de bas niveau dont le but d'atteindre l'objectif de la stratégie, en se basant sur les politiques décrites par l'administrateur.

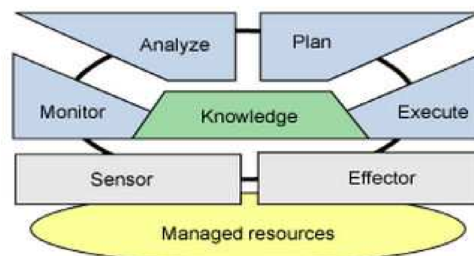


FIGURE 2.12 – boucle de contrôle autonomique MAPE-K [KC03]

ii) MAPE-K

L'observation ou "monitoring" : La boucle MAPE-K présente une tâche d'observation qui apporte des mécanismes de capture des propriétés de l'environnement, qui sont importantes pour la gestion autonomique du système. Le gestionnaire autonomique se base sur les informations nécessaires pour la reconnaissance du fonctionnement anormal des éléments du système, ensuite il les interprète. En conséquence, les capteurs (ou sondes) assurent une mise en forme des informations remontées.

L'analyse : La tâche d'analyse compare les résultats donnés par les capteurs (les informations d'observation) avec les politiques décrites par l'administrateur. Cette comparaison montre la différence entre l'état d'exécution fonctionnel réel du système et celui attendu. L'objectif est alors de prendre une décision sur le changement à apporter au système.

La planification : La tâche de planification de la boucle de contrôle autonomique produit une ensemble de changements à réaliser sur les éléments gérés. Ces changements dépendent de la décision prise par la tâche d'analyse précédente. Cette tâche planifie aussi de manière temporelle les changements à apporter au système. En effet, il peut y avoir des priorités ou des contraintes temporelles qui ralentissent les changements.

L'exécution : La tâche d'exécution se charge de respecter la planification des changements et d'exécuter en temps voulu les actions nécessaires sur les éléments gérés. Cette tâche peut être en charge de distribuer des actions sur plusieurs machines si un changement fait intervenir plusieurs éléments gérés.

La base de connaissances : La base de connaissances (knowledge) relie les différentes tâches, détaillées précédemment, de la boucle MAPE-K. Elle découle de différentes sources, automatique ou même manuelle, collectant l'information. Les quatre tâches détaillées précédemment peuvent échanger avec la base de connaissances en cherchant/rajoutant des informations dans cette base. D'où l'utilité de déterminer un processus ou une méthode de représentation/manipulation de ces données de la base.

2.5 Conclusion

Dans ce chapitre, nous avons présenté les concepts et les technologies sous-jacentes à notre sujet. Nous avons commencé par présenter les notions de base associées au contrat de niveau de service (Accord de niveau de service - SLA) ainsi que son cycle de vie. De plus, nous avons developpé l'utilisation de SLA dans l'informatique utilitaire. Ensuite, nous avons détaillé le paradigme Cloud computing en étudiant les concepts de base et les ressources virtuelles. Afin de faire face à cet environnement hautement dynamique, il devient impératif de se baser sur l'informatique autonomique. Ce modèle est présenté à la fin du chapitre.

3

Gestion des contrats de niveaux de service

L'objectif de ce chapitre est de recenser et d'analyser plusieurs travaux liés au problème de définition et gestion de contrats de niveaux de service SLA dans le Cloud. Dans un premier temps, nous nous interrogeons sur l'expression même de ces contrats SLA : comment les définir, comment exprimer les pénalités, les langages existants sont-ils adaptés au Cloud ? Ensuite, nous étudions la problématique de la définition de contrats SLA entre les différentes couches XaaS du Cloud et donc de l'inter-dépendances de ces contrats. En effet, un contrat entre deux couches découle des contrats des couches sous-jacentes, il y a donc nécessité de négociation ou de calibrage pour proposer des valeurs réalistes d'objectifs SLA. Enfin, une fois défini et validé en terme de valeurs un contrat SLA dans la "pile" du Cloud, il reste à garantir qu'il ne sera pas violé à l'exécution. La gestion dynamique de capacité de ressources est l'assurance qu'un service répondra à la demande avec un niveau spécifié de qualité. Aussi, nous étudions l'état de l'art en la matière avec un focus en particulier sur le dimensionnement automatique (Auto-scaling). A la fin du chapitre, comme bilan, nous identifions les limitations et mettons en évidence les contributions à apporter.

Sommaire

3.1 Accord de niveau de service (SLA), la définition 32

3.1.1 SLA @ service Web 32

3.1.2 SLA @ Cloud 36

3.1.3 SLA niveau IaaS 39

3.1.4 Discussion 41

3.2 Accord de niveau de service (SLA), les dépendances 43

3.2.1 Les dépendances, les concepts 43

3.2.2 Les dépendances, la pratique 45

3.2.3 Discussion 47

3.3 Gestion de Capacité des ressources 48

3.3.1 Dimensionnement automatique, les concepts 48

3.3.2 Dimensionnement automatique, la pratique 54
3.3.3 Autres approches de gestion de capacité 61
3.3.4 Discussion 64
3.4 Synthèse 66

3.1 Accord de niveau de service (SLA), la définition

De nombreuses solutions portent sur la définition du SLA. Nous reportons ici seulement les travaux directement liés aux services Web puis aux services Cloud. A la fin de la section, nous présentons un bilan des travaux.

3.1.1 SLA @ service Web

Les travaux sur le SLA se sont développés avec l'apparition des architectures orientées services SOA (Service-Oriented Architecture) où le but était d'assurer la QdS des services Web. Plusieurs approches ont été proposées pour gérer le contrat de niveau de service dans les architectures SOA : Web Service Level Agreement (WSLA) [LF03], Web Services Agreement (WS-Agreement) [Aa07], Service Level Agreement Language (SLAng) [LSEO3], Web Service Offerings Language (WSOL) [TPE+02] ou encore Rule-based Service Level Agreement(RBSLA) [Pas05].

Dans ce qui suit nous détaillons WSLA et WS-Agreement qui reprèsentent les résultats les plus aboutis. Ensuite, les autres travaux sont présentés brièvement.

i) Web Service Level Agreement (WSLA)

WSLA [LF03] [KL03] est un framework développé par IBM pour spécifier les SLA pour les services Web. La spécification WSLA permet la définition et le suivi des SLA. Elle est basée sur XML. WSLA est décrit par un méta modèle (voir Figure 3.1). Ce dernier offre la possibilité d'enrichir le répertoire des critères de performance en créant des nouvelles métriques. La dernière version de la spécification WSLA a été publiée en 2003 [LF03]. Depuis, WSLA est intégré dans WS-Agreement.

La structure de WSLA contient trois sections : une section décrit les parties, une section contient une ou plusieurs définitions de services et une section définit les obligations. WSLA supporte deux types d'acteurs : les signataires, à savoir le fournisseur de services et le client, et les parties de support (tiers de confiance). Comme un accord représente une relation bipartite, le nombre des signataires est limité à deux, alors que le nombre des parties de support est théoriquement infini. Une définition de service contient un ou plusieurs objets de service. Un objet du service est une abstraction d'un service (par exemple, un service de réservation en ligne). Un objet de service peut avoir une ou plusieurs "*SLAParameters*" pour définir les garanties associées. Chaque "*SLAParameter*" est défini par une métrique. Cette dernière est calculée en définissant une directive de mesure ou une fonction. La section des obligations contient deux types d'obligations : un objectif de niveau de service (SLO) et une garantie d'action. Un SLO est une garantie d'un état particulier de paramètres de SLA dans une période de temps donnée. Alors qu'une garantie d'action est la promesse de faire quelque chose dans

une situation définie. Un exemple : une notification est envoyée si un objectif de niveau de service n'est pas respecté. Chaque obligation a une partie obligée.

Malgré l'arrêt de spécification et une dernière version de 2003 semi-stable, WSLA reste une des spécifications la plus mature pour la définition d'un SLA. Cette spécification couvre tout le cycle de vie de SLA. Toutefois, la gestion de pénalité est limitée principalement à la notification en cas de violation.

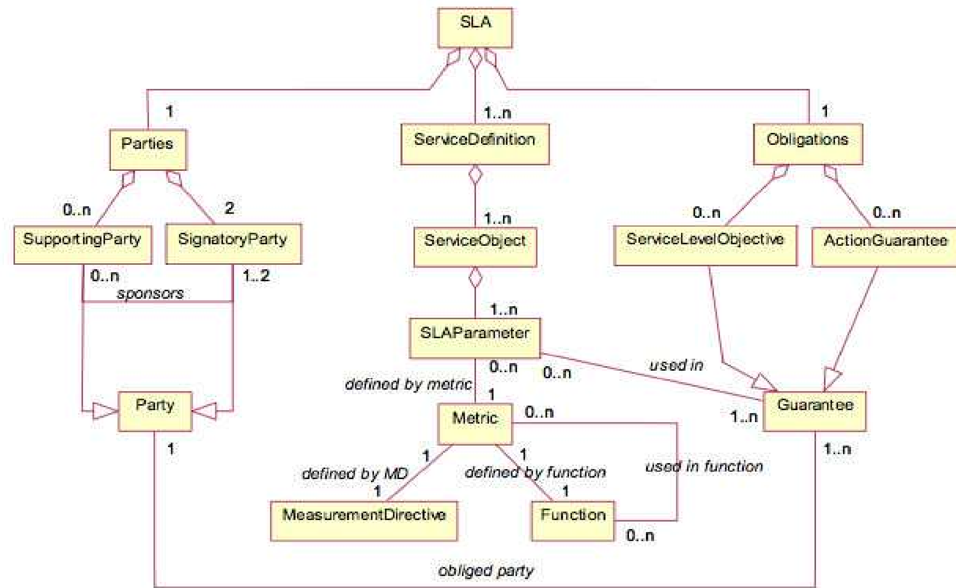


FIGURE 3.1 – Méta-modèle WSLA

L'exemple suivant illustre la section Obligations. Le temps de réponse doit être inférieur à 5 secondes. En cas de violation le client sera notifié.

```
<Obligations>
<ServiceLevelObjective name="g1" serviceObject="WSDLSOAPGetQuote">
<Obliged>provider</Obliged>
<Validity>
<StartDate>2001−08−15:1400</StartDate>
<EndDate>2001−09−15:1400</EndDate>
</Validity>
<Expression>
<Predicate xsi:type="wsla:Less">
<SLAParameter>AverageResponseTime</SLAParameter>
<Value>5</Value>
</Predicate>
</Expression>
<EvaluationEvent>NewValue</EvaluationEvent>
</ServiceLevelObjective>
<ActionGuarantee name="g2">
<Obliged>ms</Obliged>
<Expression>
<Predicate xsi:type="wsla:Violation">
<ServiceLevelObjective>g1</ServiceLevelObjective>
</Predicate>
</Expression>
<EvaluationEvent>NewValue</EvaluationEvent>
<QualifiedAction>
<Party>customer</Party>
<Action actionName="notification" xsi:type="Notification">
<NotificationType>Violation</NotificationType>
<CausingGuarantee>g1</CausingGuarantee>
<SLAParameter>AverageResponseTime</SLAParameter>
</Action>
</QualifiedAction>
<ExecutionModality>Always</ExecutionModality>
```

```
33 </ActionGuarantee>
```

ii) Web Services Agreement (WS-Agreement)

WS-Agreement [Aa07] est un protocole proposé par OGF (Open Grid Forum). Il définit un standard pour la création et la spécification de SLA pour les services Web. WS-Agreement utilise XML comme format de présentation, ce qui facilite la découverte des fournisseurs compatibles. Son interaction est basée sur la demande et la réponse. La Figure 3.2 illustre la structure de base d'un accord selon WS-Agreement. Un ac-

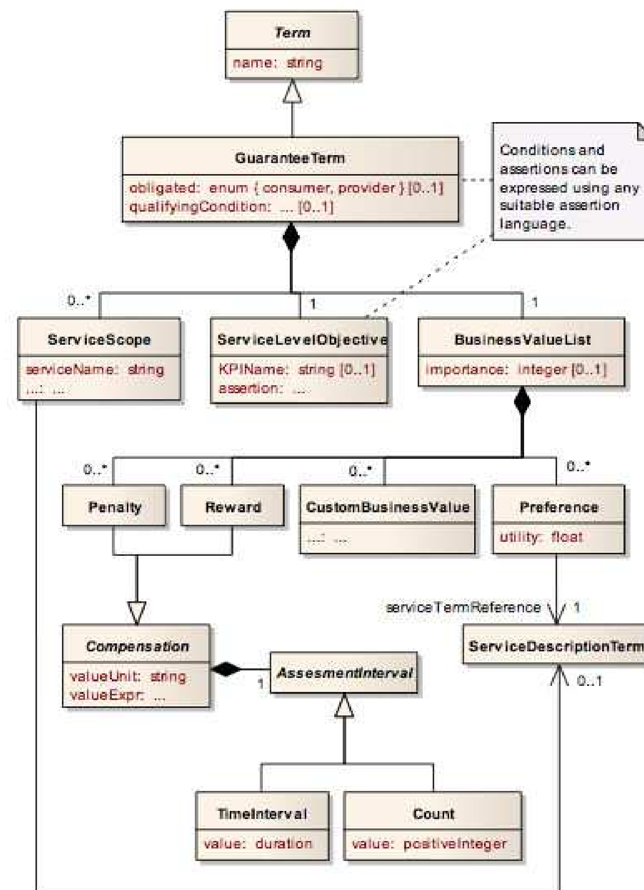


FIGURE 3.2 – Méta-modèle WS-Agreement

cord contient un identifiant d'accord, son nom, le contexte de l'accord et les termes du contrat. Le contexte contient un ensemble de métadonnées relatives à un accord à savoir les parties impliquées et la validité. Deux acteurs sont définis par WS-Agreement : l'initiateur et le répondeur. Cependant, les détails d'un acteur ne sont pas précisés afin de supporter tout type de description. Le contexte de l'accord peut également contenir une date d'expiration qui définit la validité de l'accord. Le contexte devrait également référencer le modèle qui a été utilisé pour créer le contrat. Les termes du contrat présentent les services (termes de service) et les garanties (termes de garanties). Un service est identifié par un nom. Il est décrit par un ou plusieurs termes de service. Les termes de service sont divisés en trois types : description de service, références de service, et propriétés du service. La description du service donne une description fonctionnelle du service à fournir. WS-Agreement est conçu pour être indépendant du domaine. Le contenu d'une description de service peut être n'importe quel document

XML valide compréhensible par les deux parties impliquées dans le contrat. Les références de service constituent un moyen pour se référer à des services existants au sein d'un accord. Alors que les propriétés du service fournissent un moyen de définir des variables dans le cadre du contrat. Les termes de garantie expriment les garanties de service et définissent la manière dont les garanties sont évaluées et le comportement en cas de violation. Les termes de garantie se composent de quatre éléments à savoir : la portée de service qui précise les services couverts par la garantie, la condition de qualification qui définit les conditions préalables pour garantir un objectif, un SLO qui définit un objectif à atteindre et la liste de Business Value qui détermine les pénalités et les récompenses qui sont associés à une garantie.

Depuis 2011, WS-Agreement est enrichie pour supporter la négociation. WS-Agreement Negotiation [ea11] couvre tout le cycle de vie SLA. Le résultat de sa combinaison avec WSLA pourrait gérer mieux la définition de SLA. Certains projets, par exemple comme BREIN [BRE13], ont essayé de fusionner WS-Agreement et WSLA pour leurs implémentations. Au niveau gestion de pénalité, WS-Agreement exprime les pénalités et les récompenses (Business Value) associées à une garantie via une unité (*valueUnit*) et une expression (*valueExpr*).

L'exemple suivant illustre les termes de garanties. Le temps de réponse doit être inférieur à une seconde. En cas de violation, une pénalité de 25 euros est déclenchée.

```xml
1  <?xml version="1.0" encoding="UTF−8"?>
2  <wsag:GuaranteeTerm
3  Name="FastReaction" Obligated="ServiceProvider">
4  <wsag:ServiceScope ServiceName="GPS0001">
5  http://www.gps.com/coordsservice/getcoords
6  </wsag:ServiceScope>
7  <wsag:QualifyingCondition>
8  applied when current time in week working hours
9  </wsag:QualifyingCondition>
10 <wsag:ServiceLevelObjective>
11 <wsag:KPITarget>
12 <wsag:KPIName>FastResponseTime</wsag:KPIName>
13 <wsag:Target>
14 //Variable/@Name="ResponseTime" LOWERTHAN 1 second
15 </wsag:Target>
16 </wsag:KPITarget>
17 </wsag:ServiceLevelObjective>
18 <wsag:BusinessValueList>
19 <wsag:Importance>3</wsag:Importance>?
20 <wsag:Penalty>
21 <wsag:AssesmentInterval>
22 <wsag:TimeInterval>1 month</wsag:TimeInterval>
23 </wsag:AssesmentInterval>
24 <wsag:ValueUnit>EUR</wsag:ValueUnit>
25 <wsag:ValueExpr>25</wsag:ValueExpr>
26 </wsag:Penalty>
27 <wsag:Preference>
28 <wsag:ServiceTermReference>
29 //ServiceProperties/@Name="UsabilityProperties"
30 </wsag:ServiceTermReference>
31 <wsag:Utility>0.75</wsag:Utility>
32 </wsag:Preference>
33 </wsag:BusinessValueList>
34 </wsag:GuaranteeTerm>
```

iii) Autres solutions

En plus de WSLA et WS-Agreement, nous avons identifié les solutions suivantes :

Service Level Agreement language (SLAng) SLAng [LSEO3] est un langage développé dans le cadre de projet TAPAS [TAP13]. Il est modélisé en utilisant UML (Unified Markup Language). Son objectif était de fournir une définition claire des obligations de tous les partenaires concernés par rapport à une certaine QdS. La dernière version a été mise en ligne en 2006 avec un avertissement explicite qu'il s'agit d'une version très immature qui ne devrait pas être utilisé pour de vrai SLA.

Rule-based Service Level Agreements (RBSLA) RBSLA [Pas05] se concentre sur les concepts de représentation des connaissances pour la gestion du SLA des services IT. RBSLA décrit le SLA formellement en utilisant RuleML. Il est possible via un moteur de règles de contrôler les clauses signées dans le SLA.

Web Service Offerings Language (WSOL) WSOL [TPE+02], une spécification basée sur XML, permet la spécification formelle des contraintes pour les services Web. Il permet de proposer les services Web avec différents niveaux de services via les différentes classes de service. WSOL est compatible avec WSDL et permet la spécification formelle des contraintes fonctionnelles, contraintes de QdS, droits d'accès simples, prix, responsabilités de gestion et les relations entre les offres de services. Les différentes classes de service sont appelées offre de service.

Nous pouvons citer aussi les solutions basées sur les langages sémantique comme OWL (Web Ontology Language) [HPSvH03] ou WSMO (Web Service Modeling Ontology) [Ont13].

3.1.2 SLA @ Cloud

En ce qui concerne les services Cloud, NIST [MGG11] a proposé une définition de SLA. Par contre, aucune tentative n'a été faite pour implémenter cette définition jusqu'à aujourd'hui. De même, Cloud Standards Customer Council [ea12] a présenté un guide pratique de SLA qui est un effort de collaboration qui rassemble diverses expériences axées sur le client. Ainsi, Rackspace [Rac13] a défini un ensemble de métriques via Cloud monitoring pour établir un contrat entre deux acteurs du Cloud. En plus des solutions industrielles, plusieurs projets académiques ont été développés pour répondre aux limites proposées par les services Cloud. En particulier le projet SLA@SOI[1] propose un nouveau langage le SLA* [KTK10] .

Dans ce qui suit, nous détaillons d'abord le langage SLA*. Ensuite, nous développons les solutions proposées dans des projets européens ou français.

i) SLA*

Le projet SLA@SOI s'intéresse à la prise en compte du SLA pour les infrastructures orientées service (Service Oriented Infrastructure SOI). Il met l'accent sur trois objectifs : la prévisibilité et la sûreté de fonctionnement, la gestion transparente de SLA et l'automatisation de processus de SLA. Le résultat de projet est un framework open-source. Ce dernier est évalué par quatre cas d'utilisation industriels distincts et complémentaires. La nouveauté du projet est l'aspect multi-couches de la pile de service. La solution prend en charge la configuration des hiérarchies de services complexes en permettant

[1] sla-at-soi.eu/

la gestion de bout-en-bout de ressources et de services.

SLA@SOI propose SLA* [KTK10] une syntaxe abstraite pour SLA. La solution est à la fois très expressive et extensible. Cette solution est inspirée principalement de WS-Agreement. SLA* favorise la formalisation des SLA dans n'importe quel langage pour n'importe quel service en éliminant la restriction de XML. Selon SLA@SOI (voir Figure 3.3), un SLA est un template SLA avec un ensemble étendu d'attributs précisant la validité de contrat. Un template SLA contient cinq sections : i) les attributs de template SLA, ii) les parties de l'accord, iii) les descriptions de services, iv) les déclarations de variables, et v) les termes de l'accord, en précisant les garanties de qualité de service. Les parties dans la syntaxe abstraite SLA(T) sont identifiées par leur rôle (fournisseur, client). La description de services est définie via les déclarations de l'interface. Une déclaration d'interface est utilisée pour attribuer un identificateur local à une interface qui peut être une interface fonctionnelle ou une description d'une ressource. Les déclarations de variables sont fournies pour améliorer la lisibilité et éviter la répétition de contenu. Les termes de l'accord sont formalisées comme des garanties de deux types : garantie d'action et garantie d'état.

En plus des expressions simples (par exemple, une constante, ou une proportionnalité linéaire), SLA* propose un modèle formel pour formaliser les pénalités. La motivation principale de cette proposition est l'ouverture et l'applicabilité à différents domaines sans dépendance à l'égard des langages spécifiques, des taxonomies ou des technologies.

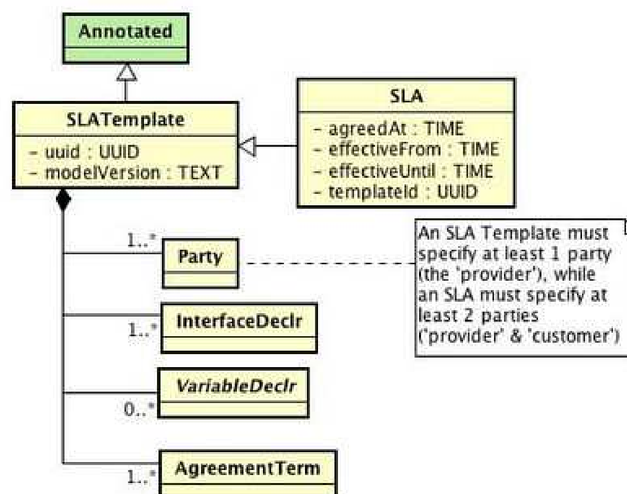


FIGURE 3.3 – Méta-modèle SLA@SOI

L'exemple suivant illustre la section "AgreementTerms". Le temps de traitement doit être inférieur à 30 minutes. En cas de violation une pénalité de 10 euros sera exécutée.

```
1 <agreementTerms>
2 <sla:AgreementTerm>
3 <name>AT1</name>
4 <guarantees>
5 <sla:State>
6 <name>G1</name>
7 <obligated>Yousri</obligated>
8 <pre>
9 <AtomicConstraint>
```

```
10 <item>X</item>
11 <AtomicDomain op="&core;equals">basic</AtomicDomain>
12 </AtomicConstraint>
13 </pre>
14 <post>
15 <AtomicConstraint>
16 <Parametric op="&qos;completion_time">S</Parametric>
17 <AtomicDomain op="&core;less_than">2 hrs</AtomicDomain>
18 </AtomicConstraint>
19 </post>
20 </sla:State>
21 <sla:State>
22 <name>G2</name>
23 <obligated>Yousri</obligated>
24 <pre>
25 <AtomicConstraint>
26 <item>X</item>
27 <AtomicDomain op="&core;equals">premium</AtomicDomain>
28 </AtomicConstraint>
29 </pre>
30 <post>
31 <AtomicConstraint>
32 <Parametric op="&qos;completion_time">S</Parametric>
33 <AtomicDomain op="&core;less_than">30 mins</AtomicDomain>
34 </AtomicConstraint>
35 </post>
36 </sla:State>
37 <sla:Action>
38 <name>G3</name>
39 <obligated>Yousri</obligated>
40 <policy>mandatory</policy>
41 <pre>
42 <Parametric op="&core;union">
43 <array>
44 <Parametric op="&sla;violation">G1</Parametric>
45 <Parametric op="&sla;violation">G2</Parametric>
46 </array>
47 </Parametric>
48 </pre>
49 <limit>2 weeks</limit>
50 <post>
51 <sla:Payment>
52 <recipient>TheCustomer</recipient>
53 <value>10 euros</value>
54 </sla:Payment>
55 </post>
56 </sla:Action>
57 </guarantees>
58 </sla:AgreementTerm>
59 </agreementTerms>
```

ii) Autres solutions

De nombreu projets portent sur le Cloud. Nous reportons ici seulement les projets qui supportent la définition de SLA.

Contrail [Con13] : Le projet FP7 Contrail a pour objectif de développer en open-source une pile logicielle complète pour le Cloud Computing. Ce projet traite en particulier la fédération du Cloud. En se basant sur le résultat de SLA@SOI, Contrail définit un SLA via SLA* avec quelques amélorations pour maîtriser la fédération du Cloud.

OPTIMIS [ZJ11] : Le projet OPTIMIS fournit un toolkit qui facilite l'utilisation des ressources en fonction des objectifs économiques et l'éco-efficacité tout en réalisant une gestion dynamique et proactive des infrastructures Cloud. OPTIMIS définit le SLA via WSAG4J[2] : une implémentation complète de WS-Agreement et WS-Agreement Nego-

[2]http ://wsag4j.sourceforge.net/site/server/architecture.html

ciation, y compris la définition et la manipulation des garanties et des pénalités.

mOSic [mOS13] Le Projet FP7 mOSic vise à fournir une API et une plateforme pour simplifier le développement d'applications orientées multi-Cloud. mOSic ne gère de SLA qu'au niveau IaaS. Une première tentative de définition de SLA était basée sur WS-Agreement couplé partiellement avec WS-Policy [WP13b]. Cette solution est remplacée par SLA* de SLA@SOI.

Cloud4SOA [Clo13b] Le projet européen Cloud4SOA permet la surveillance des applications hébergées sur de multiples environnements Clouds de façon complète. Cloud4SOA fournit une implémentation RESTful de la spécification WS-Agreement et adapte quelques résultats de SLA@SOI.

MyCloud [MyC13] L'objectif du projet ANR MyCloud est de définir et réaliser le modèle de Cloud SLAaaS (SLA aware Service). Ce modèle propose un service orthogonal aux autres modèles de Cloud (IaaS, PaaS et SaaS) et intègre de manière native les concepts de QdS et de SLA au Cloud. Cette thèse s'inscrit dans le cadre de ce projet. Notre contribution CSLA est utilisée comme langage SLA.

OpenCloudware [Ope13] Le projet FSN OpenCloudware vise à fournir une plateforme d'ingénierie logicielle ouverte permettant des développements collaboratifs d'applications Cloud, ainsi que leur déploiement et administration afin de garantir le SLA, en visant une portabilité sur des infrastructures Cloud IaaS multiples. Comme nous sommes membre de ce projet, CSLA est un candidat comme langage de définition de SLA. Une extension est possible pour répondre aux contraintes du projet.

CompatibleOne [Com13] CompatibleOne propose un système de broker open source. Il permet d'agréger, de provisionner et d'administrer différents services Cloud afin de répondre aux exigences des entreprises, qui peuvent solliciter plusieurs services de plusieurs fournisseurs. Coté SLA, la solution de CompatibleOne est basée sur une simple implémentation de WS-Agreement.

3.1.3 SLA niveau IaaS

Dans cette section, nous étudions les SLAs proposés par les fournisseurs de Cloud. Nous nous intéressons, en particulier, aux offres IaaS. En effet, c'est actuellement le seul niveau XaaS qui propose de réel SLA sur le marché. Le contrat de niveau de service est assez uniforme entre les différents fournisseurs IaaS. Le chiffre à retenir est : 99,95% de disponibilité. Cependant, le mode de calcul de l'indisponibilité de service est assez différent d'un fournisseur à l'autre.

Dans ce qui suit, nous exposons, d'abord, la disponibilité de point de vue Amazon EC2 et Microsoft Azure. Ensuite, nous présentons le crédit de service ou la pénalité en cas de violation.

i) 99,95% de disponibilité

Amazon EC2 offre un service de calcul disponible avec un pourcentage de temps utilisable annuel d'au moins 99,95% pendant l'année de service [EC213b]. Le "pourcentage

de temps utilisable annuel" est calculé en soustrayant de 100% le pourcentage de périodes de 5 minutes pendant l'année de service durant laquelle Amazon EC2 était en statut de "Région Indisponible". Microsoft Azure offre un SLA mensuel [Azu13a]. Le pourcentage de connectivité active mensuelle est donné par la formule suivante :

$$\frac{\textit{Maximum Connectivity Minutes} - \textit{Connectivity Downtime}}{\textit{Maximum Connectivity Minutes}} \tag{3.1}$$

Le nombre de minutes de connectivité maximale (*Maximum Connectivity Minutes*) représente la durée totale cumulée en minutes au cours d'un mois de facturation dont au moins deux instances sont déployées dans différents domaines de mise à jour. Les interruptions de connectivité (*Connectivity Downtime*) représentent la durée totale cumulée en minutes pendant laquelle le client ne bénéficie d'aucune connectivité externe pendant une durée de cinq minutes, mesurée et cumulée par intervalles de cinq minutes. Comme Microsoft, Google App Engine [Eng13b] offre un SLA mensuel, et est basé sur la même formule. Cependant, l'indisponibilité signifie plus d'un taux de dix pour cent d'erreur pour toute demande admissible.

TABLE 3.1 – Bilan de SLA niveau IaaS

	Disponibilité	Méth. de calcul	Pénalités
Amazon EC2	pourcentage de temps utilisable Annuel >99,95%	Année, périodes de 5 minutes	Crédit de Service : 10% de la facture si Pourcentage de Temps Utilisable Annuel est inférieur à 99,95%.
Microsoft Azure	pourcentage de connectivité active mensuelle >99,95 %	Mois, périodes de 5 minutes	Avoirs service : 10% de la facture si pourcentage de connectivité active mensuelle est inférieur à 99.95%, 25% si c'est moins de 99,00%
Google App Engine	pourcentage de connectivité active mensuelle >99,95%	Mois, périodes de 5 minutes	Crédit de service : 10% de la facture si pourcentage de connectivité active mensuelle est entre 99.00% et 99.95%, 20% si c'est entre 95.00% et 99.00%, sinon 50% si c'est inférieur à 95.00%

ii) Crédit de service

Au cas où le fournisseur ne respecte pas ses engagements de taux de disponibilité, le client pourra recevoir un crédit de service. L'utilisateur doit prouver que son service a été indisponible selon le SLA signé et envoyer la "preuve" au fournisseur, le tout dans un délai de "j" jours ouvrable (j=5 jours pour Microsoft, et 30 jours pour Amazon et Google) après l'occurrence de l'incident. Les fournisseurs de Cloud proposent tous le même type de pénalité (voir Tableau 3.1) : un avoir sur l'utilisation de leur service, plafonné sur le montant d'achat. Autrement dit, le fournisseur n'est engagé que sur la somme que le client lui a versée.

De plus, le SLA détaille explicitement l'exclusion de périmètre de responsabilité du fournisseur. L'engagement de service Amazon ne s'applique pas à toute indisponibilité,

suspension ou résiliation de Amazon EC2, ou tout autre problème de performance de Amazon EC2. Des termes très similaires existent dans le contrat Google App Engine.

3.1.4 Discussion

Suite à l'étude des travaux précédents dans la littérature, nous avons identifié principalement : WSLA [LF03], WS-Agreement [Aa07], et WS-Agreement Negotiation [ea11]. WS-Agreement est le langage le plus flexible au niveau définition des termes de contrat. Cela signifie que les utilisateurs sont libres de définir leurs modalités comme ils veulent, mais en revanche un problème d'automatisation est posé vu qu'il n'y a pas une définition commune. Ces travaux ont contribué de manière significative à la standardisation de SLA. Cependant, aucun ne propose une solution complète pour le Cloud. En effet, deux limites majeures sont identifiées : la définition de service et la gestion avancée des pénalités. XaaS (X as Service) est à la base du paradigme du Cloud. Même si nous considérons le SaaS comme un service Web pouvant être traiter avec WSLA ou WS-Agreement, ce n'est pas le cas pour le PaaS et le IaaS. De plus, ces travaux n'abordent pas la gestion de pénalité de façon spécifique ce qui peut s'avérer problématique dans un environnement hautement dynamique comme le Cloud.

Dans la communauté Cloud, la plupart des solutions (voir Tableau 3.2 où WSA signifie WS-Agreement) sont basées sur une implementation ou une extension de WS-Agreement. Optimis [ZJ11], par exemple, propose une solution basée sur WSAG4J une implémentation Java de WS-Agreement. Nous avons pu observer que le format XML est le plus utilisé pour faciliter l'intégration et l'interopérabilité des services. Alors que SLA@SOI [WY11] a constaté les limitations des normes spécifiques aux web services et a proposé SLA* pour décrire les clauses entre deux parties. Le modèle a été développé comme une amélioration de WS-Agreement afin d'éliminer la restriction de XML en tant que format de représentation et une généralisation pour dépasser la notion de "service web". Il favorise la formalisation des SLA dans n'importe quel langage pour n'importe quel service. Contrail [Con13] utilise cette solution pour la définition de SLA avec une extension pour la fédération.

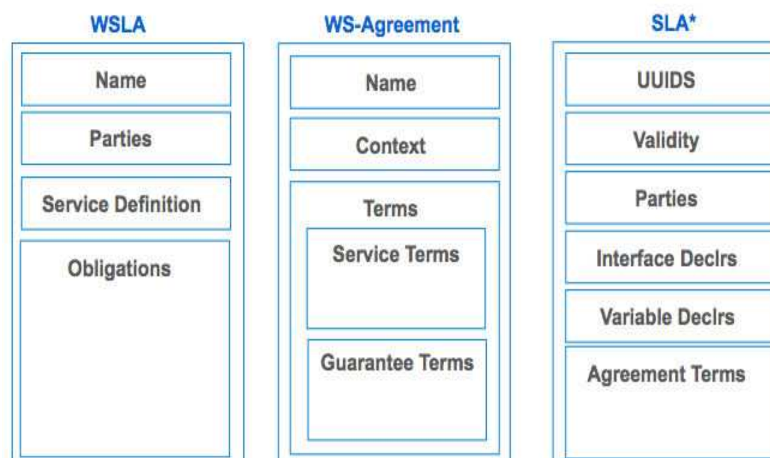


FIGURE 3.4 – WSLA, WS-agreement et SLA*

La Figure 3.4 illustre la structure de SLA selon WSLA, WS-Agreement et SLA*. Un

patron commun est remarquable : les parties, les services et les garanties. Vue que SLA* est inspiré de WS-Agreement, la similarité de structure est forte bien que le vocabulaire utilisé soit différent. Nous retrouvons, par exemple, le contexte dans la validité et les parties. Ainsi les déclarations des interfaces sont équivalentes aux termes de service. Cependant WSLA reste un peu différent en particulier au niveau détails de la section obligations.

Nous constatons que les solutions dans la communauté de Cloud ne sont pas bien avancées. En effet, très peu de travaux ont présenté clairement le niveau de Cloud abordé dans leurs solutions. La section de définition de service reste généralement restreinte à un seul niveau XaaS à savoir le SaaS. Aucune solution ne supporte une définition des ressources IaaS, par exemple, dans leur SLA. SLA@SOI recourt à un fichier OVF (Open Virtualization Format)[3] pour décrire les ressources IaaS alors que le SLA reste juste pour la définition des garanties. Malgré les efforts de standardisation de modèle de ressources comme OCCI, plusieurs modèles co-existent sur le marché : Amazon EC2, Oracle... L'absence de norme pour le Cloud est très remarquable. En effet, les définitions de IBM ou Oracle par exemple ne suivent pas la référence NIST, et sont directement liées à leurs technologies.

Au sujet de cycle de vie de SLA, la plupart des approches ne définissent pas la phase traitée dans le cycle de vie SLA. La solution SLA@SOI est la seule qui couvre tout le cycle de vie SLA. Les autres solutions ne traitent qu'une partie de cycle de vie de SLA.

Par définition, l'élasticité du Cloud offre un environnement hautement dynamique dans lequel les ressources et le réseau fluctuent énormément. Dans un tel environnement, l'instabilité de la QdS et l'incertitude de QdS augmentent. C'est pourquoi, les violations SLA sont possibles. De plus, les variations de charge cliente importantes sont parfois difficile à maîtriser pour le fournisseur et peuvent engendrer, également, des violations SLA. Nous avons remarqué que les langages existants ne tiennent pas compte des conséquences de l'élasticité. En effet, les violations sont gérées à gros grain. Nous constatons que la pauvreté d'expression est la limite principale des langages existants.

Concrètement sur le marché du Cloud, les fournisseurs de IaaS sont les seuls parmi les fournisseurs XaaS qui s'engagent vis-à-vis d'un SLA. Néanmoins, nous avons observé plusieurs limites. Le contrat de niveau de service de différents acteurs IaaS est basé sur une seule métrique : la disponibilité. D'autre part, la vérification de la non violation de SLA peut être très fastidieuse pour le client de Cloud. Amazon calcule son taux de disponibilité à l'année et non sur un mois. Avec un taux garanti de 99,95%, Amazon peut lisser plus facilement les indisponibilités sur l'année jusqu'à un peu plus de 4 heures. Si la disponibilité était calculée sur un mois, un client du service serait en droit de demander des dédommagements au-delà de 21 minutes d'interruption. Un délai très facilement atteint en cas de panne.

Les acteurs étudiés présentent le SLA soit par une page Web (cas de Amazon, Google App Engine) ou un document word (Microsoft Azure). Nous remarquons que les SLAs suivent (presque) la même structure : définitions (terme utilisés pour calculer la disponibilité), garanties, crédits de service (requête de crédit, procédures de paiement) et

[3]http ://dmtf.org/about/faq/ovf_ faq

Table 3.2 – Solutions SLA

	projet	niveau	langage	inspiré de	nouveauté
Europe	SLA@SOI	IaaS	SLA*	WSA	SLA multi-couches
	Contrail	IaaS	SLA* extended	SLA*	fédération
	Optimis	IaaS	WSAG4J	WSA	économie
	mOSAIC	IaaS	SLA*	WSA	API
	Cloud4SOA	IaaS	WSA + WS-Plicy, SLA*	WSA	monitoring
France	CompatibleOne	broker	WSA	WSA	CORDS
	MyCloud	XaaS	CSLA	WSA et SLA*	modèle SLAaaS
	OpenCloudware	PaaS	CSLA	WSA et SLA*	Calibrage de SLA

exclusions du SLA. Bien que la fonction de calcul de disponibilité soit bien déterminée, les crédits (pénalités) restent flous et transparents pour l'utilisateur de service IaaS.

3.2 Accord de niveau de service (SLA), les dépendances

Les SLAs peuvent être exprimés entre différentes couches dans la "pile" du Cloud, contractualisant de fait des dépendances entre les différentes couches XaaS. A l'exception du client final (end-user), chaque acteur dans la hiérarchie XaaS est un client-fournisseur : un client des acteurs de niveaux inférieurs et un fournisseur des acteurs de niveaux supérieurs. Afin de proposer un SLA à un niveau n, il faut que les niveaux inférieurs invoqués proposent déja des SLAs. Une violation d'un SLA de niveau n peut produire des violations dans les couches supérieures [HBS10a] [HBS10b]. Dans notre contexte, le SaaS est le client d'un service IaaS et le fournisseur de service SaaS pour le client final. Le SaaS doit calibrer et négocier le contrat à signer avec son client final (SLA_S) en se basant sur le contrat signé avec son fournisseur (SLA_R).

Il est difficile, voire impossible pour un opérateur humain de résoudre les dépendances verticales. Etant faillible par ailleurs, même un expert du domaine risque lui-même d'introduire des erreurs dans le SLA de son client, le rendant peut être irréalisable. Une solution automatique devient nécessaire. L'implantation d'un tel calibrage d'une manière automatique fait apparaître un certain nombre de défis à savoir principalement la translation des QdS de niveau $n-1$ au QdS de niveau n.

Dans ce qui suit, nous détaillons, d'abord, le problème de dépendance de point de vue théorie ensuite nous exposons l'état des lieux. Enfin, nous présentons un bilan des travaux.

3.2.1 Les dépendances, les concepts

La dépendance peut être définie comme suit : un critère A dépend d'un autre critère B si un changement de B provoque un changement de A [HWH09]. Nous étudions en particulier la dépendance verticale entre les SLAs dans une "pile" du Cloud. Dans

notre scénario, nous considérons le fournisseur IaaS comme le fournisseur final (end-provider). Le SLA proposé par ce fournisseur est trés restreint en matière de garantie. Il est basé uniquement sur la disponibilité. Avec un tel SLA, le problème de dépendance (niveau SaaS) devient plus compliqué à résoudre.

Dans ce qui suit, nous développons l'intra-dépendance et l'inter-dépendance. Ensuite, nous étudions la translation de SLA et la translation de QdS.

i) Intra-dépendance et Inter-dépendance

Les SLAs peuvent être exprimés à tous les niveaux. Lorsque les SLAs appartiennent à une même couche n, il s'agit d'*intra-dépendance* (dépendance horizontale) de niveau n alors que pour les SLAs des niveaux différents, il s'agit d'*inter-dépendance* (dépendance verticale). Des initiatives [KMK09] [KS07] [DMRTV07] proposent des solutions pour maîtiser la dépendance horizontale. P. Karanke et al. [KMK09], par exemple, ont proposé une architecture qui facilite la négociation du SLA dans des hiérarchies d'accord complexe. La solution est basée sur un modèle de SLA –sous la forme d'un graphe– et un système multi-agents. D'autres auteurs [RBHJ08] ont proposé d'utiliser des SLAs probabilistes souples en cas d'une orchestration de services Web pour faciliter la composition. Cependant, les solutions proposées ne résolvent que partiellement la dépendance verticale. En effet ces solutions ne couvrent que l'aspect de composition de SLA (QdS) alors que la dépendance verticale fait apparaître un autre défi à savoir la translation de SLA.

"SLA translation as any form of transformation of metrics and parameters, within one layer or from one (sub)layer to another ..." Theilmann et al. 2012 [WY11]

L'inter-dépendance est une spécificité du Cloud. Résoudre l'inter- dépendances SLA revient à exprimer un SLA entre un acteur de niveau i et ses clients (couches supèrieures $\forall j > i$) via les paramètres de SLA entre l'acteur et ses fournisseurs (couches inférieures $\forall k < i$). Autrement dit, c'est la *translation* ou la *traduction* de SLA_R en SLA_S pour notre contexte. Traduire la disponibilité de service SaaS en fonction de la disponibilité des ressources IaaS est un des exemples que nous cherchons à résoudre.

ii) Translation de SLA

Telles qu'illustrées par la Figure 3.5, nous distinguons plusieurs translations avec des complexités différentes. Le cas (A) est le plus simple. Il s'agit d'un seul fournisseur IaaS, donc un seul SLA_R. Alors que le cas (B) est basé sur deux ou plusieurs SLA niveau IaaS. La complexité du problème peut encore augmenter dans le cas ou il y a un mixage entre intra-dépendances et inter-dépendances (voir cas (C)).

iii) Translation de QdS

Restons avec l'exemple le plus simple (le cas (A)), plusieurs catégories d'inter-dépendances QdS sont possibles [EBMD10] comme illustrés par la Figure 3.6 : i) *un*−à−*un* (A), exemple : fournir la taille de stockage à partir de disque dur, ii) *plusieurs*−à−*un* (B), exemple : le temps de réponse peut être calculer à partir des indicateurs de CPU et RAM. et iii) *un*−à−*plusieurs* (C), exemple : le CPU peut être utilisé pour définir à la fois le temps de réponse d'une requête et aussi la disponibilité du service. Également,

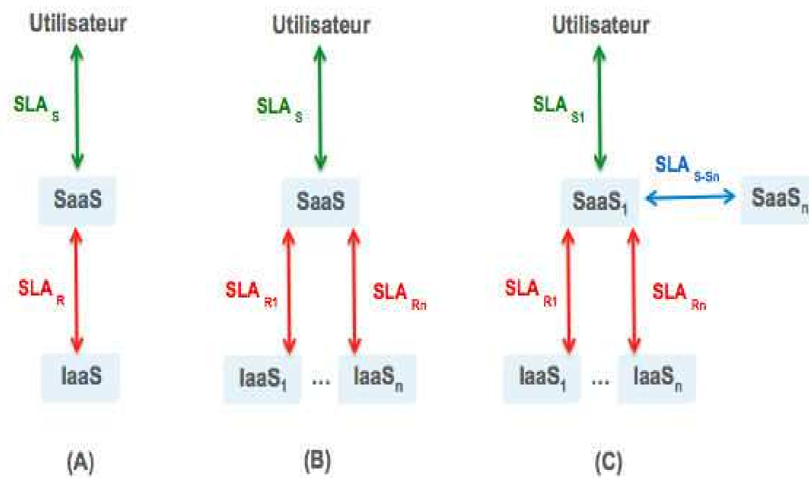


FIGURE 3.5 – inter-dépendances de SLA

nous trouvons les mêmes catégories pour des intra-dépendances. Par exemple pour le *un*−à−*un* : définir la disponibilité de service à partir du temps de réponse.

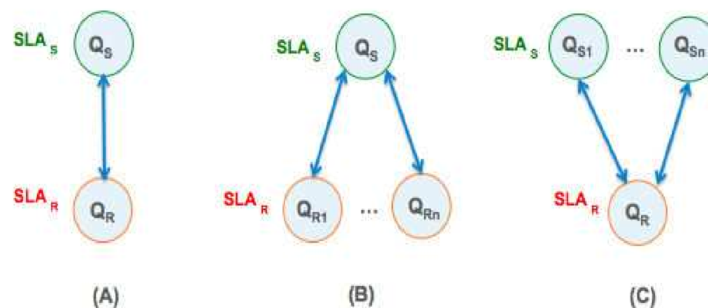


FIGURE 3.6 – inter-dépendances de QdS

3.2.2 Les dépendances, la pratique

Dans cette section, nous présentons l'état des lieux de la dépendance verticale. En premier lieu, nous précisons dans quel contexte ce type de dépendance est étudié. Nous introduisons, ensuite, comment ce problème est modélisé. Enfin, nous exposons les initiatives plus particulièrement dans le but de calibrage/négociation, prédiction et planification.

i) Quel contexte ?

Très peu de travaux [WY11] [KYG10] abordent la gestion de dépendances SLA dans le Cloud telle qu'annoncée précédemment : la définition, le calibrage et la négociation de SLA (début de cycle de vie de SLA). Cependant, la translation de SLA et QdS est traitée dans d'autres contextes. Nous citons principalement i) le monitoring (SERENITY [SKM09], MoDe4SLA [BWRJ08], LoM2HiS [EBMD10], CASViD [EFN+12]) ii) la planification (ou encore la translation top-down) [CIL+08][CSI+08] et iii) la prédiction de performance (ou la translation bottom-up) [UPS+07][JJH+08][WCSO08].

ii) Comment modéliser ?

Plusieurs approches sont utilisées pour modéliser le problème de dépendances à savoir : les approches basées sur des règles, des approches basées sur des modèles analytiques et des approches basées sur l'apprentissage.

Les connaissances et les règles d'un expert du domaine spécifique sont appliquées pour faire des déductions ou des choix. Ces approches sont sollicitées dans le domaine de réseau où les paramètres de QdS réseau apparaissent naturellement dans des couches multiples et différents niveaux d'abstraction [HKSW06]. Dans certains domaines, les règles prédéfinies ne sont pas toujours facile à gérer, ou tout simplement pas possible. Ainsi les bases de connaissances sont elles-mêmes difficiles à construire, à entretenir et à adapter aux changements du système. Les approches basées sur la modélisation analytique, en se référant particulièrement aux modèles de réseaux des files d'attente, présentent une solution alternative [CIL+08]. Ces modèles de performance analytiques sont des outils mathématiques puissants, mais leur utilisation pratique peut être limités par les hypothèses simplificatrices. Une autre approche consiste à appliquer des statistiques et des techniques d'apprentissage automatique [KSI+08][ZWL08]. A partir des traces, cette approche induit les règles de translation. Un effort est nécessaire pour identifier la méthode d'apprentissage adéquate.

iii) Initiatives

Calibrage, négociation : Peu de travaux ont traité la gestion de dépendances verticales SLA. Nous avons identifié en particulier le projet SLA@SOI. Ce projet prend en charge la gestion complète de pile de service. Il résout le problème de dépendance via une approche basée sur des règles [WY11]. Cette approche facilite la gestion de dépendances pour un expert du domaine en s'appuyant sur des graphes de dépendances. Cependant, la solution SLA@SOI ne gère pas les dépendances cycliques en déléguant cette tache à l'expert du domaine. Ainsi les règles de dépendances ne sont déclenchées que dans des plages définies par l'expert. En complément, dans [KYG10], il y a une présentation générale d'une architecture qui traite la renégociation ainsi que l'inter-dépendance SLA et la hiérarchie des SLAs mais sans indiquer les détails. Selon le meilleur de nos connaissances, il n'y a pas une solution qui répond à cette problématique d'une manière automatique à 100%. Même si certains critères de QdS peuvent avoir le même indicateur – comme la disponibilité – leur sémantique dépendra fortement de la couche et du domaine d'application.

Prédiction : Généralement, une translation ascendante (bottom-up) réfère à la prévention/prédiction des performances [UPS+07] [JJH+08]. Compte tenu des paramètres des ressources, des critères de QdS de haut niveau sont prédits via un modèle de performance. Lors de la conception, la théorie des files d'attente est la plus utilisée pour modéliser la translation. Alors que la théorie de contrôle, les séries chronologiques sont les méthodes les plus efficaces lors de l'exécution.

Urgaonkar et al. [UPS+07] ont présenté un modèle analytique pour les applications Internet multi-tiers basé sur l'utilisation d'un réseau des files d'attente. Ce modèle permet la prédiction de temps de réponse en fonction des ressources utilisées. De même, Jung et al. [JJH+08] ont proposé une approche basée sur un modèle des files d'attente avec des techniques d'optimisation pour prédire le comportement du système et gé-

nérer automatiquement des configurations optimales. En plus de la configuration en ligne, cette approche offre un traitement pour alimenter un arbre de décision. Ce dernier produit un ensemble de règles qui peut être utilisé dans un moteur de règles directement combiné avec d'autres règles prédéfinies ou simplement utilisées pour aider un expert à gérer les politiques de gestion.

La translation bottom-up peut simplifier la composition/sélection des services. Wada et al. [WCSO08] ont présenté $E3-MOGA$: un modèle de composition de service Cloud via un algorithme génétique multi-objectifs. L'algorithme permet de fournir une instance de processus métier (ensemble des services/ressources) qui satisfait un SLA. Identifier l'instance de processus métier revient à traduire chaque ensemble des services/ressources par le débit, le temps de réponse et le coût et comparer tous les cas possible.

Planification : Une translation descendante (top-down) réfère à la dérivation des seuils de haut niveau sur les ressources. Également, elle peut être modélisée par les files d'attente. Une telle translation se formule comme un problème de satisfaction de contraintes et les méthodes comme la programmation linéaire ou encore programmation par contraintes sont utilisées pour résoudre le problème. La translation top-down est utile lors de la planification.

[CIL+08][CSI+08] ont proposé un processus de traduction de temps de réponse moyen en ressources dans un environnement de e-commerce. En particulier, le processus fournit les spécifications des serveurs en terme de CPU et mémoire. Les auteurs ont modélisé le problème comme un problème de satisfaction de contraintes combiné avec un modèle des files d'attente. Les limitations de cette approche sont les suivantes : la charge traitée est stationnaire, les ressources utilisées sont homogènes et le temps de réponse utilisé est une moyenne.

3.2.3 Discussion

Nous avons présenté dans cette partie l'état de l'art sur les dépendances SLA dans le Cloud. Avec un SLA très restreint en matière de garanties au niveau IaaS, le problème d'inter-dépendances devient difficile à maîtriser. Tant et si bien que les approches de compositions de QdS, réglant efficacement l'intra-dépendance, ne résolvent pas la translation des QdS. Suite à l'étude des travaux courants dans la littérature, nous avons remarqué l'absence d'une solution automatique pour aider un acteur au niveau n pour définir, calibrer et négocier les SLA avec ses clients en fonction des SLA avec ses fournisseurs. Le calibrage de SLA est opéré jusqu'à présent exclusivement par des operateurs humains (experts du domaine).

Des initiatives telles que SLA@SOI [WY11] proposent des solutions semi-automatique où l'intervention humaine est non négligeable. Sinon, des alternatives de translation sont proposées dans d'autres contextes. En particulier, nous avons étudié deux types à savoir la planification via translation top-down et la prédiction par la translation bottom-up. Cette dernière peut résoudre en plus la sélection des services Cloud.

En plus de l'absence d'une solution automatique, nous avons relevé d'autres limites. Les solutions proposées ne tiennent pas compte de fédération (ne supporte qu'un seul

SLA au niveau fournisseur IaaS). Ainsi, le mixage entre intra-dépendances et inter-dépendances n'est pas abordé.

3.3 Gestion de Capacité des ressources

La gestion de la capacité est un processus en charge de veiller à ce que la capacité du système puisse répondre aux objectifs de niveau de service tout en optimisant l'utilisation des ressources. Elle prend en compte toutes les ressources nécessaires pour fournir le service informatique et planifie les besoins du "business" à court, moyen et long terme [4]. Une gestion efficace de la capacité de service peut avoir des effets directs sur la garantie de service. C'est l'assurance qu'un service répondra à la demande avec un niveau spécifié de QdS. La planification de capacité est une activité au sein de la gestion de la capacité ayant en charge la création d'un plan de capacité.

Parmi les approches qui contrôlent les niveaux d'un service et gèrent la capacité, nous distinguons les approches suivantes (voir Figure 3.7) [ST12] : le contrôle d'admission, l'ordonnancement, la répartition de charge, la dégradation de service et le dimensionnement des ressources. Chaque approche peut participer à l'implémentation de l'élasticité du Cloud.

Dans ce qui suit, nous étudions en détails le dimensionnement automatique des ressources. En effet, ce type de gestion de capacité rentre en résonance avec l'élasticité du Cloud. Puis, nous décrivons brièvement le reste des approches.

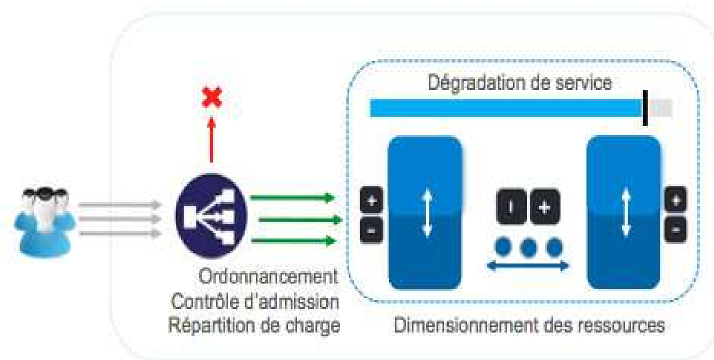


FIGURE 3.7 – Gestion du niveau de services

3.3.1 Dimensionnement automatique, les concepts

Le dimensionnement consiste à déterminer la quantité de ressources d'un système. En agissant sur la quantité de ressources du système, on contrôle sa capacité de traitement. Cela a un impact direct sur les performances des traitements effectués, et donc sur la QdS fournie. Le dimensionnement automatique est une implémentation de l'élasticité de l'informatique en nuage. Il peut être présenté par le quadruplet : (Quand, Comment, Combien, Où). Le "Quand" indique le moment de la mise en œuvre : en amont (proactif), en aval (réactif) ou hybride (réactif-proactif). Le "Comment" présente le type de

[4] http ://www.itlibrary.org/index.php ?page=CapacityManagement

dimensionnement : horizontal, vertical ou les deux. Le "Combien" indique la quantité de ressources concernées (planification de capacité). Le "Où" spécifie la cible à savoir le calcul, le stockage ou la répartion de charge.

Dans ce qui suit, nous présentons le quadruplet (Q,C,C,O) avec plus de détails. En particulier, nous exposons les différentes approches, modèles et algorithmes utilisés.

i) Quand

L'élasticité peut être mise en ouvre de trois façons différentes : dimensionnement réactif, dimensionnement proactif et dimensionnement hybride (réactif-proactif) [AMVBVL+13]. Le dimensionnement réactif est un dimensionnement automatique basé sur la demande. En utilisant un service de monitoring, le système peut émettre des éléments déclencheurs pour prendre les mesures nécessaires pour augmenter ou réduire la quantité de ressources en fonction de métriques. Alors que le dimensionnement proactif peut être i) un dimensionnement cyclique proactif : un dimensionnement périodique se produisant à des intervalles fixes (quotidiennement, hebdomadairement, mensuellement, trimestriellement), ii) un dimensionnement basé sur un évènement : un dimensionnement se produisant lorsqu'on prévoit une grande augmentation de demandes de trafic en raison d'un évènement programmé par l'entreprise (lancement d'un nouveau produit, campagnes de marketing) ou iii) un dimensionnement basé sur une prédiction de la charge dans le futur. Le dimensionnement hybride est une combinaison d'un dimensionnement réactif et un dimensionnement proactif.

La prédiction est le processus de faire des déclarations au sujet d'événements dont les résultats réels n'ont pas encore été observés. Les modèles de prédictions quantitatives, comme les séries chronologiques [BJ94] sont utilisées principalement pour prédire la charge du travail ou l'utilisation des ressources. Plusieurs méthodes sont proposées dans la littérature pour la prévision des valeurs. Nous distinguons les méthodes basées sur la moyenne, l'apprentissage automatique et le filtrage par motif. Ces méthodes sont souvent combinées pour obtenir diverses variantes de prédiction.

Moyenne : la prédiction est basée sur la moyenne pondérée des dernières valeurs consécutives. En fonction du calcul des poids, plusieurs méthodes sont définies : moyenne mobile (Moving average) [GGW10], moyenne mobile pondéré (Weighted moving average), filtre de Kalman (Kalman filtering), lissage exponentiel (Exponential smoothing), autorégressif (Auto-regression) [KGM10] ou encore autorégressif moyenne mobile (Auto-regressive Moving Average) [RDG11].

Apprentissage automatique : la prédiction est basée sur un processus d'apprentissage. Comme méthode il y a : la régression (Regression) et les réseaux de neurones (Neural networks) [IKLL12].

Filtrage par motif : La prédiction des valeurs est basée sur l'identification des motifs. Parmi les méthodes nous citons : les méthodes de traitement du signal comme la transformée de Fourier rapide (FFT) [GGW10] et les histogrammes [CGS03].

Le choix de la méthode de prédiction dépend du type de variation de charge. Parmi les types de variation, nous identifions les suivants : la variation aléatoire et la variation

horaire. La variation aléatoire est une variation imprédictible où l'accès des clients à un service suit un caractère aléatoire. Le comportement des clients peut, aussi, suivre une tendance prédictible pour produire une variation horaire. [wp13a] a proposé des patrons de charge dans le Cloud. Nous citons en particulier : *On and Off*, pic prédictible et imprédictible.

ii) Comment

Le dimensionnement peut être soit à l'echelle horizontale, verticale ou les deux [MMSW07] [DGVV12] [WCC12]. Un dimensionnement horizontal est la possibilité d'ajuster (augmenter (*scale out*) ou diminuer (*scale in*)) le nombre d'instances (VMs) du système à la demande. La virtualisation permet une autre façon de dimensionner : le dimensionnement vertical. L'hyperviseur [RG05] prend en charge le redimensionnement à chaud des VM en permettant l'ajout (*scale up*) ou la suppression (*scale down*) des ressources telles que CPU ou mémoire. Ce dimensionnement est limité par la quantité de ressource libre disponible sur le serveur physique hébergeant la machine virtuelle. La migration élargit le champ d'application de dimensionnement vertical. Elle consiste à déplacer (à froid ou à chaud) une VM d'un hyperviseur à un autre afin de réorganiser la consommation de ressources. 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.

Le dimensionnement horizontal existe dans toutes les solutions commerciales. Son coût varie selon le type de l'instance, les logiciels installés et le type de l'application (stateless, stateful). Alors que le dimensionnement vertical a aussi ses coûts connexes comme la migration à chaud. Il est largement utilisé pour la consolidation des centres de données dynamiques [VAN08b] [VAN08a].

iii) Combien

La quantité de ressources à ajouter ou supprimer peut être définie par deux manières : a) Planification meilleur effort (Best-effort) et b) Planification avec garantie.

a) Planification meilleur effort

C'est une solution simple à mettre en œuvre mais qui ne garantit pas les niveaux de service. Elle donne de bons résultats dans certaines conditions. Une expertise est nécessaire pour utiliser cette solution. Nous identifions, par exemple, les règles à base des seuils.

Règles à base des seuils : Le nombre d'instances du système varie en fonction d'un ensemble de règles. Ces dernières sont divisées en deux types : les règles pour augmenter les ressources et les règles pour diminuer les ressources. Les règles suivent en général le template suivant [DRM+10] : une règle peut utiliser une ou plusieurs métriques telles que CPU ou le temps de réponse. Pour une règle (basée sur une métrique m) plusieurs paramètres sont impliqués : un seuil supérieur $thr_{upper,m}$, un seuil inférieur $thr_{lower,m}$ et deux périodes de temps $temps_{upper}$ et $temps_{lower}$ qui définissent la durée pendant laquelle la condition doit être remplie pour déclencher une action de

dimensionnement. La règle associée est définie de la manière suivante :

$$\begin{cases} Si\ m > thr_{upper,m}\ pour\ temps_{upper}\ alors\ faire\ n = n + k_{add} \\ Si\ m < thr_{lower,m}\ pour\ temps_{lower}\ alors\ faire\ n = n - k_{remove} \end{cases} \quad (3.2)$$

Où m est une métrique (CPU par exemple), k_{add} et k_{remove} sont respectivement la capacité à ajouter ou à supprimer.

La plupart des travaux de recherche utilisent seulement deux seuils par métrique : un seuil supérieur et un seuil inférieur (par exemple, 15% et 85% de la charge CPU). Afin d'éviter les oscillations de système, Hasan et al. [HMC+12] ont envisagé d'utiliser un ensemble de quatre seuils et deux durées : un seuil supérieur, un seuil légèrement inférieur au seuil supérieur, un seuil inférieur et un seuil légèrement supérieur du seuil inférieur, et deux durées (en secondes) utilisé pour vérifier la persistance de la valeur de la métrique dessus/dessous des seuils. Simmons et al. [BB10] ont proposé des solutions basées sur des seuils dynamiques. [LBCP09] définit le seuil via une distribution de probabilité alors que Lim et al. [LBC10] ont décrit une technique de seuillage proportionnel via un contrôleur intégral.

La définition de seuils est une tâche par application, et nécessite une maîtrise de la charge du travail. Ainsi cette approche dépend fortement de choix des variables de performance à savoir les métriques.

b) Planification avec garantie

Il s'agit d'une solution basée sur un modèle. Elle considère une ou plusieurs métriques (multi-critères). Parmi les approches utilisées, nous pouvons citer : la théorie des files d'attente, la théorie de jeu, la théorie de contrôle, l'apprentissage par renforcement, la programmation par contraintes et la programmation linéaire. Dans ce qui suit, nous illustrons comment la planification de capacité est modélisée par chaque approche et les limites associées.

Théorie des files d'attente : la théorie des files d'attente est une théorie mathématique relevant du domaine des probabilités [MDA04]. La topologie étagée d'un système est capturée par la modélisation sous forme des files d'attente. Cette modélisation repose sur la probabilité de transition d'une requête d'un client entre les différents étages. Un modèle analytique basé sur la théorie des files d'attente peut prédire le comportement du système. La méthode analyse par valeur moyenne (Mean Value Analysis - MVA) [RL80] est la plus utilisée. Une telle méthode permet d'estimer la latence, la disponibilité et le débit du système en produisant certains paramètres dont la probabilité de transition entre les étages [MDA04]. La planification peut être guidée par les résultats de la méthode MVA.

Cette approche nécessite plusieurs paramètres pour fournir un plan de dimensionnement. Ainsi ces paramètres sont recalculés à chaque évolution de demande.

Théorie des jeux : la théorie des jeux constitue une approche mathématique des problèmes de stratégie [NRTV07]. Elle étudie les situations où les choix de deux protagonistes (joueurs), ou davantage, ont des conséquences pour l'un comme pour l'autre. Un jeu peut être à somme nulle ou plus souvent, à somme non-nulle. Le deux types

sont possibles pour la planification de capacité. L'algorithme de l'équilibre de Nash (équilibre bayésien) [TM10] [WVZX10] est le plus adopté pour calculer la capacité demandée. Il modélise le problème de planification par un ensemble de stratégies et une fonction des gains et le présente sous la forme d'un arbre.

La théorie des jeux présente un modèle de prise de décisions en avenir incertain sans probabilité connue. Le résultat de jeu est souvent sensible aux hypothèses émises, l'information disponible et la rationalité des décisions des joueurs.

Théorie de contrôle : la théorie de contrôle est une branche de l'ingénierie et la mathématique qui traite du comportement des systèmes dynamiques. Il existe trois types de contrôle : boucle ouverte, rétroaction (feedback) et feed-forward. Un contrôleur intégré est capable d'ajuster le nombre de machines virtuelles, par exemple, en fonction d'utilisation moyenne de CPU [LBC10]. Les contrôleurs rétroactives sont les plus adaptés dans le contexte du Cloud. Un contrôleur rétroactive peut être : à gain fixe, adaptative ou prédictif. Les contrôleurs à gain fixe sont très populaires en raison de leurs simplicité mais avec un comportement statique [LBC10]. Un contrôleur adaptif ajuste les paramètres de réglage du régulateur en ligne [AETE12][WXZ+11] alors qu'un contrôleur prédictif calibre le régulateur suite à la prédiction du comportement du système [WXZF11].

Cette approche permet de produire la capacité nécessaire en fonction de la charge du travail. Cependant, le réglage des paramètres de gain peut être une tâche difficile. De plus, comme dans le cas des règles à base des seuils, ce réglage peut provoquer des oscillations du système.

Apprentissage par renforcement : l'apprentissage par renforcement fait référence à une classe de problèmes d'apprentissage automatique [SB98]. Le but est d'apprendre, à partir d'expériences, ce qu'il convient de faire en différentes situations, de façon à optimiser une récompense numérique au cours du temps. A l'exclusion de la charge imprédictible, un système d'apprentissage peut calculer la capacité demandée en se basant sur son historique. Formellement, la base du modèle d'apprentissage par renforcement consiste en : i) un ensemble d'états S de l'agent dans l'environnement, ii) un ensemble d'actions A que l'agent peut effectuer et iii) un ensemble de valeurs scalaires "récompenses" R que l'agent peut obtenir. L'apprentissage par renforcement capture en ligne le modèle de performance d'une application cible sans aucune connaissance à priori. Parmi les méthodes qui modélisent le problème de planification, nous citons le processus de décision markovien (Markov Decision Process) et l'algorithme Q-learning. Pour un dimensionnement horizontal [DKM+11], un état s peut être défini comme (w, u, p), où w est le nombre total de demandes observés sur une période prédéfinie, u est le nombre des VMs du système et p est la performance , par exemple, en terme de temps de réponse moyen. Sinon pour un dimensionnement vertical, un état est défini par l'utilisation du processeur et la mémoire pour chaque VM [RBX+09]. L'ensemble des actions possibles A dépend du type de dimensionnement. La fonction de récompense considère à la fois le coût de ressources et le coût dérivé des violations [DKM+11] [RBX+09].

Nous soulignons que cette approche présente plusieurs limites qui sont principalement : un grand temps d'apprentissage et un grand espace d'états. Des alternatives,

comme l'apprentissage en parallèle [BHD12], réduisent le temps d'apprentissage. Alors que l'utilisation de réseaux de neurones [RBX+09], par exemple, diminue l'effet d'un grand espace d'états.

Programmation par contraintes (PPC) : la PPC est une technique mathématique qui permet la résolution de problèmes combinatoires [RBW06]. La PPC est : i) complète car elle énumère toutes les solutions (satisfaction) ou cherche la meilleure (optimisation), ii) globale comme nous savons ce qui a été fait et iii) extensible comme nous pouvons ajouter de nouvelles contraintes. Un problème de satisfaction de contraintes (CSP : Constraints Satisfaction Problem) est défini formellement par un triplet (X,D,C) représentant :

- variables : $X = (X_1, X_2, ..., X_n)$ un ensemble de variables de problème (au sens « inconnues »),
- domaines : D une fonction qui affecte à chaque variable X_i son domaine $D(X_i)$,
- Contraintes : $C = (C_1, C_2, ..., C_m)$ un ensemble des contraintes. Chaque contrainte C_i est une relation portant sur un sous- ensemble de variables de X restreignant les valeurs que ces variables peuvent prendre simultanément.

Le problème de planification est un problème d'optimisation où l'objectif est de définir la capacité optimale du système face à une demande particulière [dOL12a]. La résolution consiste alors à trouver un tuple de valeurs tel que toutes les contraintes de performance et coût financier soient satisfaites. Il suffit de définir une fonction objective $f : D(X) \longrightarrow \mathbb{R}$. Une solution optimale est alors un triplet de solution du CSP qui minimise (ou maximise) la fonction f.

La planification de capacité avec PPC nécessite un effort de modélisation et en particulier la fonction objective. De plus, un problème de scalabilité est posé.

Programmation linéaire (PL) : la PL est une branche de mathématique permettant de résoudre de problèmes d'optimisation. Comme la modélisation CSP, le problème de planification est modélisé via des variables, des contraintes et un objectif (min/max). Cependant, l'objectif et les contraintes sont décrites par des fonctions linéaires des variables [SH10] [ZPL+12].

La planification de capacité dépend de plusieurs critères (performance, coût financier). Dans ce contexte, la définition d'une fonction objective (linéaire) est une tâche compliquée.

iv) Où

Le "Où" présente la portée de l'action du dimensionnement. Une telle portée est définie par la nature de ressources (calcul, stockage ou répartition de charge). Nous faisons une distinction entre ces ressources comme leurs pré-requis ne sont pas les mêmes lors d'une augmentation de capacité. En effet, la réplication d'une ressource de stockage demande par exemple la synchronisation des données. Ceci n'est pas le cas pour une ressource de calcul. Le dimensionnement peut ajuster une ou plusieurs de ces ressources.

Typiquement, nous pouvons utiliser ces ressources sur une application Web 3-tiers standard (voir Figure 3.8)[5] où le premier tier est un tier de répartition de charge, le deuxième tier est un tier métier (calcul) et le dernier est le tier de base de données (stockage).

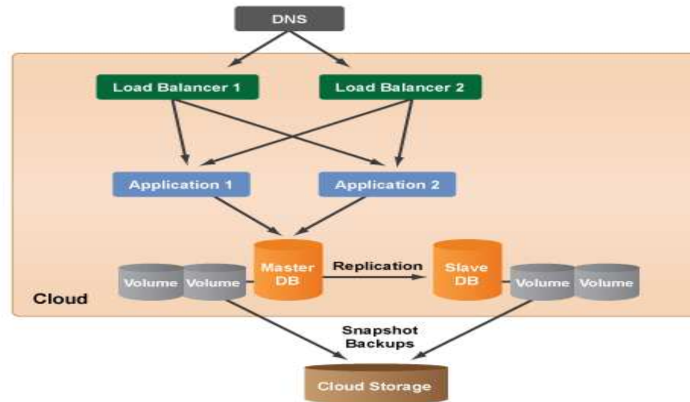


FIGURE 3.8 – Application Web n-tiers

3.3.2 Dimensionnement automatique, la pratique

Dans cette section, nous développons, d'abord les solutions de dimensionnement commerciales et open-source. Ensuite, nous détaillons les initiatives de recherche.

i) Solutions commerciales et open-source

Nous citons Amazon Auto Scaling et Microsoft Enterprise Library Autoscaling Application Block comme solutions commerciales et Scalr et RightScale comme solutions open-source.

Amazon EC2, heure normale : Amazon Auto Scaling[6] permet d'augmenter ou diminuer automatiquement la capacité Amazon EC2 selon les conditions prédéfinies. Il convient particulièrement bien aux applications qui connaissent des variations horaires, journalières, ou hebdomadaires d'utilisation. Amazon Auto Scaling est activé par Amazon CloudWatch et disponible sans frais supplémentaires, au-delà des frais d'Amazon CloudWatch, Amazon EC2, Amazon S3, Amazon Elastic Block Store (EBS), ou encore Elastic Load Balancing. Amazon Auto Scaling est un dimensionnement à base des seuils. Il permet de dimensionner la capacité Amazon EC2 dynamiquement en fonction des mesures Amazon CloudWatch, ou de manière prévisible en suivant un calendrier prédéfini. Le principe est le suivant :

Configuration de lancement : créer une configuration de lancement en utilisant la commande as-create-launch-config. Une configuration de lancement capture les paramètres nécessaires au lancement des nouvelles instances Amazon EC2.

[5] http ://support.rightscale.com/12-Guides/Designers_ Guide/Cloud_ Solution_ Architectures/Cloud_ Computing_ System_ Architecture_ Diagrams
[6] http ://aws.amazon.com/autoscaling/

Groupe Auto Scaling : créer un groupe Auto Scaling en utilisant la commande as-create-auto-scaling-group. Un groupe Auto Scaling est un ensemble d'instances Amazon EC2 auquel vous voulez appliquer certaines conditions d'ajustement.

Politiques de dimensionnement : créer une politique de dimensionnement via la commande as-put-scaling-policy. Une politique décrit une action de dimensionnement à effectuer (ajouter ou supprimer des instances Amazon EC2) :

scaleOut= as-put-scaling-policy scaleOutPolicy –auto-scaling-group group –adjustment=1 –type ChangeInCapacity –cooldown 300 –region region

Cet exemple illustre une politique "scaleOutPolicy" qui permet d'incrémenter de 1 le nombre d'instances du groupe "group".

Alarmes : créer une alarme pour chaque condition de dimensionnement via mon-put-metric-alarm. Les alarmes sont définies en fonction de n'importe quelle mesure collectée par Amazon CloudWatch. Ainsi chaque alarme est basée uniquement sur une mesure (métrique) :

mon-put-metric-alarm scaleOutAlarm –comparison-operator GreaterThanThreshold –evaluation-periods 1 –metric-name Latency –namespace "AWS/ELB" –period 60 –statistic Average –threshold 5 –alarm-actions scaleOut –dimensions "LoadBalancerName=lb" –region region

Cet exemple illustre une alarme basée sur la métrique latence et qui déclenche la politique scaleOutPolicy si la latence dépasse 5 secondes.

Auto Scaling conserve les traces des moments pendant lesquels les conditions ont été remplies et prend automatiquement des mesures d'ajustement correspondantes selon les politiques définies. En plus, Amazon Auto Scaling permet de recevoir des notifications via Amazon Simple Notification Service (SNS) afin d'être alerté d'une situation ou une action.

Microsoft Azure, heure naturelle : Microsoft propose un mécanisme de dimensionnement d'une manière implicite via Microsoft Enterprise Library Autoscaling Application Block (WASABi)[7]. Comme Amazon, la solution de Microsoft est basée sur des règles. WASABi utilise des règles et des actions afin de déterminer la capacité du système pour répondre aux changements de la demande. Nous distinguons deux types de règles : i) règle de contrainte et ii) règle réactive, chacune avec leurs propres actions. Une règle de contrainte est constituée d'une ou plusieurs actions pour définir des valeurs minimum et maximum pour le nombre d'instances d'une cible. Par défaut, une règle est toujours en vigueur sauf si un calendrier est défini. Une règle réactive déclenche une action de dimensionnement lorsque on dépasse un certain seuil. Contrairement à Amazon, Microsoft autorise des règles avec plusieurs métriques. L'exemple suivant illustre un exemple avec deux règles et les actions correspondantes : i) ajouter une instance si la longueur de file d'attente est supérieure à 5 et ii) supprimer une instance si la longueur de file d'attente est inférieure à 5.

```
1 <?xml version="1.0" encoding="utf−8" ?>
2 <rules xmlns="http://schemas.microsoft.com/practices/2011/entlib/autoscaling/rules" enabled="true">
```

[7]http ://msdn.microsoft.com/en-us/library/hh680892(v=pandp.50).aspx

```
3  <constraintRules>
4  <rule name="Default" enabled="true" rank="1">
5  <actions>
6  <range min="2" max="6" target="WasabiRole"/>
7  </actions>
8  </rule>
9  </constraintRules>
10 <reactiveRules>
11 <rule name="Scale Out" enabled="true">
12 <actions>
13 <scale target="WasabiRole" by="1" />
14 </actions>
15 <when>
16 <greaterOrEqual operand="QueueLength_Avg_5m" than="5" />
17 </when>
18 </rule>
19 <rule name="Scale In" enabled="true">
20 <actions>
21 <scale target="WasabiRole" by="−1" />
22 </actions>
23 <when>
24 <less operand="QueueLength_Avg_5m" than="5" />
25 </when>
26 </rule>
27 </reactiveRules>
28 <operands>
29 <queueLength alias="QueueLength_Avg_5m" aggregate="Average" queue="workerqueue" timespan="00:05:00" />
30 </operands>
31 </rules>
```

Scalr [8] : c'est un gestionnaire de nuage open-source. Il supporte les solutions proposées comme Amazon EC2 et Rackspace. Scalr utilise des règles pour gérer le dimensionnement. Par rapport au service de Amazon, la solution de Scalr est mieux adaptée pour réaliser le *scale in* et la réplication de base de données.

RightScale [9] : c'est un gestionnaire de nuage open-source. Il offre des fonctionnalités de dimensionnement dans différents nuages : Amazon EC2, GoGrid, etc. La solution de RightScale est basée sur le "vote". Il s'agit d'un processus de vote démocratique simple selon lequel, si la majorité des instances conviennent sur un *scale out* ou *scale in*, que des mesures soient prises, sinon aucune action ne se produit. RightScale utilise les alertes et les actions associées. Par rapport à Amazon Auto-scaling, RightScale définit une alarme via une ou plusieurs métriques. Aussi, plusieurs actions peuvent être associées à une seule alerte, et qui peuvent être déclenchées périodiquement, ou selon un calendrier défini par un administrateur.

D'autres acteurs du marché comme Google (avec App Engine)[10], GoGrid [11] et Rackspace [12] et CloudStack[13] ont également développé leurs solutions. Pour des raisons de place et/ou de manque d'informations les concernant, ces solutions ne seront pas détaillées.

[8] http ://wiki.scalr.com/display/docs/Scaling
[9] http ://www.rightscale.com/products/automation-engine.php
[10] https ://appengine.google.com/ ?
[11] http ://blog.gogrid.com/tag/auto-scaling/
[12] http ://www.rackspace.com/blog/auto-scaling-in-the-cloud-google-hangout-with-rightscale-recap/
[13] https ://cwiki.apache.org/CLOUDSTACK/autoscaling.html

ii) Initiatives de recherche

De nombreux travaux portent sur le dimensionnement automatique des ressources. Nous reportons dans ce qui suit seulement les travaux directement liés à la dimensionnement de capacité de stockage, calcul et répartion de charge.

a) Stockage (tier de données)

Suite à l'étude des travaux, nous avons pu observer que peu de travaux portent sur le dimensionnement des tiers de données dans le monde du Cloud. Lim et al. [LBC10] ont proposé un contrôleur automatisé pour le dimensionnement horizontal des systèmes de stockage. Ce contrôleur est basé sur un seuillage proportionnel (Proportional thresholding) pour déterminer la taille du cluster de stockage. Il utilise une métrique bas niveau à savoir l'utilisation de CPU. Une fonction de coût est utilisée pour trouver le meilleur compromis entre le temps de l'achèvement du processus de rééquilibrage et la satisfaction des clients. Afin d'éviter le temps de synchronisation des données, Wang et al. [WXZ+11] ont présenté une approche de dimensionnement vertical. Cette approche est modélisée avec le logique floue adaptative (adaptive fuzzy). Elle prévoit, périodiquement, les besoins pour la charge de travail actuelle en s'appuyant sur les données observées. Le nombre des requêtes, la charge CPU et la bande passante I/O disque sont les métriques utilisées pour estimer la capacité de base de données. L'approche répond correctement aux variations dynamiques des requêtes. Cependant, elle est basée sur une politique de backup prédéfinies pour faire face à des situations où la demande de ressources (CPU) du VM est mal estimée. Une méthode de prédiction sera intéressante dans ce cas. Les auteurs ont admis que la théorie de contrôle peut répondre au problème de dimensionnement de base de données dans un environnement Cloud. Ainsi, le traitement d'une VM comme une "boîte grise" est plus adaptée qu'une "boîte noire". Bien que le dimensionnement vertical absorbe le temps de duplication, une approche hybride (horizontal-vertical) serait la meilleure solution pour répondre à la fois aux demandes mais aussi aux contraintes techniques. En effet, un dimensionnement vertical n'est pas fourni par tous les fournisseurs IaaS.

b) Calcul (tier métier)

A l'inverse du tier de données, plusieurs travaux se sont focalisés sur le tier métier à la fois pour le dimensionnement horizontal [BHD12] [DRM+10] [DKM+11] [IKLL12] et vertical [GGW10] [MBS11] [PHS+09] [RBX+09].

Horizontal : Barrett et al. [BHD12] ont proposé une approche Q-learning qui utilise des agents d'apprentissage parallèles. Cette approche permet de réduire le temps nécessaire pour déterminer les politiques optimales tout en apprenant en ligne. Cependant, elle ne résout pas la nécessité d'avoir des bonnes politiques dans les premières phases de l'apprentissage. Dutreilh et al. [DKM+11] ont proposé des politiques appropriées pour les premières étapes ainsi que des accélérations de convergence appliquées tout au long des phases d'apprentissage. Même si certaines de ces propositions étaient connues dans le domaine de l'apprentissage, l'apport essentiel de ce travail était de les intégrer dans un contrôleur et de les programmer comme un workflow automatisé. [DRM+10] ont combiné l'apprentissage avec les règles statiques. Les auteurs ont signalé l'importance de seuils gérés par les règles afin d'éviter les oscillations du système.

Islam et al. [IKLL12] ont présenté un modèle prédictif pour faciliter la gestion de capacité des applications e-commerce. Les auteurs ont sélectionné deux méthodes de séries chronologiques à savoir le réseau neuronal et la régression linéaire. L'objectif du travail était de fournir une meilleure prévision en variant la fenêtre de prédiction. Comme conclusion, les auteurs ont proposé un intervalle de 12 minutes comme fenêtre de prédiction pour absorber le temps d'initiation d'une VM qui est généralement autour de 5-15 min (dans leurs évaluations). Ainsi, ils montrent que le réseau neuronal est plus efficace pour la prévision de l'utilisation des ressources. Le modèle proposé n'intègre pas des objectifs liés au performance et coût de service.

Vertical : Maurer et al. [MBS11] ont présenté une approche basée sur les règles pour le dimensionnement vertical. Ils ont défini cinq niveaux d'escalade d'actions possibles à savoir : modifier la configuration d'une VM, migrer des applications d'une VM à une autre, migrer une VM d'une PM (Physical Machine) à une autre ou créer une nouvelle VM sur une PM appropriée, ajouter une PM, et externaliser (migrer) à un autre fournisseur Cloud. Les auteurs ont utilisé des seuils de menace concernant les métriques de stockage, mémoire, cpu et bande passante pour déclencher une action. Ce travail fournit des résultats meilleurs que un raisonnement à base de cas (Case Based Reasoning CBR). Cependant, les fonctions de coûts utilisées sont très simplifiées et ne se reflètent pas le monde réel.

Padala et al. [PHS+09] ont proposé AutoControl : un système de rétroaction d'allocation des ressources qui s'adapte automatiquement aux changements de charge de travail pour atteindre un SLO. AutoControl est une combinaison d'un modèle de prédiction en ligne basé sur un modèle autorégressif et moyenne mobile ARMA (Auto Regressive Moving Average) du second ordre et un contrôleur adaptatif multi-entrées multi-sorties MIMO (Multiple-Input, Multiple-Output). AutoControl attribue la capacité nécessaire pour garantir le SLA en détectant les goulots d'étranglement. De plus, AutoControl fournit un niveau de différenciation des services en fonction de la priorité des demandes. Cette approche modélise un SLO par un triplet (priorité, métrique, cible), où la priorité représente la priorité de la demande, métrique spécifie la métrique de performance (le temps de réponse), et la cible indique la valeur (seuils) pour une métrique de performance. Les auteurs ont utilisé le CPU et le disque I/O au niveau bas, et le temps de réponse moyen comme mesure de performance haut niveau. Toutefois, AutoControl ne maîtrise pas correctement le temps de migration d'un VM pour absorber les violations.

Rao et al. [RBX+09] ont proposé VCONF une approche pour automatiser le processus de (re)configuration des VMs. VCONF utilise des algorithmes basés sur l'apprentissage par renforcement. Il dirige automatiquement la configuration de chaque machine virtuelle vers l'utilisation optimale. VCONF montre des meilleurs résultats avec des charges de travail homogènes et des VMs hétérogènes. Néanmoins, il existe plusieurs limitations de ce travail à savoir : il n'y a aucune garantie d'optimalité pour les configurations dérivées et la stratégie de collecte de l'échantillon suit une distribution uniforme.

c) Calcul et Répartition de charge (tier métier et tier répartition de charge)

Quelques travaux ont abordé le dimensionnement de tier métier et le tier de répartion de charge comme [HMC+12]. Alors que [IDCJ11] ont traité les tiers métier et données.

Hasan et al. [HMC+12] ont présenté IACRS (Integrated and Autonomic Cloud Resource Scaler) un système de dimensionnement basé sur les règles. A l'opposé des approches standards basées sur les règles, IACRS modélise une règle via quatre seuils et deux durées en secondes qui définissent la durée nécessaire à remplir pour déclencher une action. Cette modélisation suit mieux les tendances du système. Ainsi, les décisions de dimensionnement sont plus fines par rapport à l'utilisation de deux seuils uniquement. Ainsi, IACRS permet de combiné les métriques dans une seule règle de style : ajouter une VM si le temps de réponse et la charge CPU dépassent certains seuils. Toutefois, la limite principale de ce travail est la définition des seuils : définir quatre seuils nécessite une maîtrise de la charge de travail.

Iqbal et al. [IDCJ11] ont proposé un prototype de dimensionnement hybride : des règles réactives (basées sur l'utilisation de CPU) pour le *scale up* et une méthode de séries chronologiques à savoir la régression polynomiale de second degré pour le *scale down*. L'objectif est de satisfaire un temps de réponse maximale spécifique tout en minimisant l'utilisation des ressources. Ce prototype ajuste le nombre d'instances de couches métier et données. Cependant, les auteurs ne considèrent qu'une base de données en lecture seule. De cette manière, ils absorbent le temps de synchronisation des données en cas de duplication de tier de données. Or dans le monde réel les bases de données sont en mode lecture-écriture.

d) Répartition de charge, calcul et stockage (trois-tiers)

Peu des travaux ont exploré le dimensionnement d'une application trois-tier déployée sur le Cloud. Nous exposons les approches selon le dimensionnement à savoir horizontal : [USC+08] [RDG11], vertical : [LZ10] [KCH09] et hybride : [HGGG12].

Horizontal : L'article [RDG11] décrit un algorithme d'allocation des ressources basé sur un ARMA de second ordre. L'algorithme prédit dans une fenêtre prédéfinie la charge de travail d'une application multi-tier pour allouer ou libérer des VMs. Cependant, les auteurs n'abordent pas la définition de fenêtre de prédiction. De plus l'algorithme proposé ne passe pas à l'échelle.

Urgaonkar et al. [USC+08] ont proposé une technique dynamique de provisionnement des applications Internet multi-niveaux tout en respectant des contraintes de temps de réponse lors des pics de charge. Cette technique combine deux méthodes (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 à l'ordre de la minute. La prédiction est basée sur des histogrammes. Ainsi, un modèle de file d'attente flexible est utilisé pour la planification de capacité à chaque niveau de l'application. En particulier, les auteurs utilisent une distribution de probabilité pour estimer la demande de pointe. La charge de travail utilisée est basée sur des sessions. La technique proposée peut traiter la variation de la charge de travail en particulier les pics inattendus. Néanmoins, le temps d'initiation d'une VM est mal maitrisé.

Vertical : Lama et Zhou [LZ10] ont proposé une approche d'allocation autonome basée sur un contrôleur neuronal flou auto-adaptatif. Ce contrôleur combine l'apprentissage et la théorie de contrôle pour assurer le délai de bout-en-bout. Il est capable d'auto-construire sa structure et adapter ses paramètres grâce à l'apprentissage en ligne. Il est robuste aux variations de la charge de travail très dynamique en raison de ses capacités d'auto-adaptation et auto-apprentissage.

Kalyvianaki et al. [KCH09] ont présenté une approche de gestion des ressources qui intègre le filtre de Kalman dans les contrôleurs de rétroaction. Les filtres de Kalman ont été utilisés à la fois pour surveiller l'utilisation du processeur des VMs et guider l'allocation de CPU. Cette approche fonctionne correctement face à la variation de demande. Cependant, la limite de ce travail est qu'il se focalise uniquement sur les ressources CPU.

Hybride : Parmi les solutions, Han et al. [HGGG12] présente une approche pour dimensionner une application multi-tier. Cette approche intègre à la fois le dimensionnement fin (vertical) au niveau des ressources d'une VM (CPU, mémoire) en plus de dimensionnement horizontal. Toutefois, la réservation des ressources est une des limites de cette approche.

d) Autres

Nous pouvons citer encore d'autres travaux où les auteurs n'ont pas mentionné la portée de dimensionnement (calcul, stockage ou repartition de charge). Nous citons en particulier AppScale [CBP+10], CloudScale [SSGW11], SmartScale [DGVV12], Autoflex [AMVBVL+13].

AppScale [CBP+10] : AppScale est un PaaS open-source destiné à la recherche. Il émule Google AppEngine (GAE) enrichie par une extension. Il implémente certains composants qui facilitent le déploiement, le dimensionnement et la tolérance aux pannes d'un système GAE en utilisant des ressources locales. AppScale produit un framework avec lequel les chercheurs et les développeurs peuvent étudier, tester des nouvelles techniques et méthodes.

CloudScale [SSGW11] : CloudScale est un système de dimensionnent automatique des ressources pour les infrastructures multi-locataires. Ce système réduit les violations via la prédiction de la demande de ressources en utilisant la transformée de Fourier (FFT). Il gère la concurrence lorsque les ressources disponibles sont insuffisantes. Il prédit les conflits et propose des solutions (comme la migration). De plus, CloudScale ajuste la fréquence DVFS pour économiser l'énergie sans affecter les SLAs. Bien qu'il soit efficace lors d'expériences, CloudScale présente des limites à savoir principalement : l'utilisation d'un seuil prédéfini comme maximun d'utilisation des ressources (CPU, RAM) par VM.

SmartScale [DGVV12] : SmartScale est une approche pour l'allocation dynamique des ressources. Cette approche combine le dimensionnement vertical et horizontal. Elle utilise une méthode proactive (approximation polynomiale) pour converger rapidement vers la capacité souhaitée. SmartScale est scalable vis-à-vis du nombre de ressources et de la quantité de charge.

Autoflex [AMVBVL+13] : Autoflex est un framework de dimensionnement automatique. Ce framework suit une approche réactive-proactive. Il est basé sur plusieurs méthodes de prédiction. Un mécanisme de sélection permet de choisir la méthode selon des facteurs de prédiction.

3.3.3 Autres approches de gestion de capacité

i) Contrôle d'admission

Le contrôle d'admission est lié de manière générale aux travaux sur l'ordonnancement (section suivante). Il administre directement la quantité de la demande admise et effectivement traitée dans le système. En rejetant des requêtes, le contrôle d'admission préserve les ressources du système et donc la QdS des traitements en cours. Cependant, la conséquence est une baisse immédiate de la disponibilité du service aux clients fortement dommageable pour la qualité générale du service. De nombreux travaux portent sur le contrôle d'admission dans différents contextes. Nous reportons ici seulement les travaux directement liés au nuage en informatique.

Jaideep et al. [DMV10] ont proposé une approche de contrôle d'admission basée sur un système d'apprentissage. Cette approche se concentre sur la précision du contrôle d'admission, mais elle ne considère pas le bénéfice des prestataires de services. Les auteurs utilisent la contrainte de délai pour rejeter certaines demandes. Wu et al. [WKGB12] ont constaté cette limite et ont proposé une solution qui considère également la contrainte de profit pour éviter le gaspillage de ressources. [KCPV12] [KVC11] ont modélisé le contrôle d'admission via un système de modélisation algébrique général (GAMS : General Algebraic Modeling System). La modélisation aborde le problème de l'allocation optimale des services élastiques sur les ressources physiques virtualisées en intégrant une approche probabiliste en termes de garanties de disponibilité. Leontiou et al. [LDD10] ont proposé une approche autonome de contrôle d'admission visant à anticiper la surcharge, garantir un temps de réponse et adapter dynamiquement la charge de travail admis pour être en adéquation avec les changements du capacité du système. Cette approche est fondée sur un système rétroaction adaptatif couplé avec un modèle de file d'attente.

D'autres travaux proposent des solutions basées sur des théories comme la théorie des jeux [AMCC+10].

ii) Ordonnancement

L'ordonnancement applique les règles discriminant les différentes classes de requêtes admises dans le système en choisissant à quelle requête attribuer les ressources. Nous constatons l'existence de plusieurs stratégies d'ordonnancement, certaines étant parfois spécialisées pour des contextes particuliers. Les performances et la QdS du système peuvent être contrôlés en modifiant la stratégie d'ordonnancement et/ou son paramétrage ou en ajustant les niveaux de priorité. L'ordonnancement peut être lié à une requête ou une VM. Nous reportons ici seulement les travaux liés à l'ordonnancement des requêtes. Nous distinguons deux types d'ordonnancement à savoir statique et dynamique, plusieurs contexte en ligne, batch et temps réel et divers objectifs perfor-

mance, coût, énergie etc.

Ghanbari et Othman [GO12] ont proposé PJSC : un nouveau algorithme d'ordonnancement des tâches avec des priorités. Il utilise un processus de hiérarchie analytique et considère des critères multiples pour la prise de décision. Sa complexité est raisonnable en terme de temps. Les auteurs de [dBVB13] ont proposé un ensemble d'algorithmes pour planifier un sac-de-tâches (bag-of-tasks) dans un environnement public ou privé en respectant des deadlines. Ces algorithmes tiennent compte des coûts de transfert et calcul de données ainsi que les contraintes de bande passante réseau. Zhou et al. [ZZCT12] abordent l'aspect énergie. Ils considèrent un scénario où différents types de serveurs fournissent un service multi-classe sans connaître la demande des utilisateurs à l'avance. L'objectif est de fournir un ordonnancement vert. Ils ont formulé le problème d'ordonnancement comme un problème d'optimisation où la contrainte de QoS est exprimée par une formule de probabilité. Ainsi, ils proposent un schéma d'ordonnancement dynamique basé sur un échantillonnage Monte-Carlo. Bossche et al. [VdBVB10] ont proposé une solution pour optimiser l'ordonnancement dans un environnement de Cloud hybride. Il modélise le problème d'ordonnancement par un *binary integer program formulation*. Cet article [LG11] propose un modèle de planification des ressources basées sur l'algorithme de colonie de fourmis. Alors que, [Tay11] ont proposé un algorithme basé sur un algorithme génétique flou (Fuzzy-GA).

Peu de travaux abordent clairement la gestion de niveau de service. [DL11], par exemple, ont proposé une approche d'ordonnancement pour les requêtes en respectant un SLA.

iii) Répartition de charge

La répartition de charge (*load balancing*) est la technique pour distribuer le travail entre les différentes instances. Cette technique permet de réduire les temps de réponse, augmenter le passage à l'échelle et la disponibilité du système. Parmi les politiques les plus utilisées nous reportons ici [NMNAJ12] :

Aléatoire (Random) : une instance est sélectionnée au hasard, sans aucune considération. C'est la politique d'ordonnancement la plus simple mais sans garantie.

Tourniquet (Round-robin RR) : Les requêtes sont affectées aux instances dans un mode cyclique. Cette politique est la plus utilisée dans les solutions commerciales vu sa fiabilité. Elle distribue les demandes de façon uniforme entre les instances.

Weighted round-robin (WRR) : Cette politique généralise la politique RR en attribuant des poids différents aux instances, permettant ainsi aux instances avec un poids plus grand de recevoir un nombre plus grand des demandes. Les poids peuvent être définis à partir des informations telles que les ressources d'une instance (par exemple, la vitesse d'horloge du processeur), ou sa charge actuelle.

Nous pouvons aussi citer *Least-connections* (LC) où la demande est affectée au serveur avec le moins de connexions. Il existe également une version pondérée le WLC.

Dans ce qui suit nous étudions les solutions commerciales et open-source. Ensuite,

nous développons les initiatives de recherche.

a) Solutions commerciales et open-source

Amazon Elastic Load Balancing[14] distribue automatiquement un trafic d'application entrant à travers de multiples instances Amazon EC2. Il permet d'équiper les applications d'une plus grande tolérance à la défaillance, sans heurts, en fournissant la quantité de capacité d'équilibrage de charge nécessaire en réponse au trafic d'application entrant. Elastic Load Balancing détecte les instances qui ne sont pas saines à l'intérieur de son réservoir et change l'itinéraire du trafic vers des instances saines jusqu'à ce que les instances qui ne sont pas saines soient restaurées. Les clients peuvent activer Elastic Load Balancing à l'intérieur d'une seule zone de disponibilité ou sur plusieurs zones, pour des performances applicatives encore plus homogènes. Elastic Load Balancing peut également être utilisé au sein d'un nuage Amazon Virtual Private Cloud (VPC) afin de répartir le trafic entre les différents niveaux applicatifs. Le Tableau 3.3 résume les politiques utilisées par les solutions commerciales et open-source.

Un aspect très important à signaler est la gestion de session (*Session affinity* ou encore *sticky sessions*) qui permet d'affecter les requêtes à la même instance. Une telle option est supportée par Amazon ELB, Gogrid[15] et Rackspace[16] alors que Google App Engine[17], ou Microsoft Azure[18] comme solution propriétaire ou encore Cloud Foundry[19] comme solution open-source ne la supportent pas.

TABLE 3.3 – Bilan de repartiteurs de charge

	Aléatoire	RR	WRR	LC	WLC
Amazon ELB		x			
Microsoft Azure		x			
Google AE		x			
GoGrid			x		x
Rackspace	x	x	x	x	x
Cloud Foundry			x	x	

b) Initiatives de recherche

La plupart des approches considèrent principalement les métriques suivantes : débit, tolérance aux pannes, temps de réponse et passage à l'échelle. La stratégie de répartition de charge peut être liée à une instance [HGSZ10] ou à une tâche [LLM+10] [NPF10] [ZZ10] [LXK+11].

Afin d'équilibrer la répartition des machines virtuelles, Hu et al. [HGSZ10] ont présenté une stratégie basée sur un algorithme génétique. Selon les données historiques,

[14] http ://aws.amazon.com/elasticloadbalancing/
[15] http ://www.gogrid.com/products/infrastructure-load-balancers
[16] http ://www.rackspace.com/cloud/load-balancing/
[17] https ://appengine.google.com/
[18] http ://www.windowsazure.com/en-us/manage/windows/common-tasks/how-to-load-balance-virtual-machines/
[19] http ://www.cloudfoundry.com/

l'état actuel du système et l'algorithme génétique, cette stratégie calcule en amont l'influence de chaque plan de déploiement des ressources nécessaires, puis choisit la solution à travers laquelle elle réalise le meilleur équilibrage de charge et réduit ou évite la migration dynamique. Cette stratégie résout le problème de déséquilibre de charge et le coût élevé de migration par des algorithmes traditionnels. Le traitement de données à large échelle nécessite l'équilibre de charge. Liu et al. [LLM+10] ont proposé (LBVS) une stratégie d'équilibrage de charge pour le stockage virtuel. Cette stratégie fournit un modèle de stockage de données à large échelle en tant que service. Une architecture à trois couches est utilisée pour réaliser la virtualisation du stockage. Également au stockage à large échelle, les jeux en ligne demande l'équilibre de charge sur l'ensemble des serveurs. Nae et al. [NPF10] ont présenté un algorithme dirigé par les événements pour les jeux multi-joueurs en ligne (en temps réel). A partir des événements de la capacité en entrée, l'algorithme analyse l'état global de la session de jeu pour générer les actions d'équilibrage de charge. Parmi les travaux remarquables nous pouvons citer ACCLB [ZZ10] et Join-Idle-Queue [LXK+11]. ACCLB (Ant Colony and Complex Network Theory) [ZZ10] est un mécanisme d'équilibrage de charge basé sur colonie de fourmis et de la théorie des réseaux. Il prend la caractéristique du réseau complexe en considération et il est adaptable aux environnements dynamiques. Ce mécanisme permet une meilleure tolérance aux pannes. Join-Idle-Queue [LXK+11] est un algorithme pour l'équilibrage de charge à large échelle avec des répartiteurs distribués. L'idée est de découpler la découverte des serveurs à faible charge de l'attribution des tâches : d'abord diviser les processeurs inactifs sur les répartiteurs puis, attribuer le travail à chaque processeur afin de réduire la longueur moyenne des files d'attente.

iv) Dégradation de service

La dégradation de service consiste à passer d'un mode normal à un mode dégradé. Elle peut être liée à la fonctionnalité offerte par le service, la QdS associée ou les deux. Un exemple de dégradation de fonctionnalité est implanté de manière visible par le service d'indexation et de recherche bibliographique comme CiteSeer [Cit13], qui limite le nombre et le niveau de détail des réponses renvoyées au client lorsqu'il est trop chargé. Philippe et al. [PDPBG09] ont proposé le concept d'adaptation de niveau de service permettant à certains composants d'une application de se dégrader de manière dynamique. Dans le cadre de serveurs de streaming multimédia, la compression croissante permettra d'économiser la bande passante du réseau au détriment de la qualité du contenu [LH05]. Un autre exemple réside dans le contexte des systèmes de sécurité, où les algorithmes de chiffrement plus simples peuvent nécessiter moins de traitement tout en étant moins sûr [WMZ03]. La dégradation de QdS consiste à passer de l'idéal vers l'acceptable afin d'absorber la charge. La dégradation de QdS doit être bien maîtriser pour répondre aux besoins des clients. Un mixage entre la dégradation de fonctionnalité et la dégradation de QdS permet d'absorber plus de la charge qu'une simple dégradation.

3.3.4 Discussion

Afin d'implémenter la gestion de capacité, différentes approches sont possibles comme le contrôle d'admission, l'ordonnancement, la répartition de charge, la dégradation de service et le dimensionnement des ressources. Ces approches sont complémentaires mais les solutions proposées sont focalisées généralement que sur une seule approche.

Le dimensionnement des ressources est l'implémentation la plus abordée comme elle reflète directement l'élasticité du Cloud. Nous signalons l'absence d'un moyen (exemple : un benchmark), pour évaluer, comparer, tester et améliorer les approches proposées. Les auteurs ont testé leurs solutions dans différents environnements (simulation, infrastructure réelle) avec diverses charges (synthétique, trace réelle). Il est difficile de dire qu'une solution est mieux qu'une autre. Cependant, nous avons proposé une classification des travaux en se basant sur le "Où" (calcul, stockage, répartition de charge), le "Comment" (horizontal, vertical, hybride) et le "Quand" (réactif, proactif, hybride) et le "Combien" (règle, modèle). Le Tableau 3.4 illustre un bilan de quelques travaux étudiés.

TABLE 3.4 – Dimensionnement, les initiatives de recherche

		Comment		Où			Combien		Métriques/QdS	
	solution	vertical	horizontal	répartition	calcul	stockage	règles	modèles	haut niveau	bas niveau
Réactif	[GSLI11]	-	x	-	x	-	x	-	-	1
	[HGGG12]	x	x	x	x	x	x	-	1	-
	[HMC+12]	x	x	x	x	-	x	-	4	1
	[MBS11]	x	-	-	x	-	x	-	-	4
Proactif	[DKM+11]	-	x	-	x	-	-	x	2	1
	[MH11]	-	x	x	x	x	-	x	1	-
	[CDM11]	-	x	-	x	-	-	x	-	1
	[SSGW11]	x	-	-	x	-	x	-	-	2
	[IKLL12]	-	x	-	x	-	-	x	-	1
Hybride	[LZ10]	x	-	x	x	x	-	x	1	1
	[IDCJ11]	-	x	-	x	x	-	x	1	1
	[SGLI11]	-	x	-	x	-	x	-	-	1
	[AETE12]	-	x	-	x	-	-	x	2	-

Combien : Suite à l'étude des travaux courants sur le dimensionnement automatique des ressources, nous avons remarqué que la méthode "règles à base des seuils" est la plus populaire surtout dans les solutions commerciales. En effet, la simplicité de cette technique la rend plus attrayante. Toutefois, la définition des seuils est une tâche par application, et nécessite une maîtrise de la charge de travail. Cette maîtrise se traduit par le choix des variables de performance (métriques) et des seuils associés. De plus, cette méthode peut induire une instabilité du système. Une instabilité qui peut être amortie en utilisant plus de deux seuils statiques par métrique ou des seuils dynamiques.

Concernant les autres méthodes de planification de capacité, la théorie des files d'attente est largement utilisée pour l'évaluation des performances des systèmes. Elle est parfaitement adéquate pour des systèmes considérés comme un ensemble des boîtes noires (pas de détails de fonctionnement interne). Cependant, cette méthode nécessite plusieurs paramètres pour fournir un plan de dimensionnement. L'apprentissage par renforcement est capable de calculer un plan de dimensionnement sans aucune

connaissance à priori. Toutefois cette méthode souffre d'un temps d'apprentissage et d'un espace d'états important. Des alternatives sont proposées pour dépasser ces limites. Le résultat des autres méthodes dépend fortement de la modélisation : théorie de jeux (stratégies), théorie de contrôle (fonction de gain) et programmation par contrainte, programmation linéaire (fonction objective). Une telle modélisation reste une tâche complexe.

Où et Comment : Nous avons remarqué que le tier métier (calcul) est le plus traité. Ainsi, le dimensionnement horizontal est plus utilisé que le dimensionnement vertical. De plus, peu de travaux ont traité tous les tiers d'une application Cloud ou encore qui combine un dimensionnement horizontal et vertical.

Quand : Le dimensionnement des ressources dans le Cloud fait apparaître un certain nombre de défis en induisant diverses contraintes comme le modèle économique, plusieurs types de ressources et un temps d'initialisation des ressources important [MH12]. De tels aspects sont négligés ou mal traités dans les solutions proposées. Une approche réactive n'est pas capable de répondre à ces défis alors qu'une approche prédictive le permet. Cependant, la précision de la prédiction dépend fortement de la taille de la fenêtre d'historique considérée et l'intervalle de prédiction. Une approche hybride (réactive-proactive) peut être une alternative pour absorber les erreurs de prédiction.

Gestion de capacité + SLA : Coté définition de SLA, la plupart des travaux ne formalisent pas la QdS via un SLA voire aucune solution ne présente un SLA clairement. Ils se focalisent sur un objectif en l'exprimant par des contraintes QdS. Nous constatons que les critères de QdS de bas (CPU, par exemple) sont plus utilisés que les critères de QdS de haut niveau (temps de réponse, par exemple). L'utilisation des critères de QdS de bas niveau tel que CPU est un bon indicateur de l'utilisation du système, mais il ne reflète pas clairement si la QdS fournie répond aux besoins des utilisateurs.

Nous avons pu observer que peu de travaux considérent le temps d'initialisation des ressources. Wang et al. [WXZ+11] ont présenté une approche de dimensionnement vertical pour absorber le temps de duplication au niveau tier de données. Cependant, le dimensionnement vertical n'est pas fourni par tous les fournisseurs IaaS. Islam et al. [IKLL12] ont présenté un modèle prédictif pour absorber le temps d'initiation des instances. Toutefois, le modèle proposé n'intègre pas des objectifs liés au performance et coût de service. Hasan et al. [HMC+12] ont proposé un système de dimensionnement basé sur les règles. Une règle est modélisée via quatre seuils pour suivre mieux les tendances du système. Néanmoins, la limite principale de ce travail est la définition des seuils : définir quatre seuils nécessite une maîtrise de la charge de travail.

Le modèle économique est généralement négligé et peu de détails sur les violations, les pénalités et la facturation sont fournis.

3.4 Synthèse

Les approches et les outils présentés dans cet état de l'art offrent des moyens pour maîtriser les contrats de niveau de services. Notre objectif est de combler les lacunes identifiées ci-dessous.

Définition de SLA : suite à notre étude, nous avons pu observer la pauvreté d'expression des langages SLA proposés pour la communauté Cloud (cf. Section 3.1.4). Nous avons signalé principalement les limitations suivantes : l'absence de définition de service multi-niveaux (XaaS) et la gestion fine des pénalités. En effet, les langages existants ne supportent pas la définition de n'importe quel service XaaS à savoir PaaS et IaaS. Ces derniers sont généralement présentés dans des fichiers externes comme OVF. Ensuite, l'instabilité de la QdS est négligée et les modèles économiques (violations, pénalités et facturation) proposés sont très simples ce qui peut s'avérer problématique dans un environnement hautement dynamique comme le Cloud. En réponse à ces limites, nous proposons CSLA : *Cloud Service Level Agreement*. CSLA favorise la formalisation des SLA dans n'importe quel langage (XML, Java,...) pour n'importe quel service XaaS (SaaS, PaaS, IaaS). Notre langage propose un modèle économique novateur pour gérer plus finement les violations. Il sera détaillé dans le chapitre 4.

Dépendances SLA : Après un bref parcours de l'état de l'art sur les dépendances SLA, nous avons remarqué l'absence d'une solution automatique qui simplifie la définition, le calibrage et la négociation d'un SLA à partir des SLAs de niveaux inférieurs dans la "pile" du Cloud (cf. Section 3.2.3). Cependant, les travaux sur les translations de QdS couvrent plusieurs phases dans le cycle de vie de SLA comme le monitoring et la planification des ressources. Ainsi nous distinguons deux catégories top-down et bottom-up. Nous explorons une approche de type bottom-up dans le chapitre 4 basée sur la programmation par contraintes. En s'appuyant sur les préférences de ses clients, notre solution permet au fournisseur SaaS de calibrer le SLA_S et fournit une configuration optimale en terme de type et nombre de ressources IaaS.

Gestion de capacité : Tous les travaux proposés ont contribué de manière significative à l'automatisation du dimensionnement des ressources pour les applications multi-tiers. Cependant, aucun ne supporte les défis du Cloud (cf. Section 3.3.4). En effet, l'implémentation de l'élasticité du Cloud fait apparaître un certain nombre de défis en induisant diverses contraintes comme le modèle économique, plusieurs types de ressources et le temps d'initiation des ressources non négligeable. La gestion de l'élasticité au niveau infrastructure est insuffisante pour répondre aux défis présentés. Dans le chapitre 5, nous introduisons l'élasticité au niveau application en la considérant comme une boîte blanche pour absorber le temps d'initiation des ressources. Nous proposons *RightCapacity* une méthode de planification de capacité dirigée par SLA qui adresse les aspects multi-couches (*cross-layer*). Cette méthode est contrôlée par *BestPolicies* proposant des politiques de gestion efficace de l'élasticité. La gestion des accords de niveau de services d'une manière autonome est indispensable pour faire face au caractère dynamique et imprévisible du Cloud. Ces contributions seront englobées dans notre solution principale HybridScale : un framework de dimensionnement automatique dirigé par SLA implémenté via une boucle de contrôle autonomique MAPE-K [Hor01].

Contributions scientifiques

4

CSLA : accord de niveau de service Cloud

Un langage SLA est un langage qui permet de spécifier le contrat entre client et fournisseur de service portant sur le niveau de QoS que doit fournir le service. Il n'existe pas de standard reconnus par la communauté Cloud en ce qui concerne la description et l'établissement de propriétés de QoS. Certaines initiatives, comme SLA@SOI [WY11], proposent des solutions pour la définition, la négociation, l'établissement et le suivi du SLA. Cependant, les solutions proposées ne prennent pas en compte intrinsèquement l'instabilité de la QdS dans un environnement hautement dynamique comme le Cloud dans lequel les ressources et le réseau fluctuent énormément. De plus, les variations de charge cliente importantes sont parfois difficile à maîtriser pour le fournisseur et peuvent engendrer des violations SLA.

Ce chapitre présente notre contribution CSLA : *Cloud service Level Agreement*. Nous commençons par détailler la spécification du langage CSLA. Ensuite, nous présentons notre solution de calibrage de CSLA pour répondre au problème de dépendances SLA entre différentes couches du Cloud.

Sommaire

4.1 Langage CSLA **72**
- 4.1.1 Nouveautés de CSLA 72
- 4.1.2 Modèle conceptuel 72
- 4.1.3 Modèle économique 78
- 4.1.4 Exemple illustratif 84
- 4.1.5 Cycle de vie de CSLA 87
- 4.1.6 Expérimentations 88

4.2 Dépendances SLA **95**
- 4.2.1 Calibrage de CSLA template, la méthode 96
- 4.2.2 Calibrage de CSLA template, la pratique 98

4.2.3 Expérimentations 101
4.3 Synthèse 104

4.1 Langage CSLA

Cette section présente le langage CSLA et ses apports. Nous commençons par clarifier les nouveautés de CSLA. Ensuite, nous choisissons de présenter la spécification de CSLA via un méta-modèle. La grammaire CSLA est présentée en annexe A. Après, nous détaillons le modèle économique associé à CSLA. Puis, un exemple complet de CSLA est donné. De plus, nous illustrons le cycle de vie CSLA. A la fin de la section, nous proposons des expérimentations pour illustrer les apports de CSLA.

4.1.1 Nouveautés de CSLA

Nous proposons CSLA (*Cloud Service Level Agreement*), un langage pour améliorer la gestion de SLA dans le Cloud, en particulier la gestion des violations. CSLA est inspiré de SLA@SOI [WY11], WSLA [LF03] et WS-Agreement [Aa07]. Ce langage répond à la nécessité de l'interopérabilité dans le domaine du Cloud computing. CSLA suit l'architecture de référence de l'informatique en nuage de l'institut national des standards et de la technologie (NIST) et supporte l'interface OCCI (Open Cloud Computing Interface).

CSLA favorise la formalisation des SLA dans n'importe quel langage (XML, Java, ...) pour n'importe quel service XaaS (SaaS, PaaS, IaaS). Il permet de gérer des garanties multicritères avec et sans priorité pour refléter les préférences du client. Il est facilement extensible en ajoutant des types dérivés.

L'instabilité de la QdS est adressée via de nouvelles propriétés intégrées directement dans notre langage. En effet, CSLA gère les violations à grain plus fin grâce à : la dégradation fonctionnelle (différents modes de service), la dégradation de QdS (via les propriétés *fuzziness* et *confidence*) et des modèles avancés de pénalité.

La conséquence directe d'un tel langage est qu'il permet au fournisseur de service d'affiner ses stratégies de (re)configuration (par exemple, ordonnancement, contrôle d'admission, dimensionnement) pour arbitrer subtilement la gestion de ressources dans un contexte très dynamique. Par exemple, un service d'ordonnancement peut jouer sur une marge d'erreur grâce à la propriété *fuzziness* pour ordonnancer différemment les tâches à exécuter et peut-être éviter une allocation de ressource inutile. L'ensemble de ces nouveautés est détaillé dans les pages suivantes.

4.1.2 Modèle conceptuel

Le modèle conceptuel est présenté via le méta-modèle CSLA et la grammaire associée (en Annexe A). Un exemple complet est déroulé en Section 4.1.4.

i) Méta-modèle CSLA

Le méta-modèle (Figure 4.1) présente le modèle conceptuel du langage. Un contrat est spécifié concrètement sous forme d'une instance de la classe SLA. Cette dernière est

composée de : la durée de validité du contrat, des parties impliquées et d'un template CSLA.

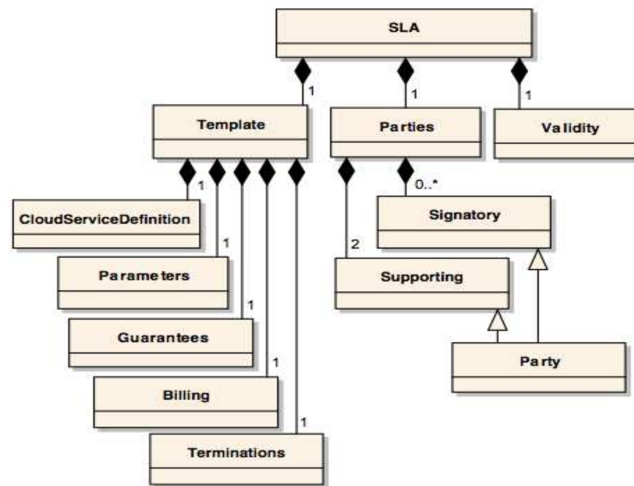


FIGURE 4.1 – Méta-modèle CSLA

a) Validité

La durée de validité d'un contrat (*Validity*) est définie par la date d'entrée en vigueur et la date de fin. Cette dernière peut être modifiée conformément aux conditions de résiliation (*Termination*).

FIGURE 4.2 – Validité

b) Parties

La section *Parties* décrit les parties impliquées dans le contrat. Elle inclut les parties signataires (ou primaires) ainsi que les parties de support (tier de confiance). Deux parties signataires sont impliquées dans le contrat selon leurs rôles : le fournisseur de service et le client. Nous distinguons un autre rôle à savoir le support. Généralement, ce rôle est attribué au fournisseur de service mais parfois au client ou les deux. Par exemple, avec EC2 d'Amazon, le client doit prouver que son service a été indisponible en capturant lui-même la panne et en la documentant en conséquence, avant d'envoyer la preuve à Amazon, le tout dans un délai de 30 jours après l'occurrence de la panne[1]. Le support peut être également assigné à un ou plusieurs tiers de confiance.

La description de la partie résume les propriétés qui sont communes et indépendantes de leur rôle particulier à savoir le nom, et les éléments de contact comme illustré dans la Figure 4.3.

[1]http ://aws.amazon.com/fr/ec2-sla/

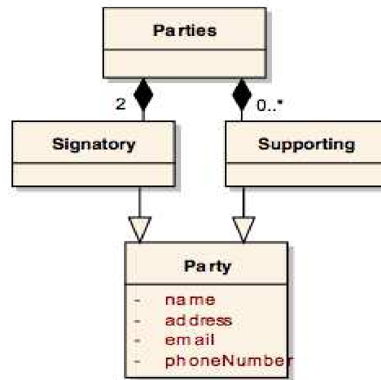


FIGURE 4.3 – Les parties

c) Template

Le template CSLA est un patron (modèle) du contrat. Il est généré avec des paramètres prédéfinis pour s'assurer que les garanties de QdS proposés sont réalistes et réalisables. Une illustration de chacun des concepts est donnée dans la Section 4.1.4. Il est composé de cinq éléments : la définition des services, les paramètres, les garanties, le billing et la terminaison.

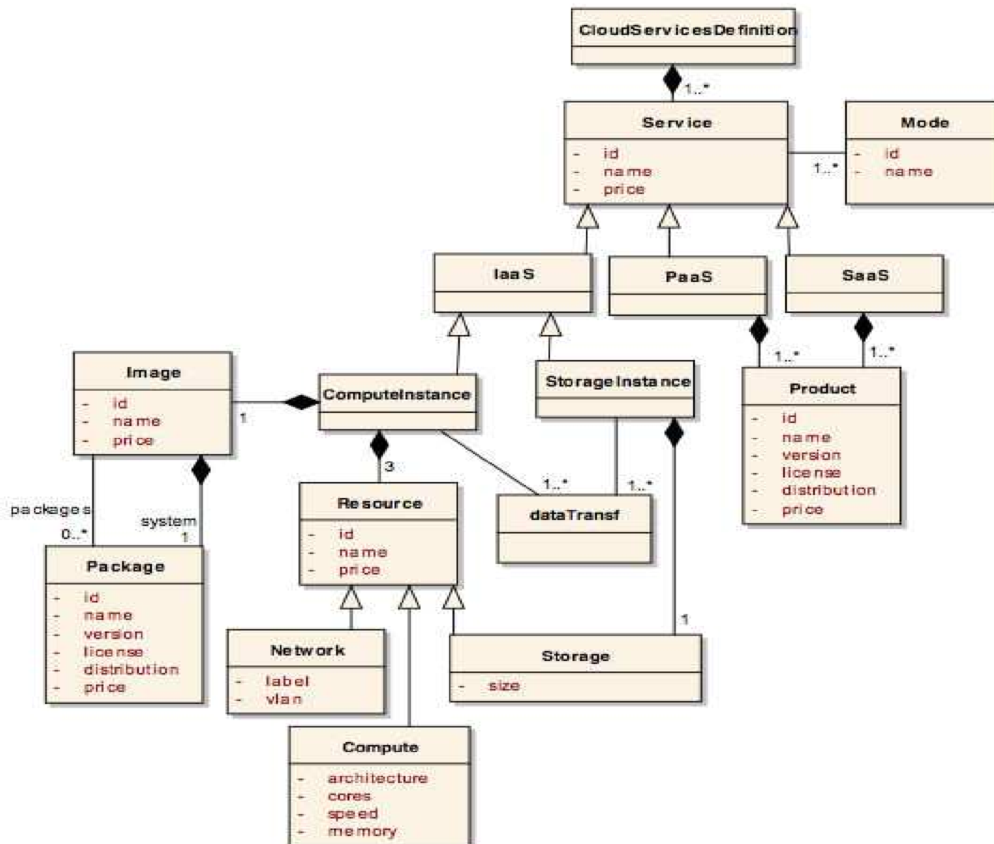


FIGURE 4.4 – Définition des services

La définition des services : CSLA formalise le SLA pour n'importe quel service XaaS. La version actuelle supporte les services suivants : SaaS, PaaS et IaaS. Cepen-

dant, CSLA est facilement extensible en ajoutant des types dérivés de la classe *Service* (voir Figure 4.4). L'originalité de CSLA au niveau définition de service est la **dégradation de fonctionnalité** (*Mode*). Un service (en particulier SaaS et PaaS) peut avoir un ou plusieurs modes de fonctionnement. Par exemple, un fournisseur SaaS peut proposer un service d'informatique décisionnelle (Business Intelligence) avec deux modes : affichage 3D comme mode normale et affichage 2D comme mode dégradé.

L'intérêt majeur de la dégradation de fonctionnalité est de pouvoir être utilisé dans des politiques de dimensionnement du fournisseur de service pour faire face à des pics de charge de courte durée ou encore pour absorber le temps d'initialisation non négligeable des instances. Le fournisseur de service peut, alors, mieux planifier ses ressources mais sans importuner le client. En effet, le prix de service sera ajusté en fonction des modes selon une stratégie gagnant-gagnant. Un tel rapport fonctionnalité-prix doit être à la fois attrayant pour le client final et rentable pour le fournisseur SaaS. Dans CSLA, nous proposons clairement de contractualiser le pourcentage d'utilisation de ce mode par une métrique (cf. Section 4.1.4).

Un service (SaaS ou PaaS) est défini comme un ensemble de produits (solutions logicielles) alors qu'un service IaaS regroupe des instances. CSLA template reconnait deux types d'instances virtuelles : calcul et stockage. Nous modélisons les ressources via le standard OCCI. Ce choix est justifié par la capacité de ce standard de répondre aux différents types de ressources. Il propose une définition de haut niveau. Une ressource est dérivée en trois types : réseau, stockage et calcul. Le réseau est identifié par un *label* et un *vlan*. Le stockage est défini par la taille de stockage. Le calcul est spécifié principalement par l'architecture, la vitesse du processeur, le nombre de cœurs et la taille mémoire. Une instance de calcul est une image plus les trois ressources (réseau, stockage et calcul). Une image est composée d'un ensemble de packages. Sinon, une instance de stockage est réduite à une ressource de stockage. Chaque service est immatriculé par un identifiant et un prix. Le prix d'une instance dépend des ressources mais aussi du type et de la taille des données transférées depuis et vers les ressources allouées.

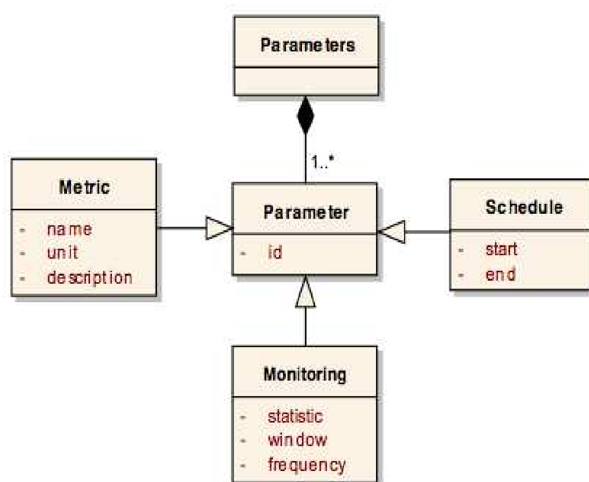


FIGURE 4.5 – Les paramètres

Les paramètres : Les paramètres offrent un moyen pour définir des variables qui sont utilisées dans d'autres sections du contrat. Ils sont utilisés pour factoriser les variables du contrat. Comme illustré dans la Figure 4.5, elles font références à un élé-

ment particulier tel que : une métrique, un monitoring ou un calendrier. Une métrique est définie par un identifiant, une description et une unité. Un monitoring précise la nature d'évaluation d'une métrique à savoir : la statistique (moyenne, maximum, minimum,...), la fenêtre d'évaluation et la fréquence de la collecte de valeurs. Un calendrier délimite une période de monitoring spécifique pendant la durée de validité de contrat.

Garanties : Les garanties présentent le noyau du contrat. Une garantie est définie dans un contexte (scope, exigences).

Un *scope* borne la portée d'une garantie. Il définit l'ensemble des services impliqués.

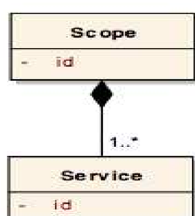


FIGURE 4.6 – Le scope

Les exigences (*Requirements*) présentent les spécifications indispensables pour le bon fonctionnement des services impliqués. Une spécification peut être obligatoire ou optionnelle. Elle décrit un ou plusieurs éléments techniques préalables à remplir comme la version d'un navigateur par exemple.

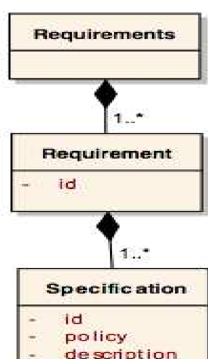


FIGURE 4.7 – Les exigences

Les termes de garanties détaillent les objectifs de niveau de service (SLO) du contrat. Un terme de garantie est une composition des termes ou objectifs via des opérateurs tels que *And* et *Or*. La Figure 4.8 illustre les termes de garanties.

Un objectif (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, un comparateur et un seuil. Cette contrainte a une priorité (*priority*) pour refléter les préférences du client. Nous introduisons la **dégradation de QdS** dans le contrat SLA via les propriétés *fuzzinessValue*, *fuzzinessPercentage* et *confidence*. Un objectif doit être garanti selon un calendrier pendant la durée de validité de contrat avec un pourcentage de confiance. La confiance (*confidence*) est

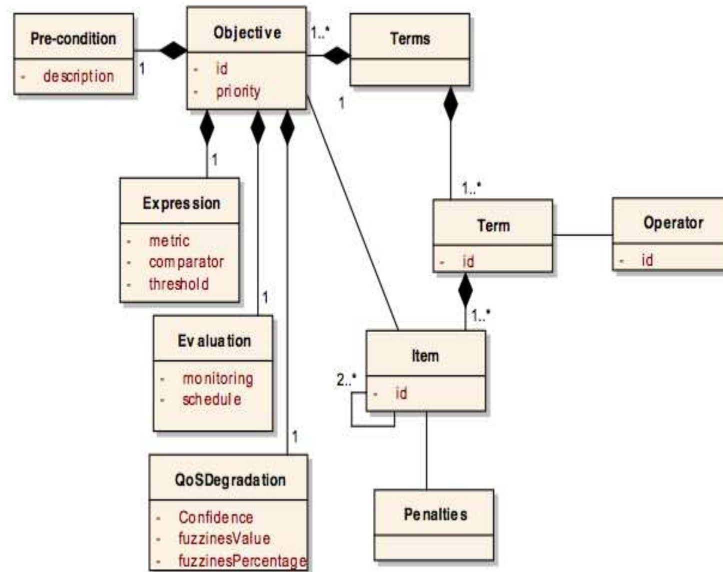


FIGURE 4.8 – Les termes de garanties

le rapport (cas adéquats/ nombre de cas) (cf. Section 4.1.3). Parmi les cas adéquats, un pourcentage (*fuzzinessPercentage*) des requêtes peut utiliser une marge d'erreur à savoir la *fuzzinessValue*. L'avantage direct de cette dégradation de QdS est la capacité d'absorber les pics imprévisibles tout en maintenant un certain degré de QdS.

Avec telle proposition, le fournisseur de service peut affiner ses stratégies mais sans importuner le client car ce dernier sera facturé via un modèle économique avancé (cf. Section 4.1.3). Ce modèle harmonise les pénalités avec la dégradation de QdS afin de fournir un bon rapport qualité-prix qui est la fois attrayant pour le client final et rentable pour le fournisseur SaaS.

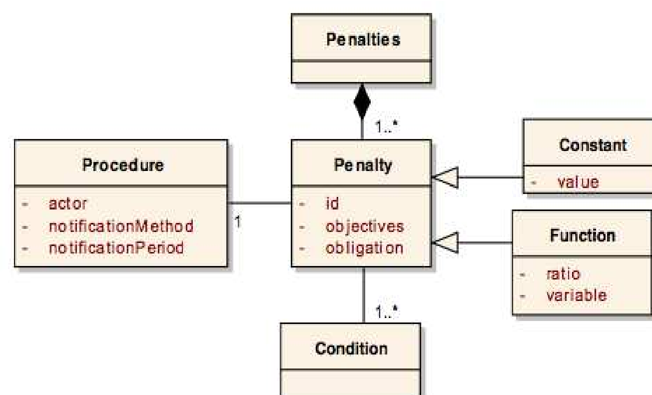


FIGURE 4.9 – Les Pénalités

CSLA gère finement les violations comme la montre la Figure 4.9. Les pénalités sont des moyens de récompenser le client pour tolérer des violations. Une pénalité est définie par une constante ou une fonction. La procédure indique l'acteur responsable de la détection de violation ainsi que la méthode et la période de notification.

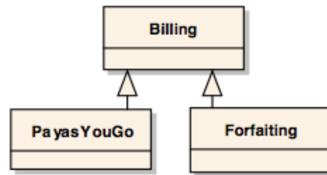


FIGURE 4.10 – Le billing

Le billing : La facturation suit deux types : un forfait ou *pay as you go*. Le prix du service dépend des modes de fonctionnement, *confidence*, *fuzziness*, et pénalités. Dans le cadre de notre travail, nous focalisons l'accent sur le billing *pay as you go*.

La terminaison : Un contrat peut être terminé naturellement selon la date de fin ou suite à l'exécution d'une résiliation conformément à la section *Termination*. CSLA offre la possibilité de formaliser des clauses de terminaison. Une terminaison est caractérisée par l'initiateur, le type de résiliation, la méthode de notification et les frais associés.

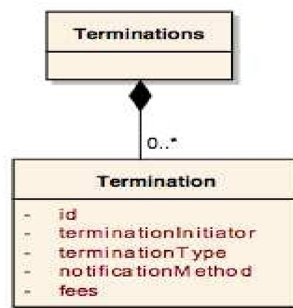


FIGURE 4.11 – La terminaison

ii) Grammaire CSLA

La grammaire CSLA présente la syntaxe abstraite du langage. Nous avons joint en annexe A la spécification du langage CSLA avec un schéma XML qui donne une vue plus détaillée sur la grammaire CSLA.

4.1.3 Modèle économique

CSLA propose un modèle économique novateur pour gérer plus finement les violations de contrat. Dans ce qui suit, nous expliquons, d'abord, le profit du fournisseur de service. Ensuite, nous développons le modèle de pénalité. Enfin, nous illustrons le coût d'une pénalité.

i) Profit du fournisseur de service

Nous proposons un modèle économique avancé qui supporte la gestion à grain fin de violations. La fourniture d'un service engendre un revenu, qui est basé sur une dépense. Le profit du fournisseur est défini par l'équation 4.1 :

$$profit = income - expense \tag{4.1}$$

Le revenu (*income*) est calculé par équation 4.1 :

$$income = \sum_{c=1}^{C} PrServ_c + creditServ \tag{4.2}$$

Où C est le nombre des clients du service et $PrServ_c$ est le montant payé par le client c. Cela permet de supporter les services multi locataires. A l'exception du client final (end-user), chaque acteur dans la hiérarchie XaaS est un client-fournisseur. Le SaaS provider, par exemple, est un client du fournisseur IaaS. Par conséquent, il pourra recevoir un crédit (*creditServ*) comme récompense en cas des violations niveau IaaS.

La dépense (*expense*) d'un service (équation 4.3) est composée des dépenses de ressources allouées (*Resources*), des pénalités payées (*Penalties*) et des dépenses diverses (*Others*) liées à l'administration du service.

$$expense = Resources + Penalties + Others \tag{4.3}$$

Le coût des ressources (équation 4.4) dépend du type de ressource et de la durée d'allocation. Nous ajoutons encore le coût lié à l'utilisation des logiciels propriétaires.

$$Ressources = \sum_{r=1}^{R} PrRes_r.duration_r + \sum_{l=1}^{L} PrLog_l \tag{4.4}$$

Où $PrRes_r$ est le prix de la ressource r par une unité de temps (heure pour Amazon EC2 par exemple), la durée (*duration*) est la période d'allocation, et $PrLog_l$ est le prix de logiciel l ainsi que le prix de la licence associée.

Nous proposons l'équation 4.5 pour calculer les pénalités associés à un service.

$$Penalites = \sum_{p=1}^{P} Penalty_p \tag{4.5}$$

Où P est le nombre total des violations, $Penalty_p$ est le coût de pénalité p. Notre modèle de pénalité est défini en fonction de plusieurs paramètres à savoir : *fuzziness*, *confidence* et le coût de pénalité. En effet, le nombre des violations dépend de principalement des propriétés *fuzziness* et *confidence* alors que le coût d'une violation dépend de modèle de coût utilisé (constante, fonction).

En plus du coût des ressources et pénalités, nous accumulons un coût supplémentaire (*Others*) lié à l'administration de service. Il est calculé via l'équation 4.6 :

$$Others = \sum_{o=1}^{O} other_o \tag{4.6}$$

Où O est le nombre de diverses dépenses supplémentaires et *other* est le coût associé à chacune de ces dépenses.

ii) Évaluation d'un objectif

Afin d'évaluer un objectif (SLO), nous introduisons deux concepts (voir Figure 4.12) : *per-interval* : l'objectif sera évalué par intervalle de temps, et *per-request* : l'objectif sera évalué par requête. En cas de violation d'un objectif, une pénalité est payée.

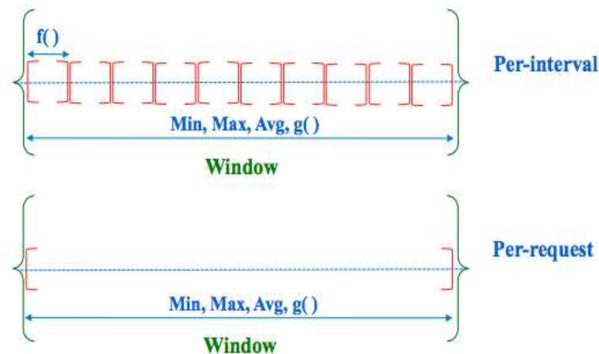


FIGURE 4.12 – Per-interval, Per-request

Nous distinguons 3 états d'une requête (ou un intervalle) à savoir : idéal, dégradé, et inadéquat comme illustre la Figure 4.13. Nous utilisons le terme "idéal" dans la suite de la thèse pour désigner que le seuil (*threshold*) de l'objectif est bien respecté. Le terme "dégradé" signifie une dégradation de QdS. Cette dégradation consiste à utiliser une marge d'erreur *fuzzinessValue*. Au-delà de cette marge, il s'agit d'un état inadéquat.

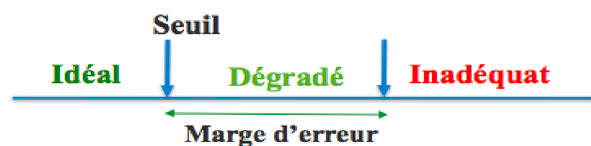


FIGURE 4.13 – idéal, dégradation et inadéquat

L'évaluation est basée sur les propriétés de *fuzziness* et *confidence* afin d'identifier les requêtes (ou intervalles) sans pénalité et les requêtes (ou intervalles) avec pénalités. La *confidence* indique le pourcentage des requêtes adéquates (ou intervalles adéquats) dans une fenêtre de temps. Ce pourcentage regroupe les états idéal et dégradé. Cependant la dégradation de QdS est bornée via *fuzzinessPercentage*. La Figure 4.14 illustre les cas possibles.

Soit l'exemple illustratif suivant pour souligner notre méthode d'évaluation. Sur une fenêtre de 30 minutes, nous avons 100 requêtes de client *c*. L'objectif "temps de réponse" est évalué selon le concept *per-request*. L'objectif "disponibilité" est évalué par le concept *per-interval* avec 3 minutes.

Cas 1 : Un seul objective Le temps de réponse doit être inférieur à 3 secondes avec une *fuzziness* égale à 0,5 seconde. Soit la confiance associée égale à 90% parmi laquelle 10% utilise la dégradation de QdS. Pour simplifier la lecture nous utilisons le pseudo-code suivant pour illustrer cet objectif (SLO) :

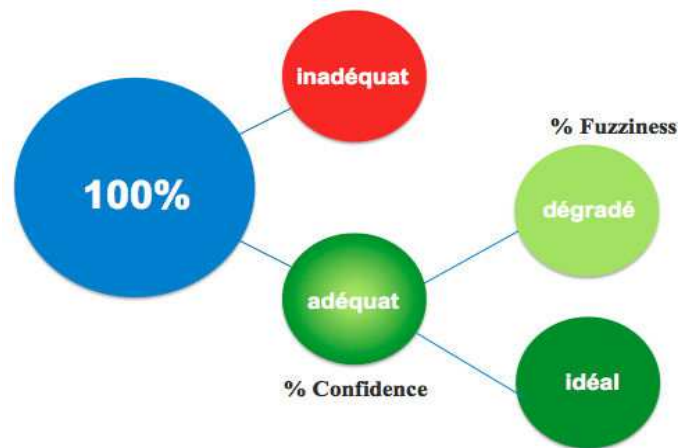


FIGURE 4.14 – pourcentage des requêtes

for-each request in window of 30 mn : Rt<3s where confidence=90, fuzziness-value=0,5, fuzziness-percentage=10.

Une requête avec un temps de réponse supérieur à 3 secondes et inférieur à 3,5 secondes est acceptée s'il y au plus 8 requêtes dégradées. Sinon elle est considérée comme inadéquate. Si le nombre de requêtes inadéquates est inférieur à 10, elle sera acceptée sans pénalité sinon une pénalité sera déclenchée. Le cas limite est d'avoir 81 requêtes avec un temps de réponse inférieur à 3 secondes, 9 requêtes avec un temps de réponse entre 3 et 3,5 secondes et 10 avec un temps de réponse supérieur à 3,5 secondes.

Cas 2 : plus d'un objectif Soit le même objectif de temps de réponse combiné avec un objectif de disponibilité via l'opérateur "*And*". La disponibilité minimum doit être supérieure à 98% avec une marge de 1% utilisée dans 25% des cas adéquats. La confiance est égale à 80%. Soit le pseudo-code de l'objectif de disponibilité :

for-each interval of 3mn in window of 30mn : Min(Av)>98% where confidence=80, fuzziness-value=1, fuzziness-percentage=25.

Dans ce cas, la disponibilité est calculée comme le rapport entre le total des requêtes envoyées par un client *c* et les requêtes rejetées dans un intervalle de 3 minutes. Un tel intervalle avec un taux de rejet inférieur à 2% est l'idéal. L'intervalle dont le taux de rejet est entre 2% et 3% est considéré comme dégradé dans la limite définie par la confiance et le pourcentage de *fuzziness*. Sur la fenêtre de 30 minutes, nous distinguons 10 intervalles de 3 minutes. Le cas limite est d'avoir 6 intervalles idéals avec une disponibilité supérieure à 98%, 2 intervalles dégradés avec une disponibilité entre 98% et 97% et 2 intervalles inadéquats avec une disponibilité inférieure à 97%. Sinon chaque dépassement déclenche une pénalité car c'est considéré comme une violation.

En utilisant l'opérateur "And", le nombre de violations peut être incrémenté suite à une violation de l'un des objectifs. Un intervalle avec une disponibilité supérieure à 98% et un temps de réponse minimum supérieur à 3,5 secondes est considéré comme un inadéquat (acceptable ou non selon les confiances). Également, un temps de réponse inférieur à 3 secondes mais dans un intervalle de disponibilité inférieur à 97% est considéré comme inadéquat.

Le temps de réponse peut être aussi évalué avec le concept *per-interval* où le temps de réponse est considéré comme le maximum des temps de réponse par intervalle de 3 minutes. Soit le pseudo-code associé :

for-each interval of 3mn in window of 30mn : Max(Rt)<3s where confidence=90, fuzziness-value=3, fuzziness-percentage=10.

Le pourcentage de *fuzziness* permet de cadrer l'utilisation de la dégradation de QdS. Nous distinguons en particulier deux cas remarquables : i) une valeur de *fuzziness* très petite avec un pourcentage d'utilisation important et ii) l'inverse c-à-d une valeur de *fuzziness* grande avec un pourcentage faible. La valeur de *confidence* dépend de la nature de service. La valeur 100% peut être vu comme la norme.

Aprés avoir illustré notre modèle d'évaluation par des exemples, nous développons le processus d'évaluation des objectifs. En cas d'évaluation par requête (*per-request*), l'objectif peut être évalué à chaque instant t alors que pour l'évaluation par intervalle (*per-interval*), l'évaluation est effectuée à la fin de chaque intervalle. Une telle évaluation (dite évaluation initiale) permet de classer une requête comme idéal, dégradé ou inadéquat. L'évaluation finale permet de vérifier un objectif dans une fenêtre de temps en appliquant les propriétés : *fuzziness* et *confidence* sur les résultats de l'évaluation initiale. Cette évaluation permet d'identifier les dégradations et inadéquations acceptées (pas de pénalité) des dégradations et inadéquations non acceptées (qui déclenchent des pénalités).

L'algorithme 1 présente la fonction qui consiste à évaluer *per-interval* un objectif dans fenêtre de temps. Dans la ligne 3, f est la fonction utilisée pour calculer une métrique dans un intervalle. Dans notre cas, la disponibilité, par exemple, est calculée comme le rapport entre le total des requêtes envoyées par un client et les requêtes rejetées dans l'intervalle de temps.

Algorithme 1: evaluateObjective

```
Input : f, threshold, comparator, fuzziness, window, frequency
Output : ideal, degraded, inadequate
1 for i = 1 to windows do
2     ideal=degradation=violation=0;
3     if f(observedValues_i) comparator threshold then
4         ideal ++;
5     else if f(observedValues_i) comparator threshold +/- fuzziness then
6         degraded ++;
7     else
8         inadequate ++ ;
9 return ideal, degraded, inadequate
```

L'équation 4.7 illustre le résultat d'une évaluation finale. Un tel résultat est composé de cinq valeurs qui sont : les requêtes (ou intervalles) qui respectent l'objectif de niveau de service (*ideal*, $degraded_a$, $inadequate_a$) et les requêtes (ou intervalles) qui violent la clause ($degraded_r$, $inadequate_r$). Pour conclure que un tel SLA est garanti, il faut que les variables $degraded_r$, $inadequate_r$ soient égales à zéro.

$$checkObjective = \begin{cases} degraded_a = min((((total * confidence)/100) * fuzzPercentage)/100, degraded) \\ degraded_r = degraded - degraded_a \\ inadequate_a = min((total * (100 - confidence))/100, inadequate) \\ inadequate_r = inadequate - inadequate_a \end{cases} \quad (4.7)$$

Où $total = ideal + degradation + inadequate$ et $ideal$, $degraded$ et $inadequate$ sont les sorties de l'algorithme $evaluateObjective(...)$.

iii) Coût d'une pénalité

Une fois considérée comme une requête non acceptée (ou un intervalle non accepté) un coût de pénalité sera attribué. Nous proposons un modèle de coût allant d'une constante à une fonction :

Pénalité par objectif : une constante (équation 4.8) ou une fonction (équation 4.9) sont définies selon la priorité de l'objectif.

$$penalty = const \quad (4.8)$$

$$penalty = \alpha + \beta dt \quad (4.9)$$

Où α est une constante, β est un ratio et dt est une variable. dt est l'écart entre le seuil idéal (ou acceptable) souhaité et la valeur courante. Pour un objectif de temps de réponse le dt est le délai d'une requête (retard). La Figure 4.15 illustre un exemple [IGC04].

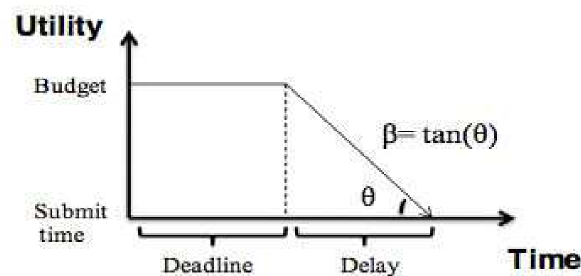


FIGURE 4.15 – Coût d'une pénalite

Pénalité pour une combinaison des objectifs : pareillement, une constante ou une fonction (équation 4.10) sont définies selon la priorité de l'objectif. Une normalisation peut être utile pour aligner les variables de chaque objectif dans la même fonction.

$$penalty = \alpha + \beta . \sum_{i=1}^{n} priority_i . dt_i \quad (4.10)$$

Où n est le nombre des objectifs, $priority_i$ est la priorité de l'objectif i et dt_i est la valeur normalisée de la variable utilisé pour l'objectif i.

4.1.4 Exemple illustratif

Cette section décrit un exemple de CSLA via une syntaxe concrète à savoir le XML. Cet exemple est un SLA entre un fournisseur de SaaS et un client final. Dans ce qui suit, nous détaillons l'exemple en montrant quelques atouts fournis par CSLA. Nous exposons en particulier la gestion fine des violations.

```
1 <csla:CSLA agreedAt="01−06−2013" id="sla1" template="gold" xmlns:csla="http://www.inria.fr/cslamodel">
```

Tous les éléments CSLA sont définis dans l'espace de nom *csla*. L'élément *CSLA* est l'élément racine.

i) Validité

```
1  <csla:validity effectiveFrom="01−06−2013" effectiveUntil="01−07−2013"/>
```

Le présent contrat entre en vigueur le 01-06-2013 et se termine le 01-07-2013 ou suite à une terminaison conformément à la section *Termination*.

ii) Parties

```
1   <csla:parties>
2     <csla:cloudProvider>
3       <csla:name> providerName</csla:name>
4       <csla:contact>
5         <csla:address> address </csla:address>
6         <csla:email> email@email.fr </csla:email>
7         <csla:phoneNumber> +33 (0)2 00 00 00 00</csla:phoneNumber>
8       </csla:contact>
9     </csla:cloudProvider>
10    <csla:cloudConsumer>
11      <csla:name> Yousri KOUKI</csla:name>
12      <csla:contact>
13        <csla:address> 4, rue Alfred Kastler. 44307 Nantes </csla:address>
14        <csla:email> yousri.kouki@gmail.com</csla:email>
15        <csla:phoneNumber> +33 (0)2 51 85 85 25 </csla:phoneNumber>
16      </csla:contact>
17    </csla:cloudConsumer>
18  </csla:parties>
```

Les parties impliquées dans ce contrat sont identifiées par un rôle (client ou fournisseur) et les coordonnées.

iii) Template

Le template suivant regroupe la définition des services, les paramètres, les garanties, le billing et la terminaison.

Services

```
1  <csla:template name="gold" version="1">
2    <csla:cloudServices>
3      <csla:cloudService>
4        <csla:software>
5          <csla:service id="S1" name="service S1" version="v1"
6                        distribution="distribution" license="license" mode="2"  price="1 euro/request"/>
7            <csla:mode id="S1−M1" name="mode M1" description="normal" >
8            <csla:mode id="S1−M2" name="Mode M2" description="degraded">
9          </csla:service>
10       </csla:software>
11     </csla:cloudService>
12   </csla:cloudServices>
```

Le fournisseur propose un service *S1*. Ce dernier est fourni selon deux modes : un mode normal (*M1*) et un mode dégradé (*M2*).

Paramètres

```
1  <csla:parameters>
2          <csla:metric id="Rt" name="Response Time" unit="second">
3            <metric−description> ... </metric−description>
4          </csla:metric>
5          <csla:metric id="Av" name="Availability" unit="\%">
6            <metric−description> ... </metric−description>
7          </csla:metric>
8          <csla:metric id="Mu" name="Mode usage" unit="\%">
9            <metric−description> ... </metric−description>
10         </csla:metric>
11         <csla:monitoring id ="Mon−1" statistic="max" window="10 minutes" frequency="1"/>
12         <csla:monitoring id ="Mon−2" statistic="min" window="10 minutes" frequency="1"/>
13         <csla:schedule id ="Sch−1" start="8:00am" end="10:00am"/>
14       </csla:parameters>
```

Les paramètres utilisés dans ce contrat sont les métriques : temps de réponse, disponibilité et taux d'utilisation d'un mode. En effet, nous proposons une métrique mesurable (*Mode Usage - Mu*) pour maîtriser et évaluer l'utilisation de la dégradation de fonctionnalité. Les autres paramètres sont le monitoring : *Mon-1* et *Mon-2* et le calendrier *Sch-1*. Le monitoring *Mon-1*, par exemple, calcule le maximum sur une fenêtre de 10 minutes des valeurs collectées chaque minute.

Garanties
Scope

```
1  <csla:guarantees>
2          <csla:Guarantee id="G−1">
3            <csla:scope id="Sc1">
4              <csla:service id="S1" subid="S1−M1"/>
5              <csla:service id="S1" subid="S1−M2"/>
6            </csla:scope>
```

La garantie *G-1* est spécifique pour le service S-1 en mode normal et dégradé.

Exigences

```
1  <csla:requirements>
2              <csla:Requirement id="R1">
3                <csla:Specification id="Sp1" policy="Required">
4                Flash Player v10.1 or above </csla:Specification>
5                <csla:Specification id="Sp2" policy="Required">
6                Screen Resolution 1280 x 800 pixels </csla:Specification>
7                <csla:Specification id="Sp3" policy="Optional">
8                32−bit color depth </csla:Specification>
9              </csla:Requirement>
10           </csla:requirements>
```

Pour le fonctionnement du service *S-1*, il faut avoir une version de flash player supérieur à v10.1 et une résolution d'écran de 1280x800. Ainsi il est préférable d'avoir 32-bit pour la couleur.

Termes de garanties

```
1  <csla:terms>
2              <csla:term id="T1" operator="and">
3                <csla:item id="responseTimeTerm"/>
4                <csla:item id="availabilityTerm"/>
5                <csla:item id="modeTerm"/>
6              </csla:term>
7              <csla:objective id="responseTimeTerm" priority="1" actor="provider">
8                <csla:precondition policy="Required">
9                  <csla:description> Data size less than 1 TB</csla:description>
```

```
10            </csla:precondition>
11            <csla:expression metric="Rt" comparator="lt"
12                             threshold="3" unit="second"
13                             monitoring="Mon−1" schedule="Sch−1"
14                             Confidence="99" fuzziness−value="0,2"
15                             fuzziness−percentage="10"/>
16          </csla:objective>
17          <csla:objective id="availabilityTerm" priority="2" actor="provider">
18            <csla:expression metric="Av" comparator="gt"
19                             threshold="98" unit="\%"
20                             monitoring="Mon−2"
21                             Confidence="99" fuzziness−value="1"
22                             fuzziness−percentage="5"/>
23          </csla:objective>
24           <csla:objective id="modeTerm" priority="3" actor="provider">
25            <csla:expression metric="Mu(S1−M2)" comparator="lt"
26                             threshold="10" unit="\%"
27                             monitoring="Mon−1"
28                             Confidence="99" fuzziness−value="2"
29                             fuzziness−percentage="5"/>
30          </csla:objective>
31        </csla:terms>
```

Les termes de garanties sont combinés avec un opérateur *And*. Il y a trois objectifs à savoir : objectif de temps de réponse (*responseTimeTerm*), objectif de disponibilité (*availabilityTerm*) et objectif de taux d'utilisation des modes (*modeTerm*).

L'objectif de temps de réponse, par exemple, est un objectif de priorité 1. Il spécifie que le temps de réponse doit être inférieur à 3 secondes. L'évaluation de la métrique temps de réponse est effectuée selon le monitoring *Mon-1* : c'est un minimum des valeurs récoltées chaque minute sur une fenêtre de 10 minutes. Cet objectif doit être atteint chaque jour entre 8h et 10h pendant la validité de contrat avec une confiance de 99%. Une marge de 0,2 seconde est acceptable comme une dégradation de QdS avec un pourcentage de 10%, c.-à-d. parmi les requêtes adéquates, il est autorisé d'avoir 10% dégradées.

L'objectif de taux d'utilisation des modes indique que le mode *M2* sera utilisé au plus sur 10% des requêtes sur une fenêtre de 10 minutes (*Mon-1*). La confiance de cet objectif est égale à 99%. Sur 10 minutes une dégradation de QoS (2% des requêtes) est possible dans l'ordre de 5% de la *confidence*.

Pénalités

```
1         <csla:penalties>
2           <csla:Penalty id="p−Rt" objective="responseTimeTerm" condition="violation" obligation="provider">
3             <csla:Function ratio="0,5" variable="delais" unit="second">
4                 <csla:Description> ... </csla:Description>
5             </csla:Function>
6             <csla:Procedure actor="provider" notificationMethod="e−mail" notificationPeriod="7 days">
7               <csla:violationDescription/>
8             </csla:Procedure>
9           </csla:Penalty>
10          <csla:Penalty id="p−Av" objective="availabilityTerm" condition="violation" actor="provider">
11            <csla:Constant value="0,1" unit="euro/request"/>
12            <csla:Procedure actor="provider" notificationMethod="e−mail" notificationPeriod="7 days">
13              <csla:violationDescription/>
14            </csla:Procedure>
15          </csla:Penalty>
16          <csla:Penalty id="p−Mu" objective="modeTerm" condition="violation" obligation="provider">
17             <csla:Constant value="0,1" unit="euro/request"/>
18            <csla:Procedure actor="provider" notificationMethod="e−mail" notificationPeriod="7 days">
19              <csla:violationDescription/>
20            </csla:Procedure>
21          </csla:Penalty>
22        </csla:penalties>
23      </csla:Guarantee>
```

```
24        </csla:guarantees>
```

Les pénalités sont liées à chaque objectif. Si l'objectif de temps de réponse est violé une pénalité de type fonction est exécutée. Alors que la violation des autres objectifs déclenche une pénalité constante 0,1 euro/requête.

Billing

```
1        <csla:billing>
2          <csla:payasYouGo>
3            <csla:description> Pay as You Go</csla:description>
4          </csla:payasYouGo>
5        </csla:billing>
```

La facturation est basé sur le modèle Pay as You Go. Le prix de service est 1 €/requête. Il sera ajusté en fonction des pénalités.

Terminaison

```
1      <csla:terminations>
2          <csla:termination id="ter1" notificationMethod="e−mail" fees="20 euros"
3                              notificationPeriod="15 days" terminationInitiator="consumer"
4                              terminationType="Voluntary Termination">
5            <csla:terminationDescription/>
6          </csla:termination>
7        </csla:terminations>
8      </csla:template>
9    </csla:CSLA>
```

Le client a le droit de résilier le contrat avec des frais de résiliation. Il suffit juste de notifier le fournisseur avec un mail.

4.1.5 Cycle de vie de CSLA

CSLA couvre tout le cycle de vie de SLA. Dans cette section, nous proposons un cycle de vie simplifié. Ce dernier contient trois phases principales : l'établissement de SLA, la configuration du Cloud et l'ajustement du Cloud.

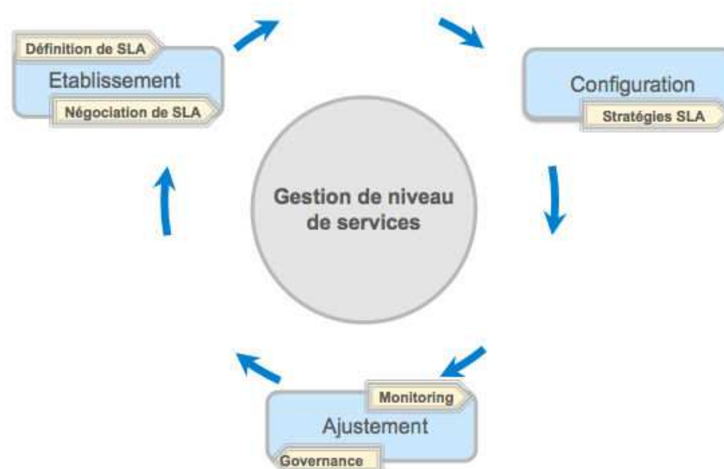


FIGURE 4.16 – Cycle de vie CSLA

i) Etablissement de SLA

Les services sont sélectionnés en fonction des préférences des consommateurs. Ensuite, le fournisseur et le consommateur définissent les termes du contrat via des paramètres prédéfinis ou suite à un processus de négociation. Cette phase sera détaillée dans la deuxième section de ce chapitre.

ii) Configuration du Cloud

Le fournisseur de service définit les stratégies (déploiement, provisionning) afin de gérer les contrats de niveaux de services. Son objectif est de maximiser son profit tout en respectant le SLA.

iii) Ajustement du Cloud

Les différents paramètres de QdS sont mesurés en continu par un système de monitoring. Le système est ajusté en fonction du contexte (courant et ou futur) afin de garantir le SLA. En cas de violation, les clauses de pénalité SLA correspondantes sont exécutées et le consommateur est informé par le service de la gouvernance.

Les deux dernières phases sont étudiées dans les chapitres 5 et 6.

4.1.6 Expérimentations

Le but des expérimentations est d'illustrer la gestion fine des violations via des études de cas. D'abord, nous illustrons les apports offerts par la dégradation de QdS (via les propriétés *fuzziness* et *confidence*). Ensuite, nous étudions la dégradation de fonctionnalités (les modes de fonctionnement). A la fin de la section, nous introduisons les éditeurs CSLA qui facilite la génération des SLAs conformément à la grammaire CSLA.

i) Étude de cas 1

Nous restons dans le cadre de notre exemple fil conducteur, où le service considéré est un service de type SaaS accessible en ligne par plusieurs clients. Nous distinguons deux niveaux de SLA : i) $\textbf{SLA}_S$ entre le client final et le fournisseur de SaaS, et ii) $\textbf{SLA}_R$ entre le fournisseur SaaS et le fournisseur IaaS.

Cette étude de cas est dédiée à l'illustration des apports de la dégradation de QdS pour affiner les stratégies du fournisseur SaaS par exemple. Nous développons notamment les stratégies de contrôle d'admission et d'ordonnancement.

a) Protocole d'expérimentation

Nous présentons dans ce qui suit l'environnement et le scénario d'évaluation.

Injecteur de charge : Apache JMeter [2] est un outil open source de test de performance et de charge. Il permet d'imiter les actions des utilisateurs réels en simulant la charge. Il est entièrement écrit en Java, ce qui lui permet d'être utilisé sur tout système

[2] http ://jmeter.apache.org/

d'exploitation supportant une machine virtuelle Java (JVM). Il supporte plusieurs protocoles en particulier HTTP.

Application synthétique : Les expérimentations réalisées ont été conduites avec des bancs d'essai synthétiques simulant un service d'informatique décisionnelle (BI - Business Intelligence). Ce service fournit 3 classes de services. Le Tableau 4.1 formalise les SLAs associés à chacune des classes. Nous énumérons deux lignes par classe (avec et sans propriétés CSLA). Le temps de réponse est évalué selon le concept *per-interval*. Il désigne le minimum des temps de réponse sur un intervalle de 10 minutes. Il est évalué sur une fenêtre de 30mn. Les prix et les pénalités sont déterminés suite à des statistiques sur des simulations.

TABLE 4.1 – SLA niveau SaaS (SLA_S)

service	métrique	oper.	valeur	fuzz.	% of fuzz.	conf.	prix	pénalité
gold	Rt	≤	0,4s	0	0	100	0,21 €/rqt	0,09€/rqt
				0,1	10	97	0,19€/rqt	0,08€/rqt
silver			0,6s	0	0	100	0,17 €/rqt	0,07€/rqt
				0,1	10	97	0,15€/rqt	0,06€/rqt
bronze			0,8s	0	0	100	0,14 €/rqt	0,05€/rqt
				0,1	10	97	0,12€/rqt	0,04€/rqt

Infrastructure : Les expérimentations présentées dans cette section ont été conduites par simulation des instances d'Amazon EC2. Le Tableau 4.2 illustre le SLA associé à ces ressources. La disponibilité est calculée en fonction du pourcentage de temps utilisable mensuel sur des intervalles de 5 minutes. La pénalité (crédit de service) est égale à un pourcentage de la facture du client.

TABLE 4.2 – SLA niveau IaaS (SLA_R)

service	métrique	oper.	valeur	fuzz.	% of fuzz.	conf.	prix	pénalité
Small	Av	≥	99,95%	0	0	100	0,046€/h	10% si 99,95 < Av ≤ 99% 30% si Av < 99%

Scénario d'évaluation L'expérimentation se déroule en deux phases : i) constater l'utilité des propriétés CSLA (*fuzziness* et *confidence*) sans modifier la stratégie (algorithme) du fournisseur SaaS et ii) intégrer des propriétés CSLA dans la stratégie du fournisseur.

Nous avons évalué `MinAvailCapacity` [WKGB12] avec les SLAs proposés dans le Tableau 4.1. Le principe de l'algorithme `MinAvailCapacity` est le suivant : affecter la requête acceptée à l'instance dont la capacité disponible est le minimum. Nous définissons la capacité d'une instance via le niveau de multiprogrammation maximal (MPL - MultiProgramming Level). Ce MPL limite le nombre des requêtes concurrentes. En appliquant l'algorithme `MinAvailCapacity`, une requête acceptée sera affectée à l'instance dont la différence entre son MPL et les requêtes en cours qu'elle traite est le minimum.

Ensuite, nous avons développé une extension de `MinAvailCapacity` pour prendre en compte les propriétés CSLA. En se basant sur la confiance (*confidence*), nous développons une heuristique qui calibre l'admission des requêtes. Au cas où l'admission

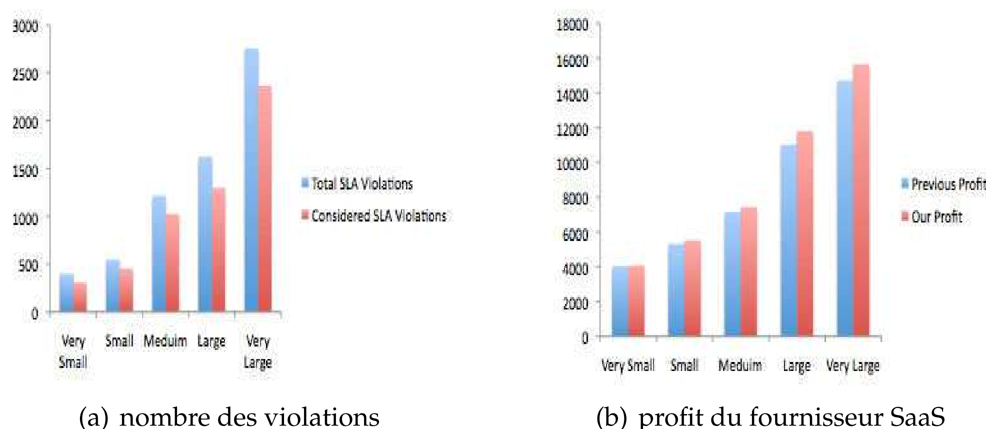
d'une requête nécessite l'ajout d'une instance, cette heuristique ralentit la décision d'admission en utilisant la *fuzziness* et vérifie la confiance. Si l'ajout d'une VM est encore indispensable, l'heuristique prend la décision en fonction de la confiance. Si cette dernière est vérifiée (temporairement) la requête est rejetée. Sinon la requête est admise et une VM est créée.

Nous avons testé le service avec 1000 clients. Nous avons fait varier le taux d'arrivée de 100 à 1000 requêtes par seconde. Nous classifions ces taux en 5 classes de charge : *very small* (de 100 à 300), *small* (de 300 à 500), *medium* (de 500 à 600), *large* (de 600 à 800) et *very large* (de 800 à 1000).

b) Résultats

Expérience 1 : Les Figures 4.17(a) et 4.17(b) illustrent la première phase d'expérimentation. Nous avons examiné l'algorithme `MinAvailCapacity` avec *confidence*=100% et *fuzziness* =0 puis *confidence*=97%, *fuzziness* =0,1 et pourcentage de *fuzziness* =10% (cf Tableau 4.1). La Figure 4.17(a) montre que la *fuzziness* et la *confidence* absorbent un nombre important des violations. Sans toucher les algorithmes définis par le fournisseur, nous constatons que notre modèle économique permet d'améliorer le profit du fournisseur (4.17(b)). Alors que la différence est très négligeable pour les taux d'arrivée very small et small, le gain devient important à partir du taux meduim.

Le profit de fournisseur reste fortement lié à la gestion de prix d'une requête en fonction des propriétés *confidence* et *fuzziness*. Un tel prix doit permettre au fournisseur de gagner sans importuner le client. Dans notre cas d'étude, le prix a été ajusté suite à des statistiques sur les simulations. Nous analysons ces statistiques afin de définir un rapport qualité-prix à la fois attrayant pour le client final et rentable pour le fournisseur SaaS. Des modèles économiques plus aboutis comme le Yield management permettraient d'affiner ces tarifs.

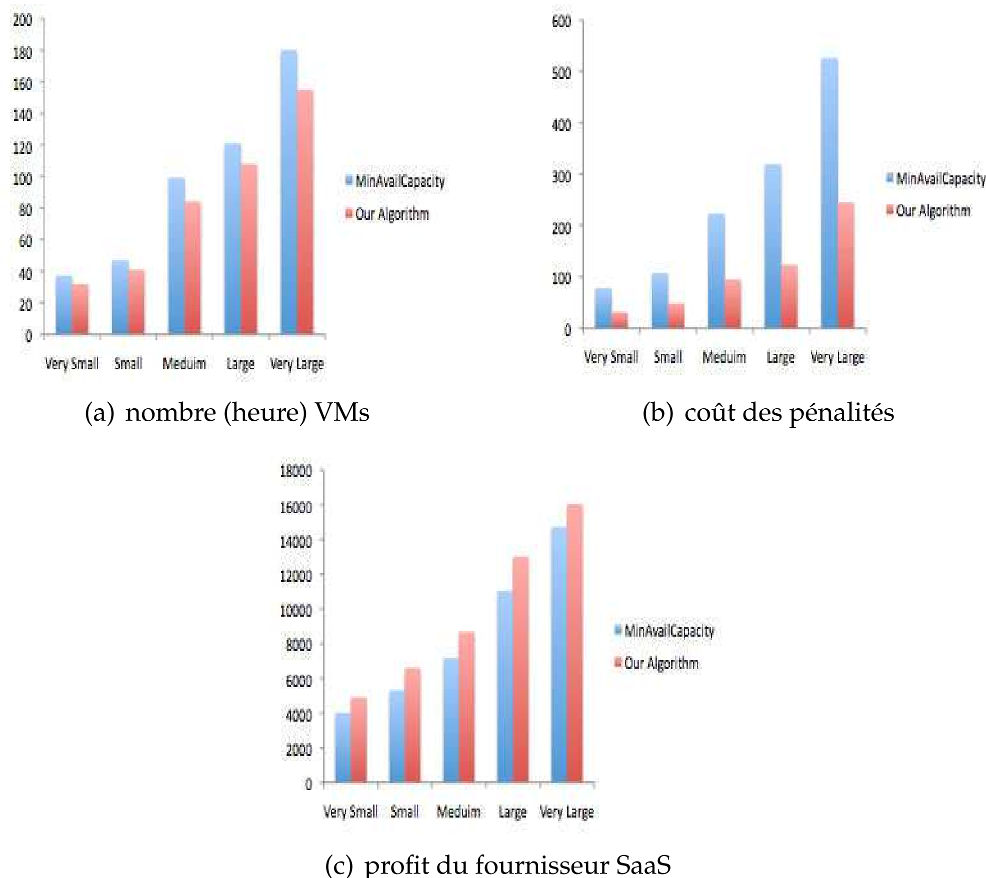


(a) nombre des violations (b) profit du fournisseur SaaS

FIGURE 4.17 – Expérience 1

Expérience 2 : Les Figures 4.18(a), 4.18(c) et 4.18(b) illustrent la comparaison d'algorithme `MinAvailCapacity` et l'extension avec *confidence*=97% ; *fuzziness* =0,1 pourcentage de *fuzziness*=10%. Notre extension permet de réduire le nombre des VMs initialisées (Figure 4.18(a)) comparée à l'algorithme `MinAvailCapacity` qui ignore la

confidence et la *fuzziness*. Ainsi notre proposition permet de réduire le coût de pénalité de moitié comme le montre la Figure 4.18(b). En réduisant les dépenses (VMs initialisées et pénalités), le profit de fournisseur est amélioré 4.18(c).



(a) nombre (heure) VMs

(b) coût des pénalités

(c) profit du fournisseur SaaS

FIGURE 4.18 – Expérience 2

ii) Étude de cas 2

L'objectif de l'étude de cas 2 est d'illustrer les apports de la dégradation de fonctionnalité. Nous considérons un système à deux niveaux XaaS (voir Figure 4.19), dans lequel, au niveau inférieur un IaaS fournit l'infrastructure physique comme un service au moyen de machines virtuelles, tandis qu'au niveau supérieur, un SaaS fournit un logiciel de publicité comme un service. Ainsi, le fournisseur SaaS est un client IaaS et les entreprises qui désirent faire des campagnes de publicité sont les clients du fournisseur SaaS. Enfin, les utilisateurs finaux sont des visiteurs du site qui consultent implicitement les annonces pendant leur navigation sur Internet.

Le SaaS fournit le service de publicité en deux modes : normal (vidéo) et dégradé (image statique). Les entreprises sont facturées via un coût par mille impressions (CPM), soit pour mille pages vues.

a) Protocole d'expérimentation

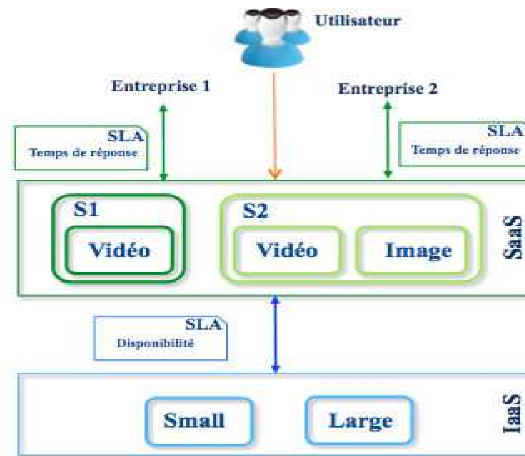


FIGURE 4.19 – Cas d'étude 2

L'évaluation est basée sur le framework décrit dans [dOL12b]. Le fournisseur IaaS peut envoyer des notifications à l'application (SaaS) en cas d'une pénurie de ressource (e.g. énergie). Une fois notifée, l'application passe en mode dégradé pour répondre aux contraintes de disponibilité des ressources IaaS.

Nous présentons dans ce qui suit l'environnement et le scénario de l'expérimentation.

Injecteur de charge : Nous utilisons également Apache JMeter pour imiter des actions des utilisateurs réels en simulant la charge.

Application synthétique : Les expérimentations réalisées ont été conduites avec des bancs d'essai synthétiques représentant un service de publicité. Nous utilisons les variantes de service de publicité : S1 et S2. Le Tableau 4.3 formalise le SLA associé à ce service. Le temps de réponse est calculé comme la moyenne de toutes les demandes sur des intervalles d'une minute.

TABLE 4.3 – SLA niveau SaaS (SLA_S)

service	métrique	oper.	valeur	fuzz.	% of fuzz.	conf.	mode	prix	pénalité
S1	Rt	≤	500ms	300ms	20	90%	vidéo(100%)	0,30$-cpm	0,10$/rqt
S2							vidéo(80%) image(20%)	0,20$-cpm	

Infrastructure : Les expérimentations présentées dans cette section ont été conduites sur des machines virtuelles de la plate-forme Grid'5000[3] qui est une grille de calcul dédiée à la recherche. Le Tableau 4.4 illustre le SLA associé à ces ressources. La plate-forme est à usage gratuit mais nous avons fixé des tarifs liés à la consommation de l'empreinte énergétique remontée par des Wattmètres. La disponibilité est calculée en fonction du pourcentage de temps où les ressources sont accessibles sur des intervalles de 6 minutes.

Scénario d'évaluation La durée des expériences a été fixée à une heure, au cours de laquelle la charge de travail de ces deux variantes SaaS augmente de 0 à 140 requêtes

[3]https ://www.grid5000.fr/

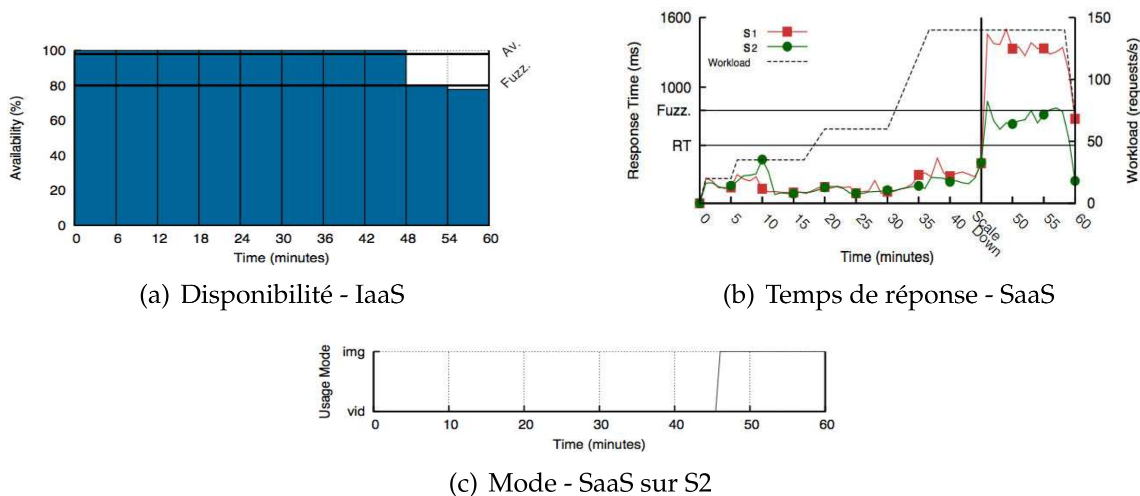
TABLE 4.4 – SLA niveau IaaS (SLA_R)

service	métrique	oper.	valeur	fuzz.	% of fuzz.	conf.	prix	pénalité
Small Large	Av	≥	98%	18%	10	100%	0,06$ 0,12$	0,05$/core

par seconde. Cela signifie que les deux variantes nécessitent la même quantité de ressources jusqu'à atteindre le pic de charge. Une pénurie d'énergie est prévue pour être déclenchée à 45 minutes après le début du scénario. L'objectif est d'observer et d'illustrer l'avantage de la dégradation de fonctionnalité.

b) Résultats

Après l'événement de pénurie, la disponibilité diminue à 80% sur le premier intervalle d'observation puis à 78% sur l'intervalle suivant (voir Figure 4.20(a)). Cette diminution s'explique par des VMs rendues indisponibles en raison de la pénurie. La courbe 4.20(b) illustre la conséquence directe de cette indisponibilité (niveau IaaS) sur le temps de réponse niveau SaaS.



(a) Disponibilité - IaaS

(b) Temps de réponse - SaaS

(c) Mode - SaaS sur S2

FIGURE 4.20 – Les performances

Le service S2 propose des performances (temps de réponse) meilleures que le service S1 comme il bascule en mode dégradé (image) pour absorber la même charge de travail tout en évitant les violations de SLA (voir Figure 4.20(c)).

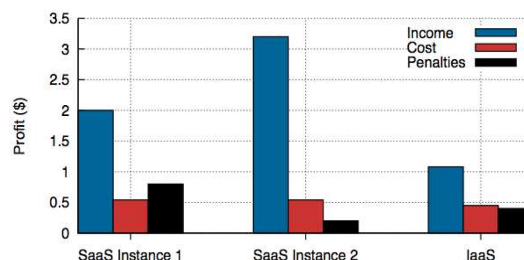


FIGURE 4.21 – Les profits

La Figure 4.21 illustre les revenus, les coûts et les pénalités liés aux deux variantes SaaS ainsi que le IaaS. Sans surprise, les deux instances SaaS ont le même coût, car la même quantité de ressources est allouée. Cependant, le revenu de la variante S1 est inférieur à celui de la variante S2, car la variante S1 traite moins de demandes dans le même laps de temps. Il est également important d'observer que les sanctions sont plus élevées pour la variante S1 en raison des nombreuses violations de SLA après la mise à l'échelle forcée, qui se produit à la fin des expériences.

Pour des raisons de simplicité, les coûts IaaS sont liés à la quantité d'énergie (dues à la consommation d'énergie de deux variantes SaaS) multipliée par les frais actuels de l'énergie. Enfin, les sanctions IaaS sont payées par les IaaS au fournisseur SaaS en raison de violations de la disponibilité causée par la pénurie d'énergie.

Pour conclure, cette étude de cas montre les apports de CSLA et en particulier la dégradation de fonctionnalité via la notion de modes de fonctionnement. La variante S2 a mieux géré les violations même face à une situation extrême telle que la pénurie d'énergie.

iii) Éditeurs CSLA

Afin de simplifier la génération et l'édition des SLAs, nous avons développé deux éditeurs : un plugin eclipse et un éditeur Web.

a) Plugin eclipse

Nous avons développé un plugin Eclipse pour la génération des contrats. Ce plugin est basé sur i) Obeo Designer : une solution propriétaire dédiée à la conception graphique et ii) Acceleo : une solution open source de génération de code basée sur une vision pragmatique du MDA (Model Driven Architecture).

Notre plugin simplifie l'édition d'un CSLA via une interface graphique (voir Figure 4.22). Cette interface génère des modèles CSLA. Nous utilisons Acceleo pour transformer ces modèles vers des contrats en format XML.

FIGURE 4.22 – Interface graphique

b) Editeur Web

L'idée qui a motivé le développement d'un éditeur Web pour CSLA est de fournir cet éditeur comme un service SaaS. L'éditeur est développé avec MySql, PHP, HTML5. Il est divisé en deux parties frontend et backend. Le backend est destiné au fournisseur d'un service Cloud afin de paramétrer ses contrats. En plus, cette partie permet de personnaliser l'affichage. Une fois que le paramétrage est terminé, l'éditeur fournit une URL de frontend. Le frontend représente l'ensemble des interfaces visualisées par le client de service Cloud. Il permet la sélection d'un service et un template d'un contrat. Une fois la phase de sélection terminée, le client doit valider son choix pour l'établissement du contrat. L'éditeur permet de fournir le contrat en format lisible pour le client (format PDF). Ce prototype d'éditeur Web a partiellement été développé dans le cadre du projet ANR MyCloud [MyC13]. Il est actuellement en cours de finalisation.

Discussion

Afin de remédier aux limitations présentées dans l'état de l'art à savoir principalement : la définition de service XaaS et la gestion fine de violations, nous avons proposé CSLA : *Cloud Service Level Agreement*. CSLA favorise la formalisation des SLA dans n'importe quelle langage (XML, Java,...) pour n'importe quel service XaaS (SaaS, PaaS, IaaS). Il permet de gérer des garanties multicritères avec et sans priorité pour refléter les préférences du client. CSLA propose un modèle économique novateur pour gérer plus finement les violations : la dégradation fonctionnelle (modes de service), la dégradation de QdS (*fuzziness* et *confidence*) et des modèles avancés de pénalité. De plus, CSLA suit l'architecture de référence de l'informatique en nuage de NIST et supporte l'interface OCCI.

Les expérimentations illustrent les apports proposés par CSLA en tant que langage et modèle économique à la fois. Le compromis –entre le profit de fournisseur SaaS et la satisfaction de ses clients– est fortement lié aux paramètres CSLA. Le calibrage du template CSLA doit permettre au fournisseur de gagner sans importuner le client. L'objectif est d'avoir un bon rapport qualité/fonctionnalité-prix. Le prix doit être ajusté en fonction des modes (dégradation de fonctionnalité) et les pénalités doivent être harmonisées en fonction des propriétés *confidence* et *fuzziness* (dégradation de QoS). Les modèles économiques tels que le Yield management montrent une voie pour améliorer cette perspective de travail.

4.2 Dépendances SLA

Les SLAs peuvent être exprimés entre différentes couches dans la "pile" du Cloud, contractualisant de fait des dépendances entre les différentes couches XaaS. Afin de proposer un SLA à un niveau n, il faut que les niveaux inférieurs invoqués proposent déjà des SLAs.

Dans cette section, nous étudions le calibrage d'un template CSLA niveau SaaS à destination d'un end-user en se basant sur le SLA signé avec son fournisseur IaaS. Nous clarifions, d'abord, notre vision du calibrage SLA. Ensuite, nous expliquons la modélisation de notre solution. A la fin de la section, nous proposons des expérimentations pour évaluer notre solution.

4.2.1 Calibrage de CSLA template, la méthode

Le calibrage de CSLA template niveau SaaS dépend des ressources allouées au fournisseur IaaS et du SLA_R associé. L'objectif du fournisseur SaaS est double : d'une part, la génération des paramètres d'un template CSLA pour s'assurer que les garanties de QdS proposées sont réalistes et réalisables par rapport à SLA_R, d'autre part, la sélection de type/nombre de ressources optimal (planification) dans le cadre de SLA_R pour le template généré. En effet, son objectif est de proposer un SLA_S rentable.

L'originalité de notre approche est de s'appuyer sur les préférences du client pour calibrer le SLA_S. Nous distinguons deux catégories de calibrage : le calibrage par défaut et le calibrage sur mesure. Le calibrage par défaut est un calibrage hors ligne où le fournisseur estime les préférences de ses clients pour proposer des SLAs. Alors que le calibrage sur mesure est un calibrage en ligne. Il est très similaire à un processus de négociation où le fournisseur récupère les préférences de son client et fournit un SLA correspondant.

Dans ce qui suit, nous commençons par clarifier la méthode de calibrage. Ensuite, nous expliquons la modélisation de dépendances SLA. Enfin, nous illustrons quelques préférences du client.

i) Méthode de calibrage, vue d'ensemble

L'objectif pour le fournisseur SaaS est de trouver le meilleur chemin à travers les couches XaaS pour proposer un SLA_S rentable.

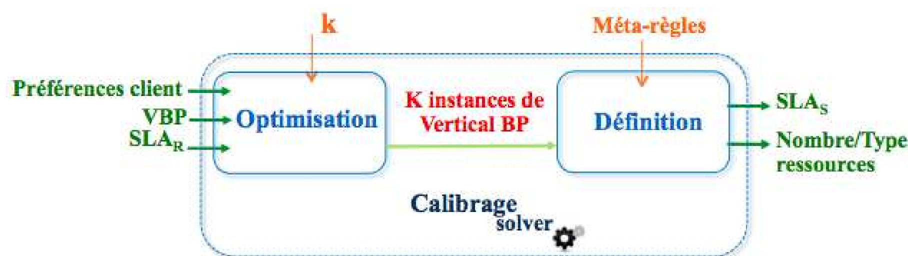


FIGURE 4.23 – La méthode de calibrage

La Figure 4.23 schématise la méthode de calibrage qui contient deux étapes : i) l'optimisation du processus métier vertical et ii) la définition du SLA_S. En effet, nous introduisons la notion de *Vertical Business Process - VBP* : un processus métier vertical (cross-layer) pour modéliser les dépendances entres les niveaux XaaS. Ainsi, notre solution permet, alors, à partir d'un *VBP* (dépendances) et les préférences du client d'instancier les k premières instances de *VBP* qui satisfassent les préférences de client.

L'instanciation d'un *VBP* revient à la selection/composition du nombre/type optimal des services/ressources concrètes dans le cadre de SLA_R. Cette sélection/composition est modélisée comme un problème d'optimisation multi-critères. Le paramètre k donne différentes possibilités de calibrage de SLA_S en particulier les paramètres *confidence*, *fuzziness* et le modèle de pénalité. La phase de définition consiste à définir un SLA_S et le nombre/type de ressources nécessaires pour le garantir. Cette phase est dirigée par des méta-règles pré-définies par le fournisseur SaaS. Ces méta-règles reflètent la

politique commerciale (politique de prix) du fournisseur en fonction de la concurrence, le positionnement de prix et la demande. Par exemple : prix élevé et *confidence*= 100%.

ii) Processus métier vertical

Un processus métier vertical (voir Figure 4.24) définit un ensemble de services abstraits qui interagissent à travers des couches afin de fournir un service (exemple : analyse des données). Chaque service abstrait encapsule certaines fonctions (informatique décisionnelle, calcul, stockage, etc.). Un service/une ressource concret(ète) met en œuvre un service abstrait. Par exemple côté IaaS, le service abstrait de calcul chez Amazon EC2 peut avoir plusieurs ressources concrètes comme *small*, *medium*, *large* et *extra-large*.

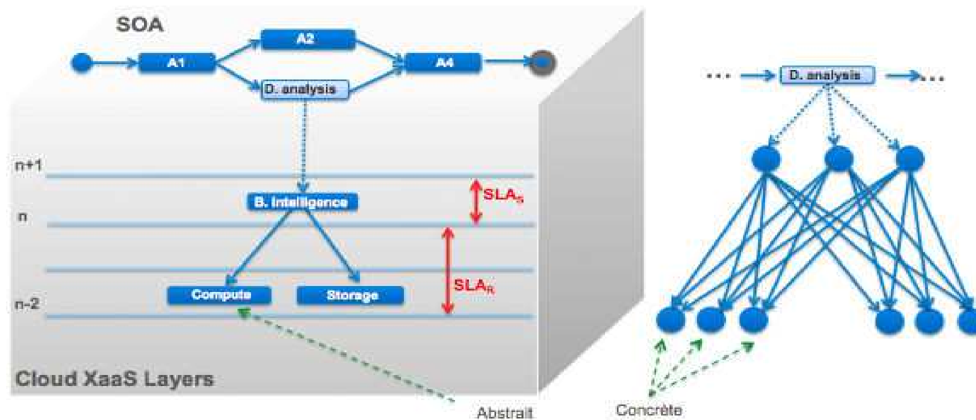


FIGURE 4.24 – Processus métier vertical

Chaque service/ressource concret(ète) est étalonné(e), puis qualifié(e) en amont par une campagne de tests pour définir ses caractéristiques de QdS. Le XML suivant présente un exemple de ressources concrètes (instances de calcul de Amazon EC2) qui ont été étalonnées suite à une campagne de benchmarking en fonction de la nature/type de requêtes en entrée.

```
1  <Abstract type="Compute">
2    <Qosfor dateSize="1 teraoctet (To)" dataComplexity ="simple data"
3            quereyClass= "simple query">
4      <Concrete name="EC2" type="On−Demand" family="Large Instance"
5              operatingSystems="Linux" region= "EU (Ireland)">
6        <qos metric="response time" unit="second"> 5 </qos>
7        <qos metric="availability" unit="%"> 99,95 </qos>
8        <qos metric="financial cost" unit="euro"> 0.34 </qos>
9      </Concrete>
10     <Concrete name="EC2" type="On−Demand" family="Micro Instances"
11             operatingSystems="Linux" region= "EU (Ireland)">
12       <qos metric="response time" unit="second"> 8 </qos>
13       <qos metric="availability" unit="%"> 99,95 </qos>
14       <qos metric="financial cost" unit="euro"> 0.02 </qos>
15     </Concrete>
16   </Qosfor>
17 </Abstract>
```

Il est également possible de s'appuyer sur des modèles statistiques pour calculer les différents critères de QdS pour chaque service/ressource concret. Une première réflexion est la fonction 4.11 qui calcule le temps de réponse en fonction du type de requête (simple ou complexe relativement au nombre de jointure par exemple), le volume et la complexité de données.

$$QoS_{(QueryClass,DataSize,DataComplexity)}(EC2Instance) = ResponseTime \qquad (4.11)$$

Cette alternative est à explorer dans les perspectives de cette thèse.

iii) Préférences du client

Chaque client a un profil qui guide ses préférences. Nous illustrons deux exemples de préférences du client du service SaaS présentés de façon informelle :

Exemple 1 : Le SLA_S peut être exclusivement dédié à la performance. Dans ce cas, étant donnée une requête client, l'objectif est de garantir au client un temps de réponse, sans limitation de coût financier. Par exemple : garantir un temps de réponse inférieur à 3 secondes quelque soit le coût financier. Le SLA_S peut être également multi-critères (performance + disponibilité). Par exemple garantir un temps de réponse inférieur à 3 secondes et un taux de disponibilité de 99% quelque soit le coût financier.

Exemple 2 : le SLA_S peut être exprimé en termes financiers. Dans ce cas, l'objectif est de garantir au client un temps de réponse, pour un coût financier fixé. Par exemple : pour un coût financier maximum de 0,10 €/requête, garantir un temps de réponse inférieur à 1 minute et un taux de disponibilité de 99%.

Notre but est de traduire/formaliser chaque préférence par un objectif (SLO). Concrètement, c'est un ensemble de quatre paramètres : les finalités (maximiser ou minimiser), les paramètres QdS (métrique), les valeurs QdS (seuils) et les poids (priorités). Au cas où le client exprime clairement des priorités, nous gardons l'ordre attendu.

4.2.2 Calibrage de CSLA template, la pratique

Dans cette section, nous détaillons la méthode de calibrage. Nous choisissons de modéliser le problème d'optimisation par la programmation par contraintes (PPC) [RBW06]. D'abord, nous présentons le problème d'optimisation de processus métier vertical avec les notions de base de PPC. Ensuite, nous développons notre fonction objective puis l'algorithme de résolution. Enfin, nous expliquons la deuxième étape de la méthode de calibrage à savoir la définition du SLA_S.

i) Programmation par contraintes

La PPC est une technique mathématique qui permet la résolution de problèmes combinatoires modélisés comme un problème de satisfaction de contraintes (CSP : Constraints Satisfaction Problem). Un CSP est défini formellement par un triplet : les variables X, les domaines D et les contraintes C. La résolution consiste alors à trouver un tuple de valeurs tel que toutes les contraintes soient satisfaites. Elle est basée sur la propagation de contraintes ou le renforcement de la consistance ainsi que le filtrage. Pour les problèmes d'optimisation, il suffit de définir une fonction objective $f : D(X) \longrightarrow \mathbb{R}$. Une solution optimale est alors un triplet de solution du CSP qui minimise (ou maximise) la fonction f.

L'idée est de modéliser et résoudre un problème de sélection/composition en explorant les contraintes qui peuvent caractériser complètement le problème. Nous résolvons

le problème de sélection/composition où les variables sont le type et nombre des services/ressources concrets(ètes), les domaines sont bornés par le fournisseur de IaaS et les contraintes sont les objectifs (SLOs). Afin de calculer les k premières solutions optimales, nous introduisons une fonction objective.

ii) Fonction objective

La fonction objective (équation 4.12) permet de trouver une instance optimale de *processus métier vertical*. Par exemple, dans notre cas du fournisseur SaaS ce sont le nombre minimal de ressources IaaS.

$$f = min \sum_{i=0}^{n-1} \alpha_i w_{Ni} q_{N_i} \tag{4.12}$$

Où n est le nombre de critères (SLO), α_i est l'objectif de critère i (maximiser ou minimiser), w_{Ni} est le poids de critère i et q_{N_i} est la valeur globale normalisée de critère i. Afin de calculer la valeur globale pour chaque critère, nous utilisons les fonctions présentées dans le Tableau 4.5.

TABLE 4.5 – Aggrégation des QoS

	Unité	Séquentiel	Parallèle
disponibilité : Av	poucentage :%	$\prod_{i=1}^{n} Av_i$	$\prod_{i=1}^{n} Av_i$
Response time : Rt	Secondes : s	$\sum_{i=1}^{n} Rt_i$	$max(Rt_i)$
Cout financier : Ct	Euros : €	$\sum_{i=1}^{n} Ct_i$	$\sum_{i=1}^{n} Ct_i$

La valeur de α_i est définie par l'équation 4.13.

$$\alpha_i = \begin{cases} +1 & \text{si } minimiser(q_{N_i}) \\ -1 & \text{si } maximiser(q_{N_i}) \end{cases} \tag{4.13}$$

La w_{Ni} est représentée par l'équation de poids 4.14 :

$$w_{Ni} = \begin{cases} \frac{w_{n-i}}{w_T} & \text{si } w_T \neq 0 \\ 1 & \text{sinon} \end{cases} \tag{4.14}$$

Où $w_T = \sum_{i=1}^{n} w_i$ et w_i est le poids de critère q_i .

Lors du traitement de plusieurs critères de décision, il est nécessaire de normaliser les critères en raison de la stabilité numérique. La normalisation des variables nécessite de prendre des valeurs couvrant un intervalle spécifique et de les représenter dans un autre intervalle. La méthode standard consiste à normaliser les variables à [0,1]. La méthode la plus simple pour normaliser les valeurs est la mise à l'échelle linéaire de transformation (équations 4.15 et 4.16) pour le positif et le négatif.

$$q_{N_i} = \begin{cases} \frac{q_i - q_m}{q_M - q_m} & \text{si } q_M - q_m \neq 0 \\ 1 & \text{sinon} \end{cases} \tag{4.15}$$

$$q_{N_i} = \begin{cases} \frac{q_M - q_i}{q_M - q_m} & \text{si } q_M - q_m \neq 0 \\ 1 & \text{sinon} \end{cases} \tag{4.16}$$

Où q_i est la valeur globale de critère i, q_m = min (q_i) et q_M = max (q_i).

iii) Algorithme

Dans cette partie, nous présentons en détails comment nous avons modélisé et résolu le problème de sélection/composition en utilisant un CSP (voir Algorithme 2). Les paramètres d'entrée sont les critères de QoS, les valeurs, les poids, *k* (nombre de solutions). L'algorithme fournit les *k* premières solutions.

La première étape consiste à créer un modèle (ligne 1). Les variables et les contraintes seront associés au modèle. Les valeurs des variables sont définies via les domaines (lignes 2-3). Les contraintes (SLOs) indique les relations entre les variables et les valeurs de QoS à respecter (ligne 4). Nous limitons le nombre des solutions avec *k* (ligne 5). La dernière étape revient à définir la stratégie de recherche puis résoudre la fonction objective (ligne 6 - 7).

Algorithme 2: cloudServiceSelection

Input : *businessprocess BP, qos Q, values V, weights W, k.*
Output : *Set of solutions.*

1 *create the constraint – programming model M* ;
2 *create variables* $X_i(BP, Q)$;
3 *for each* X_i *Define its domain* $D(X_i)$;
4 *create and post constraints* $C_j(V, W)$;
5 *configure the solution pool capacity*(k);
6 *create the solver S*;
7 *solve M*;
8 **return** *solutions*

iv) Définition d'un SLA

Une fois les *k* premières solutions calculées, le processus de calibrage définit un ou plusieurs SLA en fonction des solutions fournies. Nous étudions en particulier la définition des objectifs (SLOs). Les seuils des objectifs sont définis en fonction de la première solution alors que les propriétés (*fuzziness* et *confidence*), les prix et les pénalités sont paramétrés en fonction de différence entres les *k* premières solutions. Les seuils des objectifs ainsi que la valeur de *fuzziness* sont paramétrés automatiquement alors que le reste des paramètres est calculé d'une manière semi-automatique en se basant sur des méta-règles.

Calibrage par défaut : Dans ce cas, nous attribuons la valeur 100% à la *confidence* et la valeur 0 à la *fuzziness*. Ces paramètres peuvent être facilement raffiner par des techniques d'apprentissage.

Calibrage sur mesure : Dans ce cas, la *confidence* et le pourcentage de *fuzziness* sont calculées d'une manière semi-automatique. Le fournisseur SaaS peut utiliser soit des paramètres prédéfinis ou des politiques.

Soit l'exemple illustratif suivant : la préférence du client SaaS est de garantir un temps de réponse inférieur à 3 secondes pour un coût financier maximum de 1€/requête. Le paramètre *k* est égal à 2. Soit les deux premières solutions (temps de réponse=2,5s, coût financier=0,9 €/requête) et (temps de réponse=2,8s, coût financier=0,7 €/requête). Le SLA proposé est le suivant : un temps de réponse inférieur à 2,5 secondes

avec une *fuzziness* égale à 0,3 seconde. Le pourcentage de *fuzziness*, la *confidence*, le prix et les pénalités sont calculés selon les méta-règles définies par le fournisseur SaaS ainsi que la différence entre les k premières solutions.

4.2.3 Expérimentations

Dans cette section, nous présentons les expérimentations. En premier lieu, nous détaillons notre prototype de calibrage. Ensuite, nous développons le protocole d'expérimentation et les résultats. A la fin de la section, nous présentons un bilan.

i) Prototype

Nous avons développé un prototype pour évaluer notre solution. Nous avons utilisé Choco[4] comme solveur de contraintes. Ce solveur met à disposition une bibliothèque Java permettant la résolution de problèmes de satisfactions de contraintes. Les critères de QdS pour chaque service/ressource concret(ète) sont présentés dans un fichier XML (fourni par le fournisseur SaaS suite à des tests réalisés). Nous avons mis en place un parser XML en utilisant l'API SAX[5] (open-source) pour parser le fichier XML afin d'extraire les valeurs de QdS.

ii) Protocole d'expérimentation

Nous présentons dans ce qui suit l'environnement et le scénario de l'expérimentation.

Application synthétique : Les expérimentations réalisées ont été conduites avec des bancs d'essai synthétiques représentant un service d'informatique décisionnelle (BI - Business Intelligence). Nous distinguons entre 4 classes de ce service à savoir : gold, silver, bronze et platinum. Ces classes de service BI sont hébergées sur des ressources de calcul et stockage de Amazon. Plusieurs instances de calcul sont fournies selon le type, la zone ou région, le système d'exploitation, la famille. Également le service de stockage est fourni par plusieurs zones et il est facturé selon des paliers de volume. Les ressources concrètes de calcul sont résumées dans le Tableau 4.6 sinon pour le service de stockage, nous utilisons les mêmes zones ainsi que trois paliers de volume. Le SLA_R associé est illustré dans le Tableau 4.7 [6] [7] [8] [9].

TABLE 4.6 – service-ressource

service	type	zone	sys. d'expoitation	famille
calcul	à la demande	USA Est USA ouest UE	linux windows	small medium large extra-large

[4]http ://www.emn.fr/z-info/choco-solver/
[5]http ://www.saxproject.org/
[6]http ://aws.amazon.com/fr/ec2-sla/
[7]http ://aws.amazon.com/fr/ec2/pricing/
[8]http ://aws.amazon.com/fr/s3-sla/
[9]http ://aws.amazon.com/fr/s3/pricing/

TABLE 4.7 – SLA_R

service	métrique	oper.	valeur	fuzz.	% of fuzz.	conf.	prix	pénalité
calcul stockage	Av	≥	99,95%	0	0	100	cf. EC2 cf. S3	cf. EC2 cf. S3

Infrastructure : Les expérimentations sont réalisées au travers de simulations sur un Mac OS X 10.6.7, 2.7 GHz Intel Core i7, 4Go de mémoire DDR3et 3Mo de cache.

Scénario d'évaluation : Nous évaluons chaque service/ressource en amont pour définir les critères de QdS (le coût financier, le temps de réponse et la disponibilité). Un fichier XML de QdS est fourni suite à cette campagne de benchmarking.

Le client SaaS présente ses préférences pour analyser ses données via le service BI. Nous utilisons deux préférences :

- préférence 1 : garantir un temps de réponse inférieur à 20 secondes et un taux de disponibilité de 99%, quelque soit le coût financier.
- préférence 2 : pour un coût financier maximum de 0,10 €/requête, garantir un temps de réponse inférieur à 30 secondes et un taux de disponibilité de 99%.

Nous commençons par l'initialisation des paramètres de la méthode de calibrage à savoir principalement les dépendances et les préférences du client (cf. Section 4.2.1). Nous attribuons la valeur 2 à k. Les méta-règles fournies fixent les valeurs de *confidence*=95% et *fuzziness-percentage*=10%. Ensuite, nous utilisons notre prototype pour trouver un SLA_S pour le client de service BI.

iii) Résultats

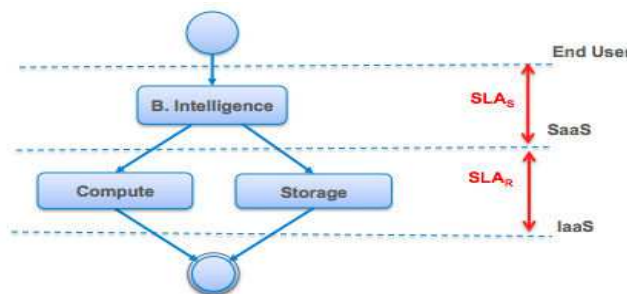


FIGURE 4.25 – Processus métier vertical utilisé

Nous modélisons les dépendances via un processus métier vertical. Le résultat est illustré dans la Figure 4.25. Ainsi, nous formulons les préférences via des objectifs (SLOs). Chaque préférence est traduite par (finalité, métrique, seuil, poids). La traduction des préférences (1 et 2) est illustrée dans le Tableau 4.8. Le poids (priorité) est généré par le fournisseur SaaS s'il n'a pas une priorité explicite dans les préférences.

La première étape de la méthode de calibrage est la recherche de k premières solutions qui satisfassent les préférences du client SaaS et optimisent le coût des ressources en terme de nombre et type. Le Tableau 4.9 illustre les deux premières solutions de la

TABLE 4.8 – Traduction des préférences

préf.	finalité	métrique	seuil	poids
1	maximiser	Av	99%	2
	minimiser	Rt	20s	1
2	minimiser	Cf	0,10 €/requête	1
	maximiser	Av	99%	2
	minimiser	Rt	30s	3

préférence 1.

TABLE 4.9 – Solutions pour préférence 1

préf.	solu.	bi	compute	storage	Rt	Av
1	1	gold	large, lunix, UE	1, UE	18,7s	99,95%
	2	gold	medium, lunix, UE	1, UE	20s	99,95%

Dans ce cas d'étude, nous avons réussi à trouver les k premières solutions qui reflètent exactement les préférences attendues du client SaaS. La deuxième solution de la préférence 1 permet de raffiner la propriété *fuzziness*. Le reste des paramètres (propriété *confidence*, modèle de pénalité) est défini en fonction des méta-règles et la différence entre les k premières solutions. Le SLA_S fourni proposé au client est le suivant (nous illustrons que les objectifs) :

```
1  <csla:terms>
2       <csla:term id="T1" operator="and">
3         <csla:item id="responseTimeTerm"/>
4         <csla:item id="availabilityTerm"/>
5       </csla:term>
6       <csla:objective id="responseTimeTerm" priority="1" actor="provider">
7         <csla:precondition policy="Required">
8         <csla:expression metric="Rt" comparator="lt"
9                          threshold="20" unit="second"
10                         monitoring="Mon-1" schedule="Sch-1"
11                         Confidence="95" fuzziness-value="1,3"
12                         fuzziness-percentage="10"/>
13      </csla:objective>
14      <csla:objective id="availabilityTerm" priority="2" actor="provider">
15        <csla:expression metric="Av" comparator="gt"
16                         threshold="99,95" unit="\%"
17                         monitoring="Mon-2"
18                         Confidence="95" fuzziness-value="0,95"
19                         fuzziness-percentage="10"/>
20     </csla:terms>
```

Le raisonnement est le même pour la préférence 2.

Les caractéristiques principales de notre approche sont la flexibilité et la scalabilité.

Flexibilité : Notre approche est flexible de point de vue fournisseur de SaaS ainsi que pour son client. Basé sur la CP, notre prototype gère les différentes préférences du client qui sont traditionnellement non considérées. Il traite différents types de contraintes et les combinaisons entre eux. Il est possible de composer en ajoutant au besoin des conditions que les solutions doivent satisfaire.

Scalabilité : La Figure 4.26 illustre la scalabilité de notre approche. Face à 50 services/ressources concrets(ètes) et 3 critères (temps de réponse, disponibilité, coût finan-

cier), le temps de calcul de la solution optimale ne dépasse pas 210 ms. Un temps qui est tout à fait acceptable lors du calibrage sur mesure. En effet, le filtrage et la propagation des contraintes permettent d'améliorer les capacités de résolution de CSP en complément de la construction d'un arbre de recherche.

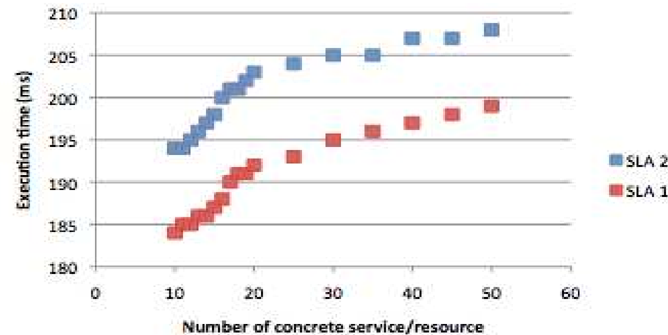


FIGURE 4.26 – Scalabilité

Discussion

Nous avons proposé une solution pour calibrer le CSLA template et aiguiller les ressources en terme de type et nombre. Cette approche est dirigée par les préférences du client du fournisseur, un aspect qui n'est pas considéré précédemment. Nous avons mis l'accent sur le fournisseur SaaS cependant notre approche est générique. Avec un simple "shift", notre modèle peut être utilisé par un fournisseur IaaS pour calibrer son SLA avec ses clients (exemple fournisseur SaaS) et ses fournisseurs (exemple fournisseur d'énergie). Basée sur la programmation par contrainte, notre approche calcule à la volée (sur mesure) une solution dans un temps raisonnable. Elle permet de prendre en compte les paramètres QdS dynamiques de manière souple pour composer le meilleur processus métier vertical.

4.3 Synthèse

Dans ce chapitre, nous avons détaillé la spécification du langage CSLA tout en essayant de fournir des éléments de réponse aux limitations soulevées au niveau des travaux de l'état de l'art. Notre contribution n'est pas qu'un moyen de définition du SLA mais aussi un modèle économique avancé pour la gestion des contrats de niveaux de service. Nous avons introduit plusieurs concepts à savoir principalement : la dégradation de fonctionnalité (mode) et la dégradation de QdS (*fuzziness* et *confidence*). Ainsi, nous avons proposé un modèle de pénalité à grain fin.

Dans la deuxième partie de ce chapitre, nous avons illustré notre solution de calibrage de template CSLA. Nous avons modélisé le problème de dépendances SLA comme un problème d'optimisation multicritères. Notre prototype est basé sur la programmation par contraintes. L'originalité de notre approche est l'appui sur les préférences des clients pour calibrer le SLA_S, l'énumération des k premières solutions, la flexibilité et la généricité. Notre approche est limitée à la phase de conception (design-time). Nous traitons la phase de l'exécution (run-time) dans les chapitres suivants.

5

HybridScale : dimensionnement automatique dirigé par SLA

Nous avons précédemment mis l'accent sur les limitations des approches existantes de gestion de capacité des ressources qui sont principalement : la non-considération du modèle économique en particulier la facturation et la non-prise en compte du temps d'initialisation des ressources. Afin de remédier à ces limitations, nous proposons *HybridScale* : une approche de dimensionnement automatique dirigée par *CSLA*.

Ce chapitre commence avec une description de vue d'ensemble de *HybridScale* en clarifiant ses apports. Ensuite, nous introduisons *BestPolicies* qui proposent des politiques de gestion de l'élasticité du Cloud. Nous présentons, également, *Forecasting*, une technique de prédiction de la demande. Enfin, nous développons *RightCapacity*, une méthode de planification de capacité de ressources dirigée par SLA.

Sommaire

5.1 *HybridScale*, vue d'ensemble 106

5.1.1 Nouveautés d'*HybridScale* 106

5.1.2 Modèle conceptuel 106

5.2 *BestPolicies* : politiques de gestion d'élasticité 107

5.2.1 Politiques des prix 107

5.2.2 Politiques de réaction 108

5.2.3 Politiques de dimensionnement 109

5.3 LoadForecasting : prédiction de charge de travail 111

5.3.1 Modélisation d'une série chronologique 111

5.3.2 Forecaster, la technique 112

5.4 *RightCapacity* : planification de capacité dirigée par SLA 114

5.4.1 Modèle Conceptuel 114

5.4.2 Modèle analytique 114

5.4.3 Planification des ressources, la méthode 120

5.4.4 Planification de l'architecture logicielle, la méthode 126
5.5 Synthèse 127

5.1 *HybridScale*, vue d'ensemble

Cette section met en évidence l'approche *HybridScale* et ses apports. Nous commençons par clarifier les nouveautés d'*HybridScale* par rapport à l'état de l'art. Ensuite, nous illustrons une vue d'ensemble de cette solution.

5.1.1 Nouveautés d'*HybridScale*

La contribution principale de cette thèse est l'approche *HybridScale* : une solution avancée pour gérer l'élasticité du Cloud. Cette approche est dirigée par *CSLA* et englobe d'autres contributions à savoir *RightCapacity* et *BestPolicies*. L'originalité de *HybridScale* est le triple hybride : ajustement réactif-proactif, dimensionnement vertical-horizontal et au niveau application-infrastructure. Elle est implémentée via une boucle de contrôle autonomique MAPE-K [Hor01].

Notre solution est modélisée par le quadruplet (Quand, Comment, Combien, Où) comme suit :

- Quand : réactif-proactif. L'ajustement est à la fois réactif et proactif. Des méthodes statistiques des séries chronologiques sont utilisées pour la prédiction de la demande. Nous proposons une technique de prédiction pour absorber le temps d'initialisation d'une instance. En cas de violation du SLA, suite à une erreur de prédiction, des actions réactives seront déclenchées.

- Comment : vertical-horizontal. *HybridScale* considère le dimensionnement vertical et horizontal au niveau infrastructure. De plus, nous introduisons l'ajustement de contrôle d'admission par instance qui peut être vu comme un dimensionnement vertical. Au niveau application, nous proposons l'ajustement applicatif en traitant l'application comme une boîte blanche. Cet ajustement peut être également vu comme un dimensionnement vertical.

- Combien : méthode *RightCapacity*. Nous proposons la méthode *RightCapacity* pour calculer la capacité optimale. Cette méthode est basée sur la théorie des files d'attente. Elle est pilotée par *BestPolicies* qui est un ensemble de politiques de gestion de l'élasticité. L'originalité de *RightCapacity* est d'adresser plusieurs niveaux/couches. Elle tient en compte deux niveaux à savoir : l'application (SaaS) et l'infrastructure (IaaS).

- Où : application Web multi-tiers. *HybridScale* peut être appliquée à n'importe quelle application Web multi-tiers (par exemple e-commerce). Elle adresse à la fois le calcul, le stockage et la répartition de charge.

5.1.2 Modèle conceptuel

Afin de faire face à un environnement hautement dynamique comme le Cloud, il devient impératif de s'appuyer sur des modèles qui permettent au système de réagir au

contexte d'exécution pour le faire fonctionner de manière optimale (en termes de QdS et de coût) sans intervention humaine. Pour ce faire, nous nous appuyons sur l'informatique autonomique pour proposer *HybridScale*. Cette solution cherche à automatiser et optimiser la gestion de la capacité des ressources. Elle est basée sur une boucle de contrôle MAPE-K [Hor01].

La Figure 5.1 présente une vue d'ensemble du modèle conceptuel d'*HybridScale*. Dirigé par des objectifs de haut niveau (compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients (SLA)), le gestionnaire autonome utilise les informations (performance, workload, application, ressources) provenant des capteurs et de la base des connaissances (*BestPolicies*) pour analyser (*RightCapacity*), planifier et exécuter des actions (dimensionnement vertical/horizontal, ajustement de contrôle d'admission, ajustement de l'application) sur l'élément géré à savoir le service SaaS.

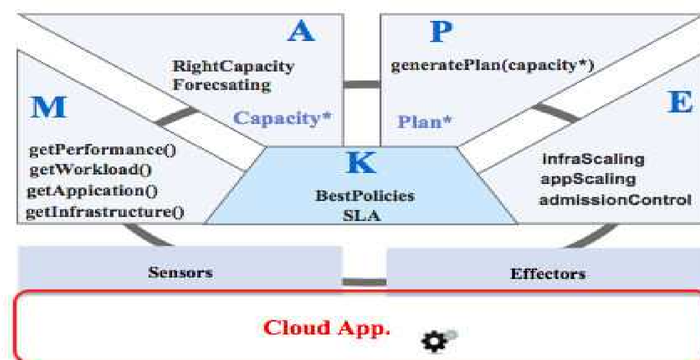


FIGURE 5.1 – HybridScale, modèle conceptuel

Les détails d'implémentation de la boucle autonomique d'*HybridScale* seront présentés dans le chapitre 6. Dans ce qui suit, nous présentons, d'abord, les politiques de gestion de l'élasticité (*BestPolicies*) qui constituent la base des connaissances (K). Ensuite, nous illustrons notre technique de prédiction de la demande. Enfin, nous développons, *RightCapacity*, notre méthode de planification de capacité (A). Notons que les différents algorithmes présentés dans ce chapitre ont été simplifiés afin d'en faciliter la lecture et la compréhension.

5.2 *BestPolicies* : politiques de gestion d'élasticité

Nous proposons de paramétrer la gestion de l'élasticité, et en particulier l'analyse (A), par des politiques (K). Nous distinguons trois types : i) les politiques des prix, les politiques de réaction et ii) les politiques de dimensionnement.

5.2.1 Politiques des prix

Les politiques des prix pratiquées par les différents fournisseurs IaaS peuvent influencer grandement les choix des consommateurs SaaS. Aussi, nous avons choisi de paramétrer le redimensionnement par les prix des ressources.

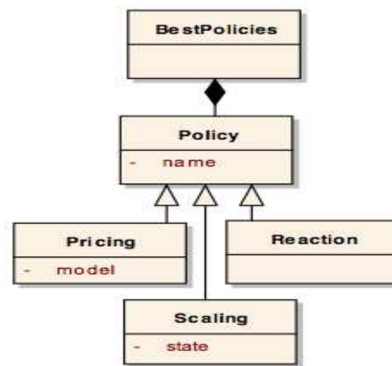


FIGURE 5.2 – BestPolicies

Un modèle de prix dynamique, comme *Amazon EC2 spot instances*[1], est préférable pour générer plus de revenus, mais est aussi plus compliqué à mettre en œuvre. C'est pourquoi, dans ce chapitre, nous considérons un modèle de prix des ressources statique.

Nous envisageons d'utiliser, également, un modèle de prix statique pour les services SaaS. Nous distinguons entre le prix à la demande (exemple : 1 €/requête) et le prix forfait (exemple : 100€/mois).

5.2.2 Politiques de réaction

Les décisions du redimensionnement automatique peuvent être prises à trois moments différents comme illustrés dans la Figure 5.3 : réactive, proactive ou hybride (réactive-proactive).

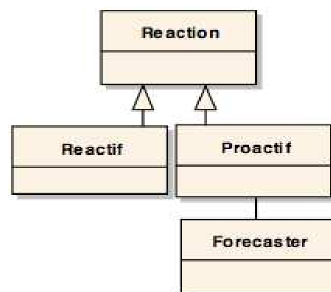


FIGURE 5.3 – Politiques de réaction

Réactive : La réaction réactive consiste à utiliser le contexte courant (la demande courante, l'état de système) pour prendre les mesures nécessaires pour augmenter ou réduire la quantité de ressources. Le modèle de provisioning à la demande convient bien à ce type de réaction. Cependant, il est difficile d'éviter les oscillations du système (l'effet "ping -pong"). De plus, le temps d'initialisation non négligeable des instances (allant de quelques secondes à quelques minutes) ralentit le retour à la stabilité.

[1]http ://aws.amazon.com/fr/ec2/spot-instances/

Proactive : La réaction proactive se base sur la prédiction de la future demande en vue de surmonter les questions soulevées par la réaction réactive. L'inconvénient de la réaction proactive est l'erreur de prédiction. En effet, l'exactitude des méthodes de prédiction dépend fortement de la fenêtre/intervalle de prédiction.

Hybride Afin de bénéficier des avantages des deux réactions et être capable d'ajuster finement la capacité, une réaction hybride sera nécessaire. Cette dernière est une combinaison des réactions réactives et proactives. La réaction proactive peut être appliquée pour ajuster la capacité à long terme en fonction de résultat de prédiction. En cas d'erreur de prédiction, la réaction réactive ajuste la capacité à court terme.

5.2.3 Politiques de dimensionnement

La méthode de planification de capacité *RightCapacity* (cf. Section 5.4) peut intégrer des politiques de dimensionnement qui vont influencer le calcul de la capacité optimale. Nous proposons les politiques suivantes : *Infrastructure Elasticity Policy* et *Application Elasticity Policy*. Chaque politique peut être activer/désactiver via l'attribut *state*.

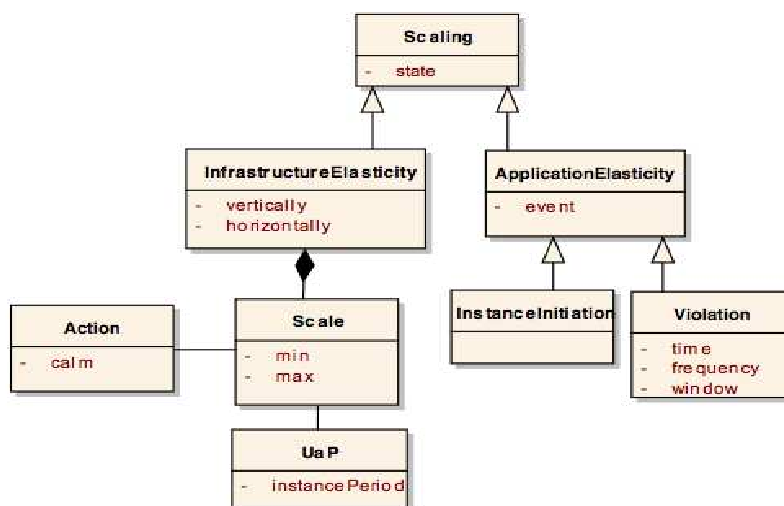


FIGURE 5.4 – Politiques de dimensionnement

i) Politique *Infrastructure Elasticity*

La politique *Infrastructure Elasticity* commande l'élasticité des ressources. Nous distinguons entre l'élasticité verticale et l'élasticité horizontale. La première consiste à ajuster (augmenter (*scale out*) ou diminuer (*scale in*)) le nombre d'instances (VMs) du système. Alors que la dernière permet d'ajouter (*scale up*) ou de supprimer (*scale down*) des ressources telles que CPU ou mémoire.

Max-Min : L'élasticité des ressources est contrôlée par plusieurs paramètres. Le paramètre *Max* définit la capacité maximale que le système peut atteindre. En effet, le mécanisme de dimensionnement ne distingue pas entre une charge de travail valide et malveillante. Sans paramètre *Max*, une simple attaque par déni de service (denial of service attack DoS) permet d'ajouter (*scale out* et/ou *scale up*) infiniment les ressources.

Nous définissons également la capacité minimum (*Min*) afin de contrôler le processus de suppression des ressources (*scale in* et *scale down*). La valeur par défaut de *Min* est égale à une instance par tier.

Use as You Pay : En plus de la limite *Min* qui freine la suppression des ressources, nous proposons la stratégie *Use as You Pay* (*UaP*) pour contrôler le moment de suppression. Cette stratégie consiste à utiliser toute "l'instance-période" avant de terminer n'importe quelle instance. En fait, la facturation par heure est souvent le modèle adopté par les fournisseurs IaaS. Par exemple, le tarif de Amazon EC2[2] correspond à une heure d'instance consommée pour chaque instance, à partir du moment où une instance est lancée jusqu'à ce qu'elle soit terminée. Chaque heure d'instance partielle consommée sera facturée comme une heure pleine ! Avec *UaP*, nous utilisons pleinement ce que nous payons exactement. En outre, cette politique permet d'éviter les oscillations du système en particulier face à une charge de travail de type "*On and Off*"[3]. Nous pouvons considérer cette politique comme une stratégie proactive. En fait, en attendant la fin d'une instance-heure avant de supprimer une instance, cette dernière peut être utile s'il y a une charge de travail croissante ou alternante (*On and Off*).

Calm : La fréquence des actions de dimensionnement est contrôlée par le paramètre *Calm*. Ce paramètre indique la durée pendant laquelle aucune décision de dimensionnement ne sera prise. L'objectif d'un tel paramètre est d'éviter les oscillations du système et de supporter les petits pics de charge. Chaque type de dimensionnement de ressources (*scale out*, *scale in*, *scale up* ou *scale down*) possède sa propre période de calme. La valeur par défaut d'une période de calme est égale à 1 minute.

ii) Politique *Application Elasticity*

La politique *Application Elasticity* dirige une dégradation de fonctionnalité (cf. chapitre 4). C'est un moyen pour absorber la charge lors d'un processus d'ajout des ressources (suite à une réaction réactive) qui est long (en particulier un *scale out*), pendant les petits pics de charge ou suite à une erreur de prédiction. Nous introduisons le terme *scale app* qui reflète la dégradation de fonctionnalité.

Absorption de démarrage : Pendant un dimensionnement horizontal (*scale out*) ou un dimensionnement vertical (*scale up*), la politique de dégradation de fonctionnalité déclenche le mode dégradé jusqu'à l'activation des ressources demandées ou ajustées. Dans ce cas, la durée de dégradation n'est pas définie à l'avance. Elle dépend fortement du temps d'initialisation maximum des instances (*scale out*) ou du temps d'ajout des ressources CPU, RAM (*scale up*).

Absorption de violation : La politique de dégradation de fonctionnalité utilise trois paramètres (une durée de dégradation, une fréquence de dégradation, une fenêtre de temps) pour faire face à une violation (suite à une augmentation de charge). La durée de dégradation est paramétrée selon le type/nature de charge de travail. Par défaut, elle est d'une durée égale à 1 minute. La fréquence d'utilisation de dégradation est limitée dans une fenêtre de temps et dépend du modèle de facturation des instances et donc du fournisseur IaaS. Nous distinguons deux cas : i) heure normale/pleine (exemple

[2] http ://aws.amazon.com/fr/ec2/pricing/
[3] http ://watdenkt.veenhof.nu/2010/07/13/workload-patterns-for-cloud-computing/

Amazon EC2), et heure naturelle/fixe (exemple Microsoft) (cf. Chapitre 2).

Pour l'heure normale, nous considérons une fréquence statique. Sinon pour le cas de l'heure naturelle, nous adoptons une fréquence dynamique de telle sorte à minimiser la fréquence entre $[0, 30]$ et à maximiser la fréquence entre $[31, 59]$. Avec de telles fréquences, l'ajout des ressources sera sollicité plus au début d'une heure naturelle afin d'utiliser le maximum possible d'une telle heure. Concrètement, une fréquence dynamique est présentée par deux (ou plusieurs) valeurs de fréquences de dégradation.

Dans le cas où le nombre des instances est limité (la politique *Infrastructure Elasticity*) et la capacité maximum est atteinte, nous notifions le fournisseur de service SaaS (sans faire l'ajout).

5.3 LoadForecasting : prédiction de charge de travail

Le temps d'initialisation d'une instance est généralement de quelques minutes. Cette période dépend d'une certaine quantité de facteurs incluant la taille de l'image et le système d'exploitation. Afin d'absorber le temps d'initialisation, nous avons proposé dans le chapitre précédent deux solutions à savoir la dégradation de QdS et la dégradation de fonctionnalité. Dans le présent chapitre, nous consolidons notre solution avec la prédiction de la charge du travail. Nous reposons sur les méthodes des séries chronologiques pour fournir un moyen de prédiction en se basant sur l'historique.

Dans ce qui suit, nous introduisons, d'abord, la modélisation des séries chronologiques. Ensuite, nous développons notre technique de prédiction.

5.3.1 Modélisation d'une série chronologique

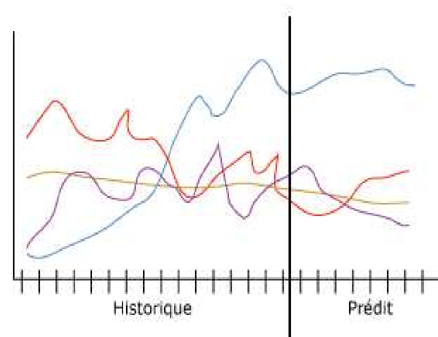


FIGURE 5.5 – Prédiction des valeurs futures

L'étude d'une série chronologique permet d'analyser, de décrire et d'exploiter un phénomène au cours du temps. Nous utilisons les séries chronologiques pour la prédiction des valeurs futures (intervalle de prédiction) de la demande à partir des valeurs observées (fenêtre de prédiction). Cette analyse est illustrée par la Figure 5.5. Les informations sur l'historique apparaissent à gauche du trait vertical alors que les prévisions effectuées apparaissent à droite du trait vertical.

Une série chronologique est constituée, en général, de trois effets (ou composantes) [BJ94] :

- un effet à long terme, appelé tendance, composante tendancielle ou *trend*,
- un effet dit saisonnier, qui réapparaît à intervalles réguliers. Cet effet se traduit par une composante de la série appelée composante saisonnière,
- un effet inexpliqué. Cet effet que l'on suppose en général dû au hasard, se manifeste par des variations accidentelles.

La modélisation d'une série chronologique spécifie les rapports entre les composantes via un modèle (une équation). Nous distinguons principalement le modèle additif et le modèle multiplicatif. Nous avons choisi le modèle additif (équation 5.1) pour modéliser nos séries chronologiques. Ce modèle est le modèle de base.

$$X_t = f_t + s_t + E_t + p_t \tag{5.1}$$

Où f_t est la tendance, s_t représente la composante saisonnière, E_t représente la variation aléatoire due à de nombreuses causes pas forcément bien identifiées, mais de répercussion limitée et p_t représente les perturbations majeures liées à des événements importants mais qui ne se répètent pas et dont l'influence est limitée dans le temps.

5.3.2 Forecaster, la technique

Nous proposons un composant *Forecaster* pour prédire les valeurs futures de charge. Dans ce qui suit, nous commençons par illustrer une vue d'ensemble de notre solution. Ensuite, nous développons le processus de prédiction.

i) Modèle conceptuel

La Figure 5.6 illustre notre *Forecaster*. A partir des observations (l'historique), notre algorithme fournit les valeurs futures comme sortie. L'algorithme de prédiction se base sur des méthodes de prédiction (*ForecastingMethod*) et modèles de précision (*ForecastingAccuracy*).

Nous utilisons la librairie OpenForecast [4] qui fournit l'implémentation de méthodes de prédiction. Nous rappelons que l'objectif de la prédiction de la demande est d'absorber le temps d'initialisation des instances. Ce temps est de l'ordre de quelques minutes. C'est pourquoi, nous nous intéressons à cette librairie qui fournit des méthodes de prévision à court terme. Parmi ces méthodes nous pouvons citer : la moyenne mobile (Moving Average), le lissage exponentiel (Exponential Smoothing) et la régression (Regression) [BJ94]. Néanmoins, il est possible d'utiliser d'autres librairies implémentant plus de méthodes de prédiction ou d'implémenter ces propres méthodes.

Les modèles de précision permettent d'évaluer les méthodes de prédiction. Intuitivement, plus l'erreur de prévision est petite et meilleure est la méthode. L'équation

[4] http ://www.stevengould.org/software/openforecast/index.shtml

5.2 illustre l'erreur de prévision : la différence entre la valeur réelle et la valeur prévue pour une période donnée.

$$E_t = Y_t - F_t \tag{5.2}$$

Où E est l'erreur de prédiction dans la période t, Y est la valeur courante à la période t et F est la valeur prédite à la période t. Parmi les modèles de précision, nous pouvons citer : l'erreur absolue moyenne (Mean absolute error - MAE) et l'écart absolu moyen (Mean Absolute Deviation - MAD) [HK06].

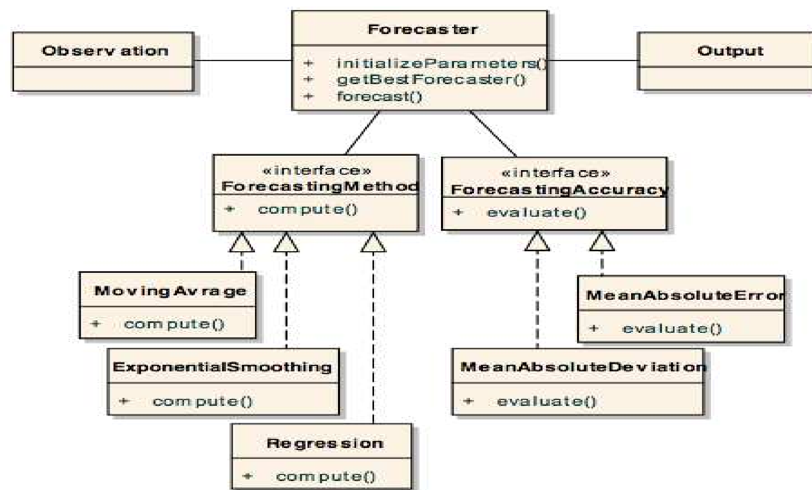


FIGURE 5.6 – Méta-modèle Forecaster

ii) Technique de prédiction

La technique de prédiction développée est basée principalement sur trois méthodes : i) initializeParameters, ii) getBestForecaster et ii) forecast (voir la Figure 5.7) :

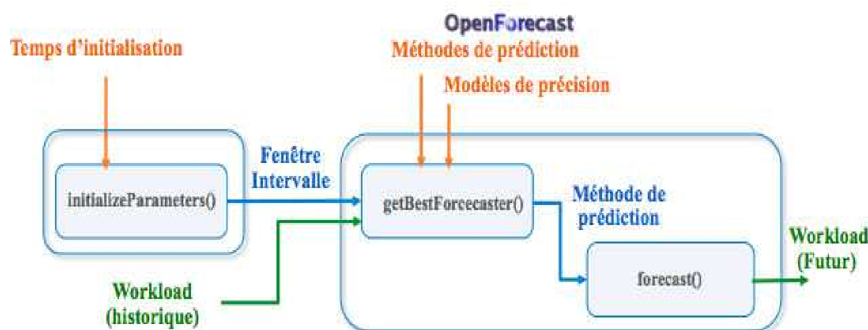


FIGURE 5.7 – Technique de prédiction

initializeParameters : paramétrage de l'intervalle et fenêtre de prédiction L'intervalle de prédiction (futur) est paramétré en fonction des temps d'initialisation des instances. Nous proposons deux manières d'utiliser cette méthode : i) statique et ii) dynamique. La manière statique consiste à faire un seul appel à cette méthode pour initialiser l'intervalle de prédiction. Dans ce cas, les temps d'initialisation sont calculés suite à des tests réalisés sur les différents types d'instances utilisées par le système. La

deuxième alternative est de calculer les paramètres de prédiction à chaque prédiction en se basant sur les derniers temps d'initialisation observés. Le temps d'initialisation inclut le temps de démarrage des instances et le temps d'enregistrement auprès du répartiteur de charge dans notre architecture n-tiers dans notre cas. Dans le reste de ce chapitre, nous adoptons la manière statique.

La valeur de l'intervalle de prédiction est égale à la moyenne de temps d'initialisation. La fenêtre de prédiction est égale au moins à 60 observations.

getBestForecaster : sélection de méthode de prédiction Cette partie est basée sur la librairie OpenForecast. Elle consiste à appliquer les différentes méthodes de prédiction aux valeurs observées en utilisant diverses combinaisons des variables indépendantes puis sélectionner la meilleure méthode qui a la plus petite valeur d'erreur de prédiction.

forecast : prédiction des valeurs Une fois, la méthode de prédiction sélectionnée, la dernière partie est dédiée à la prédiction des valeurs futures de la charge de travail.

Ainsi, notre composant *Forecaster* permet de générer les futures charges en paramétrant aisément les intervalles/fenêtres de prédiction.

5.4 *RightCapacity* : planification de capacité dirigée par SLA

Dans cette section, nous proposons *RightCapacity* : une méthode de planification de capacité. L'originalité de *RightCapacity* est d'adresser les aspects multi-couches (*cross-layer*). Il prend en compte deux niveaux à savoir : l'application (SaaS) et l'infrastructure (IaaS).

Dans ce qui suit, nous présentons, d'abord, le modèle conceptuel de *RightCapacity*. Ensuite, nous développons la méthode de planification niveau IaaS puis niveau SaaS.

5.4.1 Modèle Conceptuel

La Figure 5.8 illustre le modèle conceptuel de *RightCapacity*. Notre solution calcule la capacité optimale (*getOptimalCapacity()*) pour répondre aux objectifs de niveau de service (SLA) d'une manière rentable.

Nous introduisons l'élasticité au niveau application en la considérant comme une boîte blanche. *RightCapacity* calcule l'architecture applicative appropriée (mode) via *getAppropriateArchitecture()* afin d'absorber le démarrage des instances (temps d'initialisation) et absorber les violations (les petits pics de charge et/ou erreur de prédiction).

5.4.2 Modèle analytique

Nous proposons un modèle analytique, basé sur la théorie des files d'attente, qui fournit une estimation précise de la performance, la disponibilité et le coût d'un service SaaS avec une configuration donnée pour une charge déterminée.

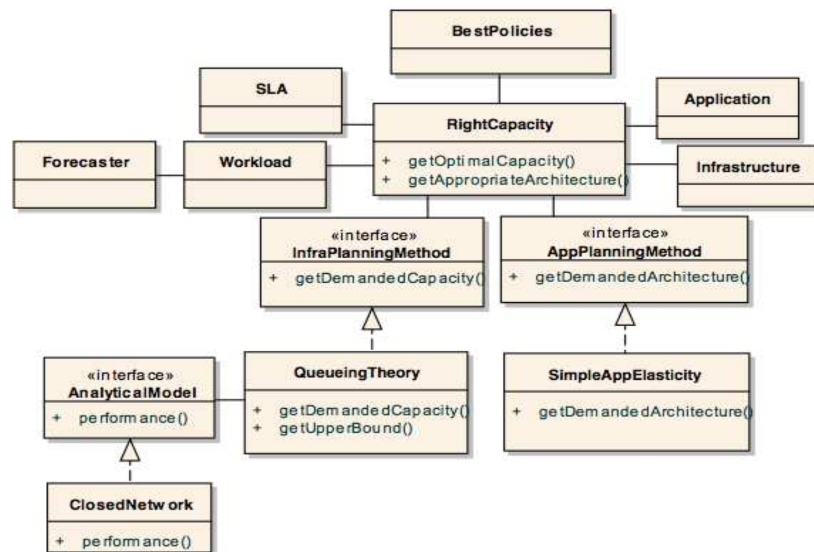


FIGURE 5.8 – Méta-modèle RightCapacity

Dans ce qui suit, nous détaillons notre modèle analytique en illustrant d'abord la modélisation puis l'évaluation de performance.

i) Théorie des files d'attente

Dans cette section, nous justifions, d'abord, le choix de la théorie des files d'attente. Ensuite, nous développons la modélisation de notre modèle analytique.

a) Pourquoi ?

Le concept de file d'attente représente à la fois une ressource critique en terme de performance, ainsi que la file de requêtes en attente pour accéder à cette ressource.

Un réseau des files d'attente est un ensemble des files d'attente interconnectées, dans lesquelles circulent une ou plusieurs classes de clients (requêtes). Un réseau mono-classe est un réseau parcouru par une seule classe de clients alors que le réseau multi-classes est un réseau dans lequel circulent plusieurs classes.

La théorie des files d'attente est largement utilisée pour l'évaluation des performances des systèmes. Elle est parfaitement adéquate pour des systèmes considérés comme un ensemble des boîtes noires (pas de détails de fonctionnement interne). Ainsi, les modèles d'évaluation de performance associés à cette théorie comme l'analyse par valeur moyenne (Mean Value Analysis- MVA) [RL80] évaluent les métriques nécessaires comme le temps de réponse et la disponibilité de service.

b) Analyse par valeur moyenne - MVA

MVA un algorithme récursif, développé initialement par Reiser [RL80]. Il consiste à exprimer les paramètres de performance moyenne d'un réseau à l'étape où il contient N clients, en fonction de ceux de l'étape de N-1 clients. Il suffit alors d'itérer ces équations en partant des conditions initiales jusqu'à atteindre le nombre total de clients. Les équations qui relient les paramètres de performance se basent sur deux concepts clés :

la formule de Little et le théorème des arrivées [MDA04].

La **formule de Little** est une loi générale qui s'énonce comme suit : le nombre moyen des clients dans un système est égal au produit de la fréquence moyenne d'arrivée (débit du système) par le temps moyen passé dans le système par chaque client.

Le **théorème des arrivées** (*arrival theorem*) consiste à exprimer le temps moyen de séjour d'un client à la station i quand le réseau contient N clients en fonction du nombre moyen de clients à la station i lorsque le réseau contient $N-1$ clients.

ii) Modélisation d'un service SaaS

Nous suivons une approche de réseau des files d'attente, où un système multi-niveaux est modélisé comme une file d'attente $M/M/c/K$. Les services SaaS sont modélisés comme des réseaux fermés pour refléter le modèle de communication synchrone. Autrement dit, un client attend une réponse à sa demande en cours avant d'envoyer une autre demande. La Figure 5.9 illustre un exemple d'un service SaaS implémenté par une architecture multi-tiers.

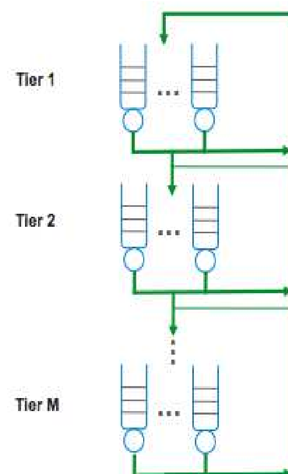


FIGURE 5.9 – Modélisation d'une architecture M-tiers avec des réseaux fermés

Nous étendons l'algorithme MVA pour tenir compte des caractéristiques principales suivantes :

- multi-niveaux : supporter n'importe quel service multi-tiers,
- multi-station : considérer la réplication par niveau,
- multi-classe : admettre plusieurs SLAs.

Notre algorithme prend l'ensemble des paramètres suivants (voir Figure 5.10) : l'application (service SaaS), l'infrastructure (ressources), le workload et les SLAs. Il fournit pour chaque classe c des clients le temps de réponse (σ_c), le taux de rejet (α_c) et le coût financier (ω).

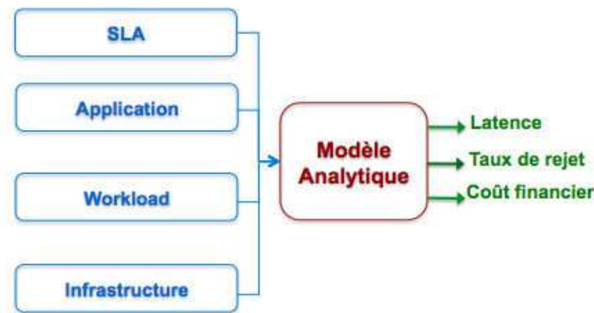


FIGURE 5.10 – Modèle Analytique

a) Infrastructure

La configuration des ressources d'une application SaaS est caractérisée par sa structure topologique (voir Figure 5.11). Soit M le nombre de niveaux, $T_1, ..., T_M$. Chaque niveau est un ensemble d'instances. Une instance peut être composée de trois types de ressource (réseau, stockage et calcul) plus une image. Cette dernière contient un ensemble de packages. Nous utilisons K_m pour désigner le degré de réplication au niveau T_m ($m = 1, ..., M$). MPL_m représente le MPL (Multi-Programming Level) au niveau T_m. Le MPL consiste à fixer une limite pour le nombre maximal de clients autorisés à accéder simultanément à une instance. Soit Pr_m le prix, par exemple : heure consommé, pour chaque instance au niveau T_m. Nous utilisons Sc_m pour désigner le prix des logiciels et coûts de licences au niveau T_m.

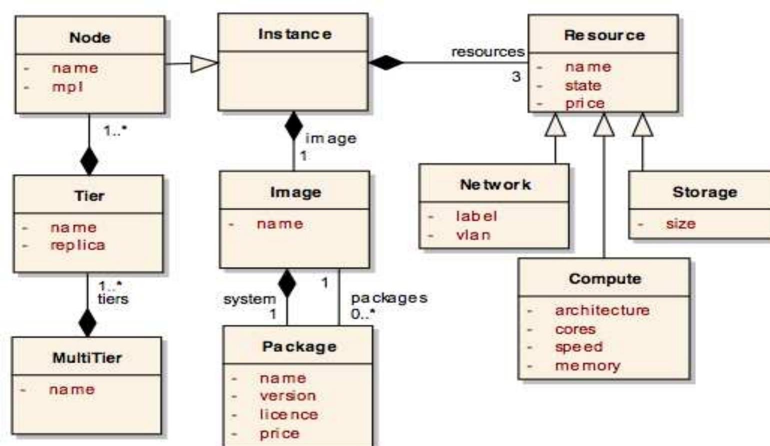


FIGURE 5.11 – Méta-modèle Infrastructure

b) Application

Une application est caractérisée par son mode de fonctionnement. Ce mode (voir Figure 5.12) dépend des composants qui sont en cours d'exécution, leurs propriétés (intra-composants), et comment ces éléments sont liés les uns aux autres (inter-composant). Soit I le nombre de composants $Com_1, ..., Com_I$. Chaque composant Com_i fournit J services. Un service j a un nombre fixe de modes L. Dans le cadre de ce travail, nous distinguons deux modes de fonctionnement : i) mode normal et ii) mode dégradé. Un exemple de dégradation est l'affichage 2D (vs 3D comme mode normal) ou la limitation

du nombre et du niveau de détails des réponses renvoyées au client. Nous utilisons *Mode* et $mode_{i,j}$ pour désigner respectivement le mode de l'application et le mode de service j de composant i

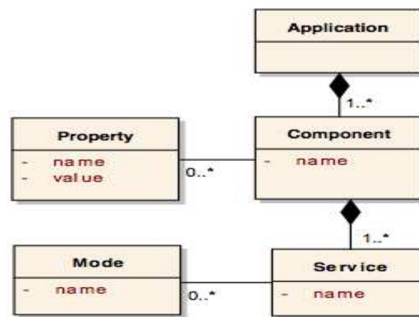


FIGURE 5.12 – Méta-modèle Application

c) Workload

La charge de travail du service se caractérise, d'une part, par le nombre de clients qui tentent d'accéder à un serveur, et d'autre part, par la nature de la charge de travail (voir Figure 5.13). Soit C le nombre de classes d'utilisateurs. Chaque classe C a un nombre fixe d'utilisateurs N_c avec un Z_c qui est le temps écoulé entre la réception d'une réponse et l'envoi de la prochaine demande d'un client. $S_{c,m}$ et $V_{c,m}$ désignent respectivement le temps de service et le taux de demande de classe c au niveau m. D_c est le délai de communication inter-niveaux. Soit N le nombre total des clients défini comme $N = \sum_{c=1}^{C} N_c$.

Selon la politique de réaction, le paramètre workload peut être soit la charge courante (gestion réactive) soit la future charge (gestion proactive).

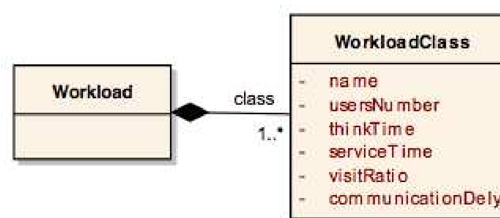


FIGURE 5.13 – Méta-modèle Workload

d) SLA

Du point de vue du fournisseur de SaaS, il y a deux niveaux de SLA :

- SLA_S entre l'utilisateur final et le fournisseur SaaS,
- SLA_R entre le fournisseur SaaS et fournisseur IaaS.

Dans le reste de ce chapitre, nous considérons que SLA_S sera basé sur le temps de

réponse et la disponibilité de service, tandis que SLA_R sera basé sur la disponibilité des ressources (cf. fil conducteur, Chapitre 1). Chaque SLA est modélisé avec le langage *CSLA*.

iii) Évaluation de performance, disponibilité et coût financier

Cette section décrit l'évaluation de performance, disponibilité et coût financier d'un service SaaS. Notre évaluation est divisée en deux parties : i) identifier le nombre des requêtes admises et rejetées et ii) prédire la latence, la disponibilité et le coût financier.

a) Contrôle d'admission

La première partie est dédiée au calcul de la quantité de requêtes admises et la quantité de requêtes rejetées sur chaque tier de service SaaS. Afin de gérer plusieurs classes de requêtes, nous considérons une politique de priorité statique basée sur les classes. Les requêtes de même priorité (classe) sont regroupées dans une file de type FIFO. Il y a autant des files qu'il y a de niveaux de priorité (classe).

L'algorithme 3 décrit la première partie d'évaluation. En fonction des ratios de visite, les lignes 1 à 7 calculent $Nt_{c,m}$, le nombre de requêtes par classe tentant d'accéder au tier T_i. En particulier, la ligne 7 reflète le modèle de communication synchrone où le nombre de requêtes tentant d'accéder à T_m sont au plus le nombre de requêtes admises à T_{m-1}.

Les lignes 8 à 14 appliquent le MPL pour calculer le nombre de requêtes admises ($Na_{c,m}$) et le nombre de requêtes rejetées ($Nr_{c,m}$) par classe à chaque tier T_m. La fonction *admissionControl(c,m)* (ligne 13) définit le nombre des requêtes acceptables de la classe c selon des priorités prédéfinies.

Une requête admise sur T_m peut être rejetée par un des tiers suivants ($T_{m+1}...T_M$). Les lignes 15 et 18 calculent le nombre de requêtes admises sur T_m et non rejetées par les étages suivants. Enfin, les lignes 19 à 23 produisent $Nr_{c,m}$, le nombre total de requêtes rejetées, et le nombre total de requêtes admises $Na_{c,m}$ par classe c.

b) Prédiction

Dans ce travail, la performance est réduite à la latence ou le temps de réponse : c'est le temps qui sépare l'envoi de la requête et la réception du résultat par un client. La disponibilité de service SaaS est calculée en fonction de taux de rejet sur un intervalle de temps donné.

La deuxième partie de l'évaluation est illustrée par l'algorithme 4. Ce dernier consiste à prédire la latence, le taux de rejet et le coût financier. Les lignes 1 à 5 initialisent la longueur des files d'attente et la demande de service sur chaque tier. Les lignes 6 à 19 parcourent les différents tiers, depuis le dernier vers le premier. La ligne 9 calcule la demande de service alors que la ligne 10 estime le temps de réponse des requêtes. La fonction *Max* garantit que la demande de service n'est pas inférieure au temps incompressible de traitement. Puis, les lignes 11 à 16 cumulent les temps de réponse pour calculer la latence des requêtes admises aux tiers $T_m...T_M$, et la latence

des requêtes admises par T_m mais rejetées par un des tiers suivants. Les lignes 17 à 19 calculent la longueur des files en utilisant la loi de Little.

Algorithme 3: admissionControl

```
   Input : M, K_m, MPL_m, N_c, Z_c, S_cm, V_cm (m = 1, ..M, c = 1...C)
   Output : Nr_c, Na_c (c = 1...C)
1  for m = 1 to M do
2      if m == 1 then
3          for c = 1 to C do
4              Nt_{c,m}=N_c ;
5      else
6          for c = 1 to C do
7              Nt_{c,m}=Min(Na_{c,m-1}, Na_{c,m-1} * V_{c,m}/V_{c,m-1});
8      Nt_m=∑_{c=1}^{C} Nt_{c,m} ;
9      TMPL_m = ∑_{k=1}^{K_m} MPL_k;
10     Na_m = Min(TMPL_m, Nt_m);
11     Nr_m = Nt_m − Na_m;
12     for c = 1 to C do
13         Na_{c,m}=Nt_{c,m}- admissonControl(c, m);
14         Nr_{c,m}=Nt_{c,m} − Na_{c,m};
15 for m = 1 to M do
16     Nap_m=Na_m-∑_{j=m+1}^{M} Nr_j;
17     for c = 1 to C do
18         Nap_{c,m}=Na_{c,m}-∑_{j=m+1}^{M} Nr_{c,j} ;
19 Nr = ∑_{m=1}^{M} Nr_m;
20 Na = N − Nr;
21 for c = 1 to C do
22     Nr_c=∑_{m=1}^{M} Nr_{c,m};
23     Na_c=N_c − Nr_c;
24 return Nr_c, Na_c (c = 1, ...C)
```

Latence : La latence totale d'une requête d'une classe c est définie par la ligne 21 comme étant la latence d'une requête admise sur T_1 et jamais rejetée par les étages suivants.

Taux de rejet : Le taux de rejet par classe (ligne 25) est défini en fonction des requêtes rejetées au tier T_1 et des requêtes admises par T_1 mais rejetées par un des tiers suivants.

Coût financier : Le coût financier est calculé en terme de ressources allouées et des pénalités payées. Nous utilisons le modèle économique présenté dans le chapitre précédent (cf. chapitre 4).

5.4.3 Planification des ressources, la méthode

Les variables MPL_m et K_m ($m = 1, ..., M$) sont les inconnues de la méthode de la planification de la capacité. Les valeurs des variables sont prises à partir d'un domaine qui est défini par le modèle analytique et la base des connaissances. Les contraintes sont les SLAs et les politiques de gestion de l'élasticité. L'objectif est de maximiser le revenu

Algorithme 4: performance

Input : $M, K_m, Pr_m, Sc_m, MPL_m, SLA_c, N_c, Z_c, S_{cm}, V_{cm}$ $(m = 1, ..M, c = 1...C)$
Output : $\sigma_c, \alpha_c, \omega$ $(c = 1, ...C)$

1 (Nr_c, Na_c)=*admissionControl*(..);
2 **for** $m = 1$ **to** M **do**
3 $Ql_m = 0$;
4 **for** $c = 1$ **to** C **do**
5 $W_{c,m} = (S_{c,m} - D_{c,m}) * V_{c,m}$;
6 $W_c = \sum_{c=1}^{C} W_{c,m}$;
7 **for** $m = M$ **to** 1 **do**
8 **for** $c = 1$ **to** C **do**
9 **for** $n = 1$ **to** $Na_{c,m}$ **do**
10 $Wp_m = \frac{(1+Ql_m)*W_{c,m}}{K_m}$;
11 $R_{c,m} = Max(Wp_m, W_{c,m}) + D_{c,m} * V_{c,m}$;
12 **if** $m == 1$ **then**
13 $\sigma a_{c,m} = R_{c,m} + \sigma a_{c,m+1}$;
14 $\sigma r_{c,m} = R_{c,m} + \sigma r_{c,m+1}$;
15 **else**
16 $\sigma a_{c,m} = R_{c,m}$;
17 $\sigma r_{c,m} = R_{c,m}$;
18 $\tau a_{c,m} = \frac{(n*Nap_{c,m}/Na_{c,m})}{(\sigma a_{c,m}+Z_c)}$;
19 $\tau r_{c,m} = \frac{(n*(Na_{c,m}-Nap_{c,m})/Na_{c,m})}{(\sigma r_{c,m}+Z_c)}$;
20 $Ql_m = ((\tau a_{c,m} + \tau r_{c,m}) * R_{c,m})$;
21 **for** $c = 1$ **to** C **do**
22 $\sigma_c = \sigma a_{c,1}$;
23 $\tau a_c = \frac{Na_c}{\sigma a_{c,1}+Z_c}$;
24 $\tau r_c = \frac{Nr_c-Nr_1}{\sigma r_{c,1}+Z_c}$;
25 $\tau rp_c = \frac{Nr_1}{Z_c}$;
26 $\alpha_c = \frac{\tau r_c+\tau rp_c}{\tau r_c+\tau rp_c+\tau a_c}$;
27 **return** $\sigma_c, \alpha_c, \omega$ $(c = 1, ...C)$

de fournisseur SaaS (minimiser le coût de service et minimiser les violations SLA). La valeur de workload dépend de la nature de l'analyse : valeur courante observée (analyse réactive) et valeur future suite à un appel au composant *Forecaster* (analyse proactive). La méthode de planification de capacité (voir Figure 5.14) se compose de trois algorithmes principaux : i) *getUpperBound*, iii) *getDemandedCapacity* et *getOptimal-Capacity*.

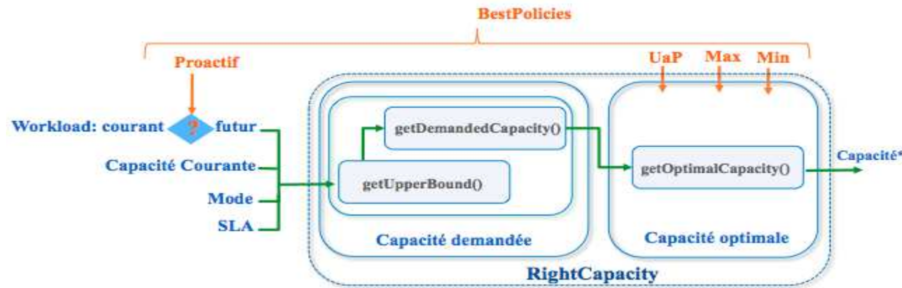


FIGURE 5.14 – Méthode de planification de capacité

Dans ce qui suit, nous détaillons les différents algorithmes de la méthode de planification de capacité au niveau infrastructure.

i) Capacité demandée

Nous rappelons que le SLA_S spécifie la performance (temps de réponse maximal) et la disponibilité de service (taux de rejet maximum) pour chaque classe des clients. La performance de service SaaS pour une classe c est illustrée par l'équation 5.3.

$$\rho_c(\sigma_c, \alpha_c) = (\sigma_c \leq \sigma_{max,c}).(\alpha_c \leq \alpha_{max,c}) \tag{5.3}$$

Où $\sigma_{max,c}$ et $\alpha_{max,c}$ sont respectivement le seuil de temps de réponse et le seuil de disponibilité pour un client de la classe c. σ_c et α_c sont les valeurs de temps de réponse et la disponibilité observées ou calculées via notre modèle analytique.

L'équation 5.4 calcule la performance pour C classes

$$\rho_C = \prod_{c=1}^{C} \rho_c(\sigma_c, \alpha_c) \tag{5.4}$$

Où C est le nombre de classes

Nous introduisons une fonction d'utilité (voir équation 5.5 qui permet de calculer la capacité demandée.

$$\theta(M, \rho_C, \omega) = \frac{M.\rho_C}{\omega} \tag{5.5}$$

Où M est le nombre des tiers et ω et coût du service.

La fonction d'utilité (θ) sera nulle si les objectifs (SLOs) ne sont pas respectés. Notre but est de trouver une valeur élevée de θ qui reflète un respect des objectifs avec un

coût faible. Pour cette fin, nous introduisons deux algorithmes *getUpperBound* et *getDemandeCapacity*.

La méthode de planification de la capacité vérifie d'abord si la capacité courante garantit les objectifs de niveau de services pour la demande. Si les objectifs sont garantis, nous considérons que les valeurs actuelles de K_m ($m = 1, ..., M$) comme borne supérieure et l'algorithme *getDemandedCapacity* sera invoqué. Sinon, en cas de violation, nous augmentons le nombre des instances affectées à tous les tiers de l'application Cloud jusqu'à trouver une valeur de K_m qui réponde aux objectifs de niveau de services. Nous utilisons notre modèle analytique à chaque incrémentation pour calculer les performances (voir Algorithme 5).

Algorithme 5: getUpperBound

```
Input : M, K_m, MPL_m, SLA_c, N_c, Z_c, S_cm, V_cm (m = 1,..M, c = 1...C)
Output : K_m, MPL_m (m = 1,..M)
1  (σ_c, α_c, ω)=performance(...);
2  while ρ_C! = 1 do
3      for m = 1 to M do
4          K_m++;
5          if m == 1 then
6              MPL_m = (Σ_{c=1}^{C} N_c * V_{c,m}) / K_m;
7          else
8              MPL_m = (MPL_{m-1} * K_{m-1}) / K_m;
9      (σ_c, α_c, ω)=performance(...);
10 return K_m, MPL_m (m = 1,..M)
```

De la borne supérieure trouvée de K_m, l'algorithme *getDemandedCapacity* estime la capacité demandée. Nous définissons la capacité demandée comme la valeur minimum de K_m. Pour estimer efficacement la valeur minimale de K_m, une recherche dichotomique sur l'intervalle $[1..K_m]$ est effectuée comme mentionné dans l'algorithme 6.

La ligne 4 modifie la valeur de K_m. Le MPL_m est ajusté en conséquence (lignes 5-8) pour répartir la demande sur les K_m instances par tier. Nous distinguons deux cas après l'invocation de notre algorithme de performance : i) les objectifs (SLOs) sont garantis donc nous diminuons la borne K_{max} (ligne 11), ou ii) les objectifs (SLOs) ne sont pas garantis. Dans ce cas, nous différencions entre le dépassement de seuil de disponibilité (lignes 13-14) et le dépassement de seuil de temps de réponse (lignes 15-29).

Le dépassement de seuil de disponibilité reflète soit des priorités mal ajustées ou une capacité insuffisante. Nous traitons en particulier la deuxième cause en augmentant la borne K_{min} (ligne 14).

Le dépassement de seuil de temps de réponse signifie que le contrôle d'admission est mal paramétré. Nous utilisons une recherche dichotomique pour calculer les valeurs de MPL_m pour chaque tier. Le raisonnement est le même. Nous modifions la valeur de MPL_m (ligne 20). L'invocation de notre algorithme de performance permet d'ajuster les bornes MPL_{max} (ligne 23) et MPL_{min} (ligne 25).

Après l'ajustement de MPL_m, si les objectifs sont garantis, nous réduisons la borne K_{max} (ligne 27) sinon nous augmentons K_{min} (ligne 29). L'algorithme poursuit la recherche dans le nouvel intervalle.

Algorithme 6: getDemandedCapacity

```
Input : M, K_m, MPL_m, SLA_c, N_c, Z_c, S_cm, V_cm (m = 1, ..M, c = 1...C)
Output : K_m, MPL_m (m = 1, ..M)
1  for m = 1 to M do
2      K_min = 1; K_max = K_m;
3      while K_min ! = K_max do
4          K_m = K_min + (K_max − K_min)/2;
5          if m == 1 then
6              MPL_m = (Σ_{c=1}^{C} N_c * V_{c,m}) / K_m;
7          else
8              MPL_m = (MPL_{m−1} * K_{m−1}) / K_m;
9          (σ_c, α_c, ω)=performance(...);
10         if ρ_C == 1 then
11             K_max = K_m;
12         else
13             if one of(α_c > α_{max,c}) then
14                 K_min = K_m + 1;
15             else
16                 for mm = 1 to M do
17                     MPL_min = 1;
18                     MPL_max = MPL_mm;
19                     while MPL_min ! = MPL_max do
20                         MPL_mm = MPL_min + (MPL_max − MPL_min)/2;
21                         (σ_c, α_c, ω)=performance(...);
22                         if ρ_C == 1 then
23                             MPL_max = MPL_mm;
24                         else
25                             MPL_min = MPL_mm + 1;
26                 if ρ_C == 1 then
27                     K_max = K_m;
28                 else
29                     K_min = K_m + 1;
30 return K_m, MPL_m (m = 1, ..M)
```

ii) Capacité optimale

La capacité optimale est le résultat de l'application des politiques de gestion de l'élasticité sur la capacité demandée (voir Algorithme 7). Dans un premier temps, nous calculons la différence entre la capacité (configuration) courante et la capacité demandée afin d'identifier les ajustements nécessaires. La capacité optimale est le résultat de zéro ou plusieurs ajustements : ajustement de contrôle d'admission ou ajustement des ressources.

Algorithme 7: getOptimalCapacity

```
    Input : currentCapacity, demandedCapacity, Min, Max, instanceTime
    Output : K_m, MPL_m (m = 1, ..M)
 1  for m = 1 to M do
 2      MPL_m = demandeCapacity.getMPL(m);
 3      K_m = demandeCapacity.getK(m);
 4  delta = diff(currentCapacity, demandedCapacity);
 5  if add then
 6      if demandedCapacity > Max then
 7          Notification;
 8      K_m = min(demandeCapacity, Max);
 9  if remove then
10      if demandedCapacity < Min then
11          Notification;
12      if currentCapacity == Min then
13          Notification;
14      else
15          for m = 1 to M do
16              K_m = currentCapacity.getK(m);
17              if delta.getK(m)! = 0 then
18                  for k = 1 to currentCapacity.getK(m) do
19                      if (duration_i%instanceTime) > (0, 95.instanceTime) then
20                          toTerminate_m.add(i);
21                  toTerminate_m.sort();
22                  K_m = K_m − min(toTerminate_m.sort(), delta.getK(m));
23  if oneof(K_m! = demandeCapacity.getK(m)) then
24      for m = 1 to M do
25          if m == 1 then
26              MPL_m = (∑_{c=1}^{C} N_c * V_{c,m}) / K_m;
27          else
28              MPL_m = (MPL_{m−1} * K_{m−1}) / K_m;
29  return K_m, MPL_m (m = 1, ..M)
```

Nous distinguons 3 cas possibles d'ajustement des ressources :

Ajout des ressources (*scale up*, *scale out*) : La capacité optimale est limitée par la politique *Infrastructure Elasticity* (la capacité maximum pré-definie). Dans le cas où la capacité demandée est supérieure à la capacité maximum autorisée, une notification sera déclenchée.

Suppression des ressources (*scale down*, *scale in*) : La capacité optimale est dirigée par la stratégie "UaP" (si elle est activée). Une instance sera supprimée si elle a consommé plus de 95% de son temps-instance. Nous vérifions, également, la capacité minimale.

Ajout et suppression : Nous distinguons deux possibilités :i) ressources homogènes entre tiers, ii) ressources hétérogènes entre tiers. Si les ressources sont homogènes, une migration des instances entre tiers est possible. Dans ce travail, nous utilisons des ressources hétérogènes entre tiers.

A la fin de l'algorithme, si la capacité optimale est différente de la capacité demandée, nous adaptons les valeurs de *MPL* avec la capacité optimale pour ajuster le contrôle d'admission.

5.4.4 Planification de l'architecture logicielle, la méthode

La planification de l'architecture logicielle consiste à définir le mode de fonctionnement pour chaque service de l'application. Les variables $mode_{i,j}$ $(i = 1, ..., I; j = 1, ..., J)$ sont les inconnues de la méthode de la planification de l'architecture. Cette méthode est basée sur les algorithmes : i) *getAppropriateArchitecture()* et ii) *getDemandedArchitecture()*.

Nous proposons la planification de capacité applicative *SimpleAppElasticity* (cf. Figure 5.8). Pour des raisons de simplicité, nous considérons deux modes pour une application :

- mode normal : tous les services de l'application sont en mode normal,
- mode dégradé : tous les services de l'application (qui proposent deux modes ou plus) sont en mode dégradé.

L'architecture demandée sera donc soit en mode normal ou en mode dégradé. L'algorithme 8 consiste à planifier le mode de chaque service de l'application en fonction du paramètre *mode*.

L'algorithme 9 permet de fournir l'architecture appropriée en fonction des évenements. Suite à un dépassement d'un seuil d'un des objectifs du SLA, l'architecture appropriée est le mode dégradé. Après la durée de dégradation ou l'activation des instances, l'architecture appropriée passe en mode normal. Une fois le mode est défini, il sera utilisé en invoquant l'algorithme 8 pour spécifier le mode de chaque service.

Algorithme 8: getDemandedArchitecture

```
   Input : mode, mode_{i,j} (i = 1, ..I, j = 1...J)
   Output : mode_{i,j} (i = 1, ..I, j = 1...J)
 1 for i = 1 to I do
 2     for j = 1 to J do
 3         if L_{i,j} >1 then
 4             mode_{i,j} = mode;
 5 return mode_{i,j} (i = 1, ..I, j = 1...J)
```

Algorithme 9: getAppropriateArchitecture

```
   Input : event, context
   Output : mode_{i,j} (i = 1, ..I, j = 1...J)
 1 if event == thresholdExceeding then
 2     mode = degraded;
 3 else if event == endDegradationDuration then
 4     performance(mode = normal, context);
 5     if ρ_C == 1 then
 6         mode = normal;
 7 else if event == activateInstances then
 8     mode = normal;
 9 getDemandedArchitecture(mode, ...);
10 return mode_{i,j} (i = 1, ..I, j = 1...J)
```

5.5 Synthèse

Ce chapitre détaille les éléments de base de la solution *HybridScale* qui fournissent des réponses aux limitations soulevées au niveau des travaux de l'état de l'art à savoir principalement : la non-considération du modèle économique en particulier la facturation et la non-prise en compte du temps d'initialisation des ressources.

HybridScale est basé sur une boucle autonomique pour automatiser et optimiser en continue la gestion de capacité des ressources. D'abord, nous avons présenté *BestPolicies* : des politiques de gestion de l'élasticité. Ces politiques permettent de contrôler l'analyse. L'originalité de *BestPolicies* repose principalement sur trois types de politiques : i) les politiques de prix, ii) les politiques de réaction et iii) les politiques de dimensionnement. En particulier, nous proposons la stratégie de terminaison pour harmoniser le dimensionnement automatique avec le modèle économique (facturation à l'heure).

Nous avons proposé dans le chapitre précédent deux solutions à savoir la dégradation de QdS et la dégradation de fonctionnalité pour absorber le temps d'initialisation des ressources. Dans ce chapitre, nous consolidons notre solution avec la prédiction de la charge du travail. Nous proposons une technique de prédiction basée sur les séries chronologiques. Notre technique de prédiction permet de générer les futures charges en paramétrant aisément les intervalles/fenêtres de prédiction.

Enfin, nous avons proposé *RightCapacity* : une méthode de planification de capacité de ressources dirigée par SLA. L'originalité de RightCapacity réside dans la planifica-

tion multi-couches (*cross-layer*). Il prend compte deux niveaux à savoir : l'application (SaaS) et l'infrastructure (IaaS). Au niveau infrastructure, notre méthode de planification est basée sur la théorie des files d'attente. Elle calcule la capacité des ressources optimale. Notre méthode est suffisamment générale pour modéliser n'importe quelle application Web déployée sur le Cloud. Au niveau application, notre méthode de planification, calcule l'architecture logicielle appropriée.

6

Prototype HybridScale et expérimentations

Dans le chapitre précédent, nous avons présenté les éléments de base pour mettre en œuvre un auto-dimensionnement dirigé par SLA. Nous nous intéressons dans ce chapitre à présenter et à expérimenter notre prototype de recherche pour valider nos contributions autour d'*HybridScale*.

Nous commençons par détailler notre prototype *HybridScale* : un framework de dimensionnement automatique. La réutilisabilité des élements d'*HybridScale* comme l'intégration des politiques de gestion de l'élasticité (*BestPolicies*) dans une solution de dimensionnement existante comme Amazon Auto-Scaling est détaillée en annexe B. Ensuite, nous proposons un certain nombre d'expérimentations pour valider notre contribution dans l'environnement IaaS Amazon EC2.

Sommaire

6.1 Prototype *HybridScale* **130**

6.1.1 Boucle de contrôle MAPE-K 130

6.1.2 Cycle de vie du gestionnaire autonome 135

6.1.3 Implémentation 136

6.2 Expérimentations **139**

6.2.1 Protocole d'expérimentation 139

6.2.2 Résultats 141

6.3 Synthèse **148**

6.1 Prototype *HybridScale*

HybridScale est une solution avancée pour gérer l'élasticité du Cloud. Elle est basée sur une boucle de contrôle MAPE-K [Hor01].

La boucle de contrôle a été présentée dans le chapitre précédent. Dans ce qui suit, nous développons, d'abord, les éléments de la boucle de contrôle. Ensuite, nous illustrons le cycle de vie du gestionnaire autonome.

6.1.1 Boucle de contrôle MAPE-K

La Figure 6.1 illustre le méta-modèle de boucle de contrôle MAPE-K. Le gestionnaire autonome exécute une séquence de tâches autonomes (*AutonomicTask*) [Hor01], qui sont déclenchées lors d'un événement donné et en tenant compte de sa base de connaissances (*Knowledge*).

Le gestionnaire autonome observe les informations remontées via des capteurs (*Monitor*). Il utilise ses données et sa connaissance interne pour analyser (*Analyzer*), planifier (*Planner*) et exécuter (*Executor*) des actions pour atteindre un objectif déterminé. Les actions sont exécutées via des actionneurs.

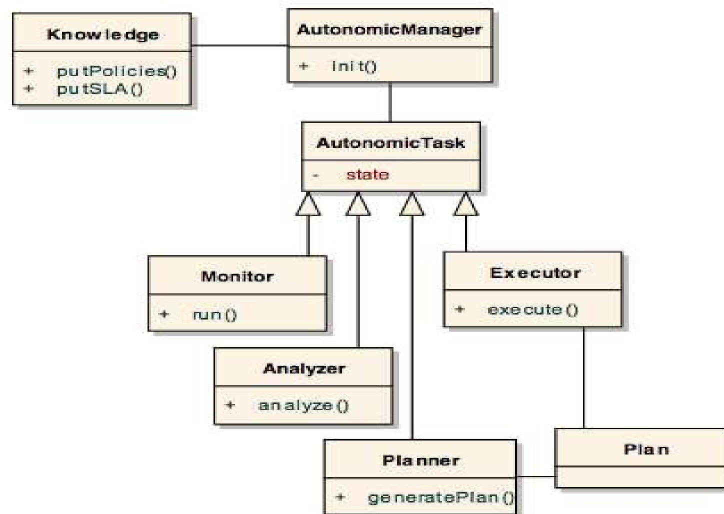


FIGURE 6.1 – Méta-modèle MAPE-K

i) Surveillance - Monitor

Le monitor permet de collecter les informations sur le système (workload, performance, application et ressources) via des sondes. Nous proposons deux modes de collecte : *Pull* et *Push* (voir Figure 6.2).

a) *Pull*

Le mode *Pull* consiste à fournir des mesures en se basant sur des sondes. Nous distinguons plusieurs types de sondes dédiées au Workload (quantité de charge, temps

d'attente), SLA (temps de réponse, disponibilité), Application (mode de chaque service) et Infrastructure (nombre de réplicas pas tier, MPL associé).

Les sondes collectent des mesures selon le mode *Pull* à des intervalles réguliers (par exemple 5 secondes). Ces informations peuvent être agrégées pour fournir une seule valeur (moyenne, minimum, maximum ou f()) sur une fenêtre de temps (par exemple, la moyenne de temps des réponses sur une fenêtre d'une minute). Ces données seront envoyées par le *Monitor* à l'*Analyzer* d'une manière régulière via le gestionnaire autonome. Les valeurs des intervalles et fenêtres dépendent du type d'application (e-commerce, calcul scientifique) et la nature de son workload (aléatoire, cyclique, événementiel).

b) *Push*

Le mode *Push* permet de notifier les événements suivants :

Le cycle de vie d'une instance :

- *instance-activated* informe de la mise en fonctionnement d'une instance. Cette notification permet le retour en mode normal.
- *instance-fullTime* annonce la fin de la période-instance pour chaque instance du système (stratégie de terminaison *UaP*). Elle est envoyée avant la fin de la période pour prendre une décision.

Le cycle de vie de dimensionnement

- *scaling-calmStarted* mentionne le début d'une période de calme.
- *scaling-calmEnded* avertit la fin d'une période de calme.

Le cycle de vie d'une application

- *application-degradationPeriodEnded* indique la fin de l'utilisation du mode dégradé. Cette notification permet le retour en mode normal.

Le cycle de vie de SLA

- *sla-thresholdExceeded* capture un dépassement de seuil d'un SLO. Cette notification déclenche le mode dégradé.

Dans le prototype actuel, pour des raisons de coordination entre événements notifiés et cycle de vie de la boucle autonomique, les notifications seront enregistrées dans une base de données nommée *base de notifications* et lues de façon synchrone par le gestionnaire autonome.

ii) Analyse - Analyze

La Figure 6.3 illustre le méta modèle *Analyzer*. Les informations collectées par le *Monitor* sont envoyées à l'*Analyzer* régulièrement. Ce dernier traite les informations via la méthode *RightCapacity* de diverses manières : proactive et/ou réactive selon les politiques

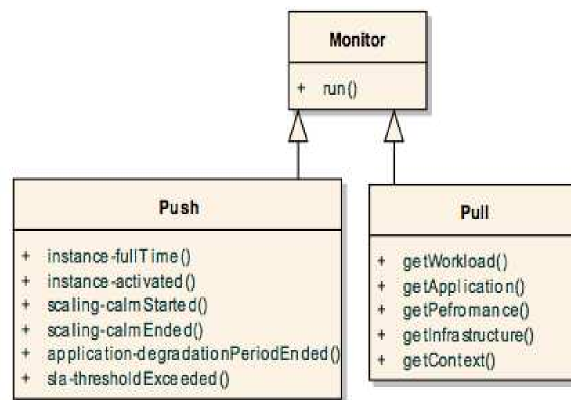


FIGURE 6.2 – Méta-modèle Monitoring

de gestion de l'élasticité. L'originalité de notre *Analyzer* est d'adresser le multi-couches (*cross-layer*). Il prend en compte deux niveaux à savoir : l'application (SaaS) et l'infrastructure (IaaS).

a) Réactive, proactive, hybride

Réactive : L'analyse réactive consiste à prendre des décisions en fonction de la demande courante. La dégradation de fonctionnalité est utilisée conformément à la politique *Application Elasticity* pour absorber les petits pics des charges ainsi que le démarrage des instances.

Proactive : L'analyse proactive consiste à ajuster la capacité en se basant sur la future demande. Notre solution de prédiction, développée précédemment dans le chapitre 5, estime la future charge. En cas d'une erreur de prédiction, il n'y a pas de réaction particulière à faire. La terminaison des instances est dirigée par le contexte courant en corrélation avec la base des notifications (*instance-fullTime*).

Hybride : Cette analyse mélange une réaction proactive et une réaction réactive. Par défaut, l'analyse proactive permet d'anticiper une augmentation possible de capacité d'une manière régulière. L'analyse réactive permet de corriger les erreurs de prédiction et/ou terminer des instances en se basant sur le contexte courant. Elle est activée si l'*Analyzer* lit une notification dans la base des notifications. L'analyse réactive est exécutée avant l'analyse proactive. Cette dernière considère le résultat de l'analyse réactive comme contexte courant.

b) Capacité optimale, architecture appropriée

Capacité optimale : Les politiques (*BestPolicies*) encadre le processus d'analyse. La capacité optimale est le résultat de l'application des politiques de gestion d'élasticité sur la capacité demandée.

Architecture appropriée : L'architecture appropriée est calculée en fonction des événements liés à la performance ou aux actions de dimensionnement. En cas de dépassement d'un seuil d'un objectif ou une action d'ajout des ressources, l'architecture appropriée passe en mode dégradé.

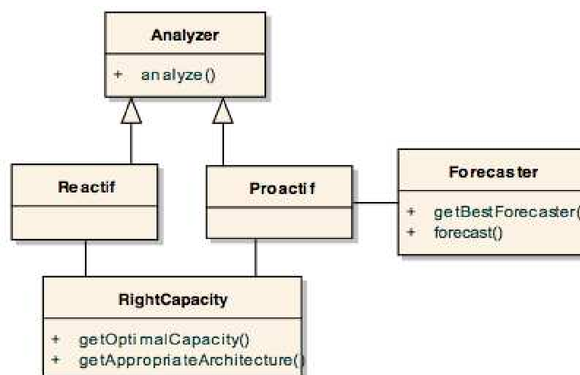


FIGURE 6.3 – Méta-modèle Analyze

iii) Planification - Plan

Après avoir étudier l'état (courant et/ou futur) du système via l'*Analyzer*, le *Planner* prépare le plan d'actions. Nous distinguons entre deux types d'actions immédiates et différées. Le plan est le résultat de zéro ou plusieurs actions (ajustements ou notifications) immédiates et/ou différées (voir Figure 6.4).

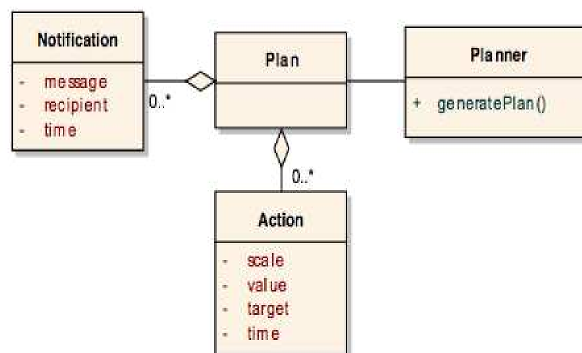


FIGURE 6.4 – Méta-modèle Plan

a) Actions

Immédiates : Les actions immédiates sont calculées en fonction de la différence entre la capacité courante et la capacité optimale. Ces actions sont exécutées directement :

- ajout de capacité (*scale up* et/ou *scale out*),
- calibrage du mode de l'application (*scale app*),
- ajustement du contrôle d'admission (*scale mpl*).

Différées : La stratégie de terminaison *UaP* gouverne la réduction de capacité. Une instance sera supprimée si elle a consommé plus de 95% de son temps-instance. Si la capacité demandée est inférieure à la capacité optimale, une action (*scale down* et/ou *scale in*) est différée. Lors des prochains cycles, l'*Analyzer* permet de forcer la réduction ou d'annuler l'action en se basant sur le contexte courant et sur la base de notifications

(*instance-fullTime*).

b) Notifications

La politique *Infrastructure Elasticity* contrôle l'augmentation de capacité. Nous proposons d'envoyer une notification (email) au fournisseur de service (SaaS) si la capacité demandée est supérieure à la capacité définie par la politique *Infrastructure Elasticity*. Cette notification peut être immédiate ou différée (fin de la journée, fin de la semaine).

iv) Exécution - Execute

L'*Executor* consiste à exécuter le plan défini par le *Planner* (voir Figure 6.5).

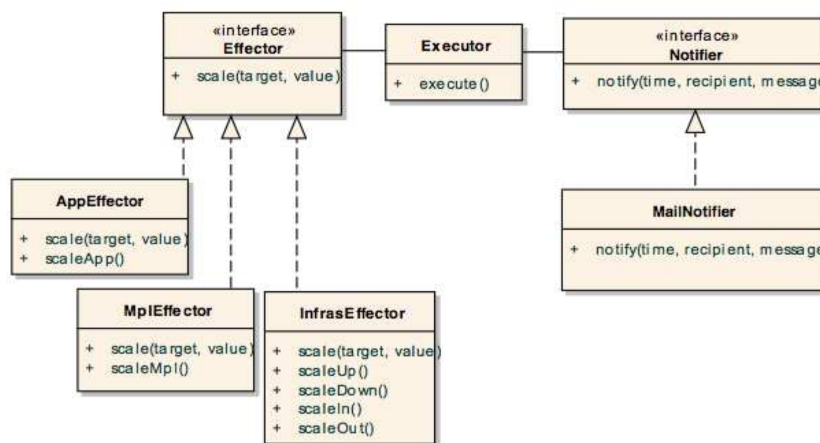


FIGURE 6.5 – Méta-modèle Execute

a) Actions

Nous énumérons trois actionneurs possibles : infraEffector, appEffector et mplEffector :

Infrastructure : *infraEffector* exécute un ensemble des actions sur les ressources à savoir principalement : *scale in*, *scale out*, *scale up* et *scale down*.

Application : *appEffector* permet d'interagir avec l'application comme une boîte blanche. Il accomplit l'action *scale app* qui ajuste le mode de fonctionnement de l'application.

Contrôle d'admission : *mplEffector* permet de piloter le contrôle d'admission. Il consiste à ajuster le MPL de chaque instance via l'action *scale mpl*.

b) Notifications

En plus des actions, l'*Executor* permet d'envoyer des notifications au fournisseur de service afin d'être alerté lorsque la capacité demandée est supérieure à la capacité maximum définie par la politique *Infrastructure Elasticity*.

v) Connaissances - Knowledge

Dans ce travail, les connaissances consistent à partager l'ensemble des politiques (*Best-Policies*), leurs paramètres associés, variables publiques comme la durée de cycle MAPE-K ainsi que la base des notifications.

6.1.2 Cycle de vie du gestionnaire autonome

Cette section illustre le cycle de vie du gestionnaire autonome. Dans ce qui suit, nous détaillons l'initialisation de la boucle, un exemple de cycle réactif/proactif et un exemple de notification.

La Figure 6.6 illustre l'initialisation de gestionnaire autonome (*Autonomic Manager*) : mettre en place les politiques, récupérer le SLA, initialiser les variables publiques (exemple :timer), lancer le monitoring et démarrer la gestion réactive et/ou proactive.

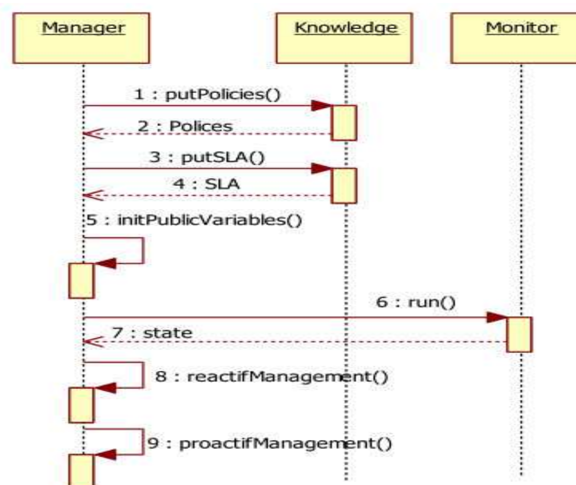


FIGURE 6.6 – Initialisation de la boucle

La Figure 6.7 illustre un cycle de la boucle MAPE-K. Chaque cycle, le gestionnaire autonome collecte les informations du contexte via le *Monitor*. Ces informations sont envoyées à l'*Analyzer* pour les traiter. L'*Analyzer* fournit la capacité optimale en se basant sur la demande courante (réaction réactive) ou future demande (réaction proactive). Le *Planner* utilise cette capacité pour produire le plan de redimensionnement. Ce dernier sera exécuté via *Executor*. Les variables publiques sont mises à jour après l'exécution à savoir principalement les variables liées à la période de calme. Le cycle suivant sera lancé selon la durée du cycle (par exemple une minute).

La Figure 6.8 illustre le comportement du *Autonomic Manager* en cas de dépassement de seuil d'un objectif. Nous considérons le seuil d'un objectif (*threshold*) comme un seuil de menace [BEM+10]. Une fois dépassé, une analyse réactive est déclenchée pour absorber les futures violations. Si la politique *Application Elasticity* est activée (*state=active*), une dégradation de fonctionnalité est programmée. L'application bascule vers le mode normal après la période de dégradation. En cas où la limite d'utilisation (*fréquence*) de l'élasticité applicative est atteinte, la dégradation de fonctionnalité est accompagnée par un ajustement de la capacité en fonction du contexte courant. L'application reste

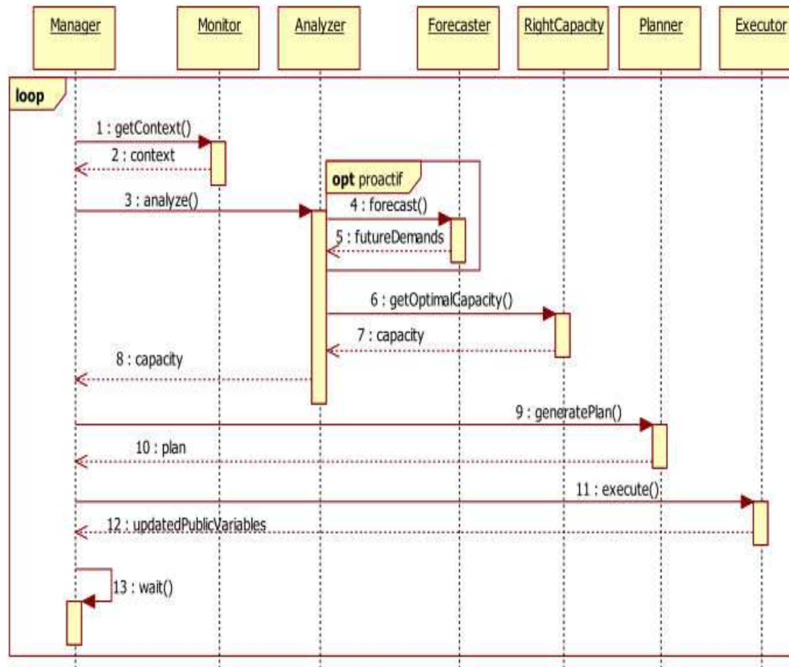


FIGURE 6.7 – Cycle MAPE-K réactif/proactif

en mode dégradé jusqu'à l'activation des instances ajoutées.

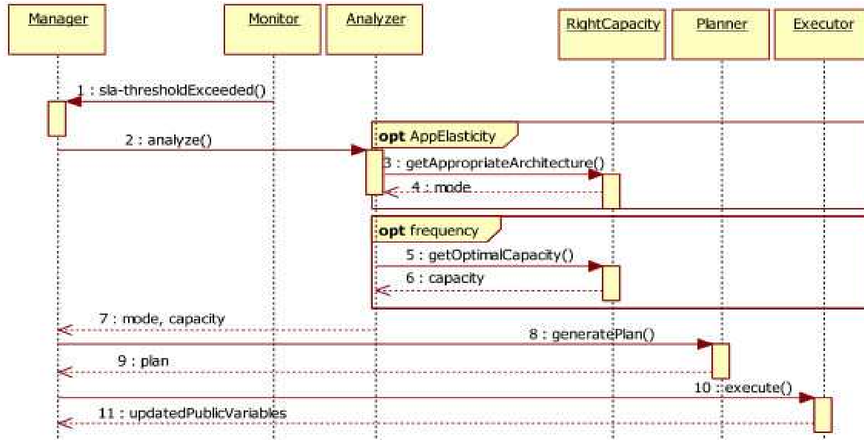


FIGURE 6.8 – Notification d'un dépassement de seuil

Le diagramme d'activités 6.9 résume les transitions possibles pour l'application.

6.1.3 Implémentation

Le prototype *HybridScale* est organisé en différents composants, dont les principaux sont : Monitor, Analyzer, Planner, Executor et Forecaster. Chaque composant est indépendant des autres. Ceci permet de tester, remplacer ou faire évoluer facilement n'importe quel composant. L'ensemble du prototype a nécessité l'écriture des lignes de codes Java et de script (~10000 de lignes de code).

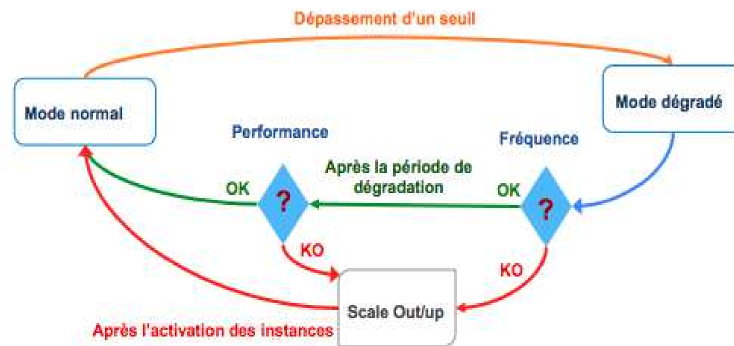


FIGURE 6.9 – Cycle de vie de l'application

i) Monitor

Afin de prédire la performance et la disponibilité de service SaaS, notre modèle analytique nécessite plusieurs paramètres en entrée. En pratique, les paramètres de la charge de travail (Z_c, $S_{c,m}$, $V_{c,m}$, $D_{c,m}$ $(c = 1, ..., C, m = 1, ..., M)$) peuvent être estimés en surveillant la charge de travail de l'application comme mentionné dans [UPS+07] par exemple.

Le temps de réponse et la disponibilité sont calculés sur le premier tier (Loadbalancer dans notre cas). Pour les notifications, nous avons utilisé un simple modèle abonnement-notification (patron observer).

ii) Analyzer

L'*Analyzer* est fondé sur la méthode *RightCapacity*. La version actuelle ne supporte que l'implémentation de MVA pour des réseaux fermés. Cependant, il est possible d'enrichir l'*Analyzer* avec d'autres extensions de MVA [RL80] (comme réseaux ouverts ou mixtes) via l'interface *AnalyticalModel* ou d'autres implémentations (comme l'apprentissage par renforcement [SB98]) via l'interface *InfraPlanningModel*. Au niveau application, *RightCapacity* fournit une implémentation simple de l'élasticité architecturale (dégradation de fonctionnalité). Des extensions sont possibles grâce à l'interface *AppPlanningModel*.

iii) Planner

Ce composant prépare le nouveau plan en décrivant la capacité optimale via une syntaxe abstraite. Nous proposons un langage dédié [vDKV00](Domain specific language ou DSL) pour décrire un plan. Ce langage aide, également, le fournisseur de service SaaS à préparer des plans (manuellement) pour des raisons de tests ou évaluation.

Nous présentons, ici, la grammaire du DSL suivi d'un exemple qui ajoute 5 instances au tier *Compute* et calibre le MPL à 100 :

```
1    HybridScalePlan ::= PLAN : planName { (action)+ }
2    action ::= SCALE: scale target value | NOTIFY: msg time ;
3    scale ::= SCALE_OUT | SCALE_IN | SCALE_UP | SCALE_DOWN | SCALE_MPL | SCALE_APP
4    planName, target, msg : string
5    value : int
6    time : dateTime
```

```
1        PLAN : Example {
2                SCALE: scale_out Compute 5;
3                SCALE: scale_mpl Compute 100;
4        }
```

Le XML suivant est généré automatiquement à partir de l'exemple précédent :

```
1 <hybridScale:plan id="Example" xmlns:hybridScale="http://www.inria.fr/hybridscalemodel">
2   <hybridScale:actions>
3     <hybridScale:action scale="scale_out" target="Compute" value ="5"/>
4     <hybridScale:action scale="scale_mpl" target="Compute" value ="100"/>
5   </hybridScale:actions>
6   <hybridScale:notifications/>
7 </hybridScale:plan>
```

iv) Executor

L'*Executor* traduit chaque ligne de plan par une ligne de commande. Nous proposons trois types d'actionneurs :

infraEffector : Afin d'interagir avec n'importe quelle infrastructure Cloud (publique ou privée), nous introduisons la notion du proxy qui permet d'implémenter et unifier les différentes APIs proposées. Dans un premier temps, nous avons développé un proxy pour Amazon EC2. *HybridScale* supporte principalement l'élasticité horizontale. La ligne de commande 1 permet d'ajouter le nombre d'instances *value* au tier *Compute*. Alors que la ligne de commande 2 permet de terminer la liste des instances *instancesToTerminate* (de tier *Compute*) calculés par *RightCapacity* :

```
1 as-set-desired-capacity Compute --desired-capacity value
2 as-terminate-instance-in-auto-scaling-group instancesToTerminate --decrement-desired-capacity -f
```

appEffector : Cet actionneur permet d'ajuster le mode de service. Il consiste à basculer entre le mode normal et le mode dégradé. La version actuelle permet de changer les paramètres des services et/ou activer/désactiver des services.

mplEffector : Comme *infraEffector*, cet actionneur est implémenté via la notion proxy. L'idée est de s'adapter avec n'importe quel type de tier et n'importe quel type de serveur. Il permet d'ajuster le MPL.

Les lignes suivantes présentent des exemples d'ajustement de MPL. La ligne 1 permet d'ajuster un serveur MySQL (tier de stockage). Les lignes (2-5) sont liées à un tier de calcul : lignes (2-3) un exemple d'un serveur Tomcat et lignes (4-5) un exemple de serveur Apache HTTP.

```
1 ssh storageInstance "mysql -u root -proot forecasting -e 'set global max_connections = mpl_storage;'"
2 ssh computeInstance "sed -i "s/maxThreads=\"[0-9]*\"/maxThreads=\"mpl_compute\"/" /etc/tomcat7/server.xml"
3 ssh computeInstance "service tomcat7 restart"
4 ssh computeInstance "sed -i 's/ServerLimit [0-9]*/ServerLimit mpl_compute/' /etc/apache2 apache2.conf"
5 ssh computeInstance "service apache restart"
```

vi) Knowledge

Le composant de connaissance permet de manipuler les politiques de gestion de l'élasticité, les varaibales publiques ainsi que la base des notifications. L'ensemble de ces

connaissances peut être enregistré dans des fichiers XML ou une base de données MySQL.

L'XML suivant présente un exemple des politiques de gestion de l'élasticité : la dégradation de fonctionnalité est déclenchée pour une minute (time=1) dans la limite de 3 utilisations successives dans une fenêtre de 10 minutes (window =10). En cas de *scale out*, la stratégie de terminaison *UaP* est activée (policy=uap). La période de calme est égale à une minute (calm=1). La capacité maximum est égale à 10 instances pour le tier *Compute* et 2 pour le tier *Balancer*.

```
1  <hybridScale:bestPolicies id="bp1" xmlns:hybridScale="http://www.inria.fr/hybridscalemodel">
2    <hybridScale:scalingPolicies>
3      <hybridScale:appEasticity time="1" frequency="3" window ="10"/>
4      <hybridScale:infraEasticity>
5        <hybridScale:target name="Balancer">
6          <hybridScale:horizontal min="1" max="2">
7            <hybridScale:out policy="uap" calm ="1"/>
8            <hybridScale:in calm ="1"/>
9          </hybridScale:horizontal>
10       </hybridScale:target>
11       <hybridScale:target name="Compute">
12         <hybridScale:horizontal min="1" max="10">
13           <hybridScale:out policy="uap" calm ="1"/>
14           <hybridScale:in calm ="1"/>
15         </hybridScale:horizontal>
16       </hybridScale:target>
17     </hybridScale:infraEasticity>
18   </hybridScale:scalingPolicies>
19 </hybridScale:bestPolicies>
```

vii) Forecaster

Ce composant est basé sur la librairie java OpenForecast 0.5 [1]. Cette librairie est open-source. Le *Forecaster* est facilement extensible. Afin d'enrichir l'ensemble des méthodes de prédiction ou les modèles de précision, il suffit d'étendre respectivement l'interface *ForecastingMethod* ou l'interface *ForecastingAccuracy*.

6.2 Expérimentations

Dans cette section, nous présentons les évaluations de nos contributions. D'abord, nous développons le protocole expérimental et les résultats des expérimentations. A chaque fois, nous proposons une discussion.

6.2.1 Protocole d'expérimentation

Nous présentons dans ce qui suit l'environnement et les scénarios de l'expérimentation.

i) Environnement

Injecteur de charge : Nous utilisons Apache JMeter pour imiter des actions des utilisateurs réels en simulant la charge.

[1] http ://www.stevengould.org/software/openforecast/index.shtml

Application synthétique : Les expérimentations réalisées ont été conduites avec des bancs d'essai synthétiques simulant un service de cryptographie. Le SaaS fournit ce service en deux modes : normal (par exemple chiffrement symétrique 3DES) et dégradé (par exemple chiffrement symétrique DES). Le Triple DES (3DES) est un algorithme de chiffrement symétrique par bloc, enchaînant 3 applications successives de l'algorithme DES sur le même bloc de données de 64 bits, avec 2 ou 3 clés DES différentes. Nous utilisons deux variantes de services S1 (0.004 $/rqt) avec le mode normal seulement et S2 (0.0045 $/rqt) qui intègre les deux modes normal et dégradé. Le Tableau 6.1 formalise les SLAs associés. Le temps de réponse désigne la moyenne des temps de réponse sur un intervalle d'une minute. La disponibilité est calculée comme le rapport entre le total des requêtes envoyées par un client c et les requêtes rejetés dans un intervalle d'une minute. L'utilisation du mode dégradé est le pourcentage des requêtes dégradées sur une minute. Toutes les métriques sont évaluées sur une fenêtre de 10 mn.

TABLE 6.1 – SLA niveau SaaS (SLA_S)

service	métrique	oper.	valeur	fuzz.	% of fuzz.	conf.	pénalité
S1/S2	Rt	≤	0,1s	0,02 s	11,11%	90%	0,003$/rqt
	Av	≥	99%	2%			0,0020$/rqt
S2	Mu(degradé)	≤	30%	5%			0,0015$/rqt

Comme illustrée dans la Figure 6.10, l'application est basée sur une architecture 3-tiers ($M = 3$) distribuée et répliquée dans le Cloud : i) un répartiteur de charge qui est chargé de distribuer le trafic entrant sur plusieurs serveurs d'applications, ii) un serveur d'application qui implémente des fonctionnalités de logique métier, et iii) une base de données.

Infrastructure : Les expérimentations présentées dans cette section ont été conduites sur des instances d'Amazon EC2. Le Tableau 6.2 illustre le SLA associé à ces ressources. La disponibilité est calculée en fonction du pourcentage de temps utilisable mensuel sur des intervalles de 10 minutes. La pénalité (crédit de service) est égale à un pourcentage de la facture du client.

TABLE 6.2 – SLA niveau IaaS (SLA_R)

service	métrique	oper.	valeur	fuzz.	% of fuzz.	conf.	prix	pénalité
Small	Av	≥	99,95%	0	0	100	0,046$ /h	10% si 99,95 < Av ≤ 99% 30% si Av < 99%

Les instances fonctionnent sous un environnement Ubuntu 12.04.2 LTS avec les intergiciels suivants : le répartiteur de charge Amazon Load Balancer, le serveur d'applications Apache en version 2.2.22 et le serveur de bases de données MySQL en version 5.5.32.

Base de connaissances : Les expérimentations réalisées ont été conduites avec les politiques résumées dans le Tableau 6.3. Les valeurs *max* sont estimées en utilisant notre modèle analytique. La durée du cycle MAPE-K est égale à une minute.

Forecaster : Le temps d'initialisation est déterminé suite à une campagne de benchmarking. Nous considérons un intervalle de prédiction (futur) d'une minute.

TABLE 6.3 – Scaling Policies

AppElasticity	time=1, frequence=3, window=10 ;
InfraEasticity	target=Balancer, type=horizontal, min=1, max= 2 ; calm out=1, in=1, UaP=true ;
	target=Compute, type=horizontal, min=1, max=10 ; calm out=1, in=1, UaP=true ;
	target=Storage, type=vertical, min=1, max= 1 ; calm up=1 down=1 ;

ii) Scénario d'évaluation

Ce scénario vise à démontrer les apports de *HybridScale* pour résoudre le compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients. Nous évaluons la méthode de planification *RightCapacity*, la politique de dégradation de fonctionnalité et la stratégie de terminaison *UaP*.

Nous proposons *HybridScaleTbR* une implémentation de *HybridScale* où nous remplaçons la méthode *RightCapacity* par des règles à base des seuils. Une telle impémentation définit la capacité selon des règles prédéfinies. Nous choisissons $+1/-1$ comme règles en accordant une attention sur le coût des ressources. Nous désactivons la politique d'élasticité applicative pour *HybridScaleTbR*. Les deux implémentations *HybridScale* et *HybridScaleTbR* sont combinées avec les deux stratégies de terminaison *UaP* et "plus récent" (*Newest*). Cette dernière consiste à terminer immédiatement l'instance la plus récente en cas de *scale in*. Le Tableau 6.4 résume les solutions testées.

TABLE 6.4 – Expérimentations

solution	planification de capacité	stratégie de terminaison	elasticité applicative
HybridScale (UaP)	RightCapacity	UaP	Oui
HybridScale (Newest)	RightCapacity	plus récent	Oui
HybridScaleTbR (+1/-1 UaP)	règles à base des seuils (TbR)	UaP	Non
HybridScaleTbR (+1/-1 Newest)	règles à base des seuils (TbR)	plus récent	Non

Nous effectuons respectivement une évaluation avec une politique de réaction réactive et une politique de réaction hybride.

6.2.2 Résultats

Nous présentons maintenant les résultats principaux obtenus à l'issue de différentes évaluations réalisées sur l'environnement décrit précédemment. L'ensemble des résultats justifie les apports de nos contributions.

Dans ce qui suit, nous développons quelques workload qui illustrent clairement les apports d'*HybridScale*. Dans un premier temps, nous détaillons les avantages de la réaction réactive face à une charge imprévisible puis une charge *On and Off*. Ensuite, nous présentons l'intérêt de la réaction hybride face à une charge prévisible (suit une tendance).

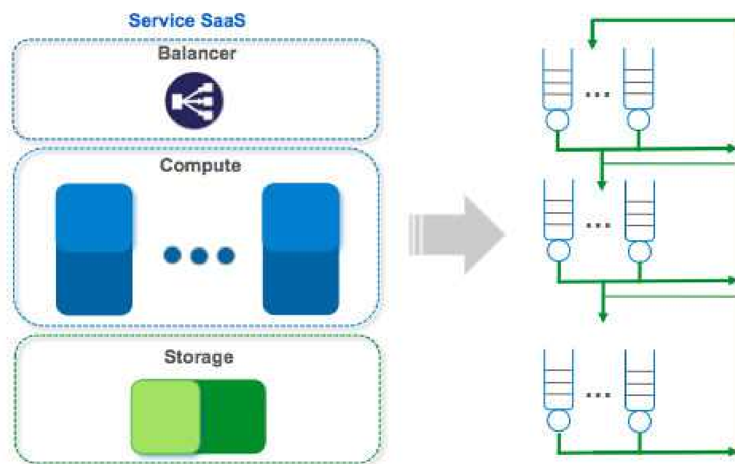


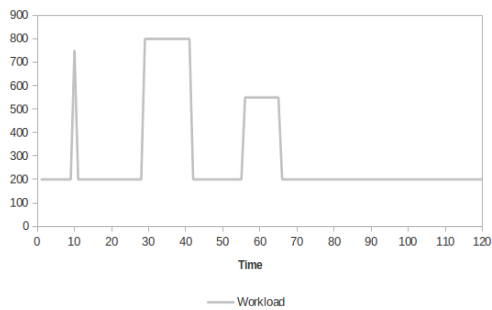
FIGURE 6.10 – Environnement d'évaluation

i) Réaction réactive

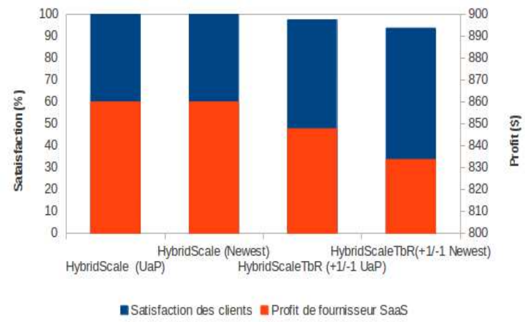
Les figures 6.11 et 6.12 montrent l'intérêt des éléments d'*HybridScale* (en particulier la méthode de planification de capacité, la stratégie de terminaison *UaP* et la dégradation de fonctionnalité) dans l'objectif d'assurer une capacité toujours optimale, et notamment en maintenant la stabilité du système.

Workload imprévisible : La Figure 6.11(a) présente l'allure de la charge de travail que nous appliquons au service. L'expérimentation débute avec une charge modérée correspondant à 200 clients. La charge de travail monte brutalement i) à 750 clients à la 10ème mn, ii) à 800 clients à la 29ème mn et iii) à 550 clients à la 56ème mn en redescendant à 200 clients après chaque pic de charge. Les figures 6.11(c) et 6.11(d) illustrent le comportement du service SaaS en réponse à ces pics de charge en utilisant les implémentations résumées dans le Tableau 6.4. Le comportement du service est caractérisé sur la Figure 6.11(c) par le nombre d'instance-heures à payer et par le pourcentage de dépassement des seuils (temps de réponse, disponibilité, mode d'usage) sur la Figure 6.11(d). Cette Figure illustre également le pourcentage des requêtes dégradé (QdS) et inadéquates absorbées par les propriétés de CSLA. Nous présentons, aussi, les transitions entre le mode normal (1) et le mode dégradé (0) pour *HybridScale (UaP)* et *HybridScale (Newest)* sur les figures 6.11(e) et 6.11(f). La Figure 6.11(b) illustre la satisfaction des clients *vs* le profit de fournisseur SaaS. La satisfaction des clients fait référence à la conformité aux objectifs du SLA. Le profit est calculé comme détaillé dans le chapitre CSLA (Modèle économique 4.1.3). Le coût de service dépend principalement du coût des ressources et des pénalités.

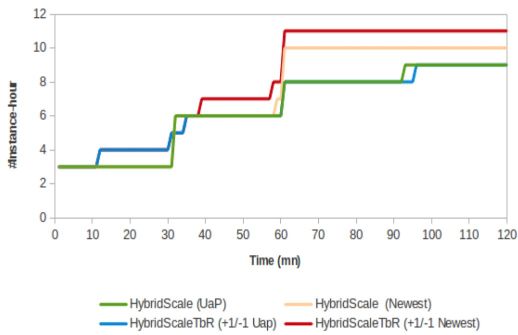
La politique d'élasticité applicative (voir figures 6.11(e) et 6.11(f)) a amorti l'ajout des ressources lors de pic de charge à t=10*mn*. De plus, la stratégie *UaP* fonctionne mieux pour réduire le nombre d'instance-heure à payer à 9 seulement contre 11 pour la stratégie "plus récent" (voir Figure 6.11(c)). La méthode *RightCapacity* produit le nombre minimum de dépassement des seuils par rapport à la méthode règles à base des seuils *TbR*. Dans la Figure 6.11(c) à $t = 29mn$, grâce à la méthode *RightCapacity* environ 3*mn* seulement sont nécessaires pour répondre aux exigences de SLA (performance, disponibilité et mode usage), tandis que la méthode *TbR* demande au moins 6*mn* (pour *HybridScale +1/-1 UaP*). La méthode *RightCapacity* garantit le retour à la stabilité dans



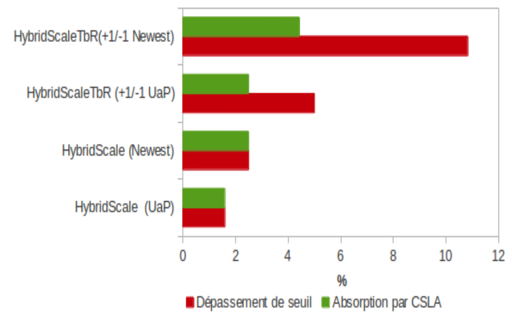
(a) Workload



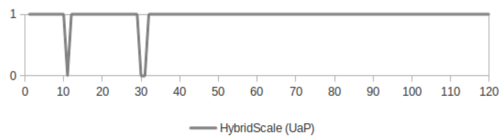
(b) Satisfaction des clients vs Profit de SaaS



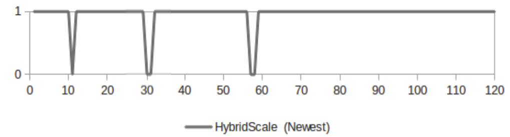
(c) Coût des instances



(d) Dépassement de seuils



(e) Dégradation de fonctionnalité



(f) Dégradation de fonctionnalité

FIGURE 6.11 – Un workload imprévisible

un minimum de temps. De plus, elle ajuste le contrôle d'admission automatiquement selon la demande et donc permet d'absorber des dépassements des seuils sans augmenter le coût.

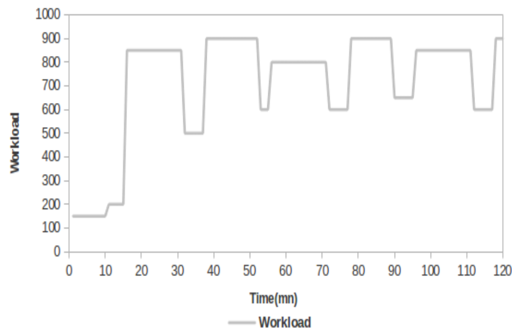
Le pourcentage de dépassement des seuils (voir Figure 6.11(d)) est le rapport du nombre des intervalles dégradé (QdS) + nombre des intervalles inadéquats/ nombre total des intervalles. Nous distinguons 120 intervalles (total) d'une minute durant notre expérimentation. L'évaluation des objectifs de niveau des services est effectuée sur des fenêtres de 10mn. Un intervalle (d'une minute) est classé comme i) dégradé si ou moins un des objectifs (temps de réponse, disponibilité ou mode usage) utilise la marge d'erreur associée, ii) inadéquat si ou moins un des objectifs (SLOs) dépasse la marge d'erreur. Le SLA_S est calibré de telle sorte à accepter un intervalle inadéquat et un intervalle dégradé chaque 10mn, sinon une pénalité sera déclenchée. La Figure 6.11(d) indiquent que *HybridScale UaP* produit le minimum de dépassement des seuils : moins de 1%.

En utilisant *HybridScaleTbR +1/-1 Newest*, le profit du fournisseur SaaS est le moins important avec une satisfaction des clients de 93,6% (voir Figure 6.11(b)). En effet, la méthode de planification *TbR* et la stratégie de terminaison *Newest* causent l'instabilité de système. Une telle instabilité produit plus de dépassement des seuils (voir Figure 6.11(d)). De plus, la terminaison immédiate avec *Newest* provoque plus d'instance-heure à payer. Le profit de fournisseur SaaS est presque le même en utilisant *HybridScale UaP* et *HybridScaleTbR Newest* avec une satisfaction de 100%. Néanmoins, *HybridScale UaP* présente la meilleure QdS perçue par les clients de service : il y a moins de requêtes avec une dégradation de fonctionnalité (3mn vs 5mn sur 120mn) (voir figures 6.11(e) et 6.11(f)). Le profit de fournisseur SaaS en utilisant *HybridScaleTbR +1/-1 UaP* est légèrement moins important que *HybridScale UaP* et *HybridScaleTbR Newest* mais avec une satisfaction des clients plus grande que *HybridScaleTbR Newest*. Ce résultat montre l'avantage de la stratégie de terminaison *UaP*.

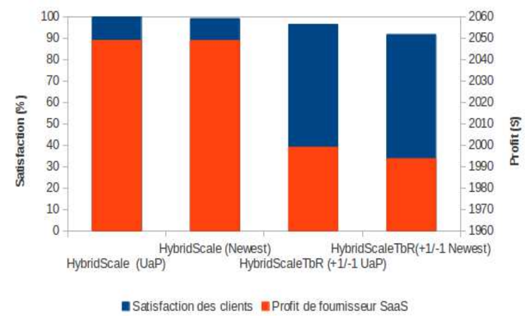
Workload *On and Off* : La Figure 6.12(a) présente un autre type d'allure de charge de travail que nous appliquons au service. La charge de travail oscille entre 500 et 900 clients. Les résultats sont présentés de la même manière que pour le premier type de charge de travail.

Face à une charge de type *On and Off* (voir Figure 6.12), *HybridScale* atteint le coût d'économie maximum et le minimum de nombre de dépassement. La Figure 6.12(c) montre que lors de la variation de la demande, la méthode *RightCapacity* minimise le coût à 12 $instances - heure$ seulement par rapport à 17 $instances - heure$ pour la méthode *TbR*. La Figure 6.12(d) montre que la stratégie *UaP* génère moins de dépassement des seuils que la stratégie "Newest". En fait, la stratégie *UaP* permet d'éviter les oscillations du système et, implicitement, les dépassement des seuils. La Figure 6.12(d) illustre le pourcentage de dépassement des seuils qui est moins de 1% pour *HybridScale UaP*. Les figures 6.12(e) et 6.12(f) illustrent l'élasticité applicative.

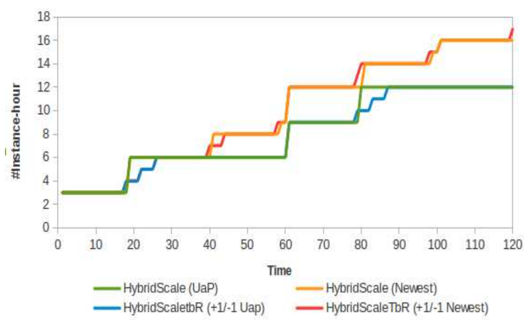
Comme pour le workload imprévisible, le profit de fournisseur SaaS en utilisant *HybridScaleTbR +1/-1 Newest* est le moins important. La satisfaction des clients est de 91,67% (voir Figure 6.12(b)). Nous remarquons que la stratégie *UaP* permet de minimiser l'utilisation de mode dégradé : 2mn sur 120mn alors que le mode dégradé est utilisé



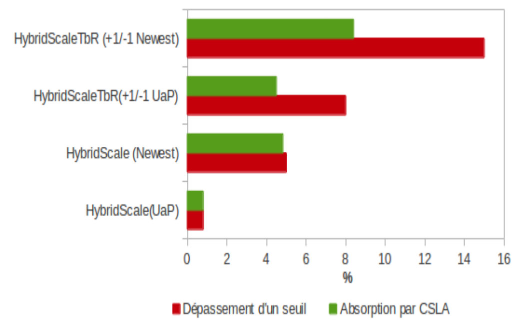
(a) Workload



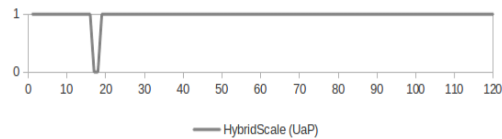
(b) Satisfaction des clients vs Profit de SaaS



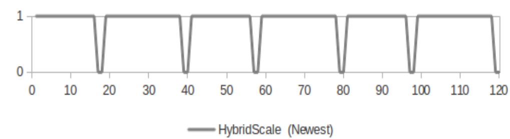
(c) Coût des instances



(d) Dépassement de seuils



(e) Dégradation de fonctionnalité



(f) Dégradation de fonctionnalité

FIGURE 6.12 – Un workload *On and Off*

12mn sur 120mn avec la stratégie "Newest" (voir figures 6.12(e) et 6.12(f)). L'objectif est d'assurer la disponibilité/la performance de service même avec une dégradation de fonctionnalité. Le profit de fournisseur SaaS en utilisant *HybridScale UaP* est le plus important avec 100% de satisfaction des clients.

Discussion

La réaction réactive répond bien au compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients avec une charge imprévisible ou une charge *On and Off*. La dégradation de fonctionnalité permet d'absorber les petits pics de charge. La stratégie de terminaison *UaP* permet d'éviter les oscillations du système en utilisant pleinement les ressources. La méthode de planification *RightCapacity* assure le retour à la stabilité le plus vite possible en calculant la capacité optimale. La méthode *RightCapacity* combinée avec la stratégie de terminaison *UaP* et la dégradation de fonctionnalité permet de réduire le coût en gardant le dépassement des seuils toujours inférieur à 1%.

Notre modèle est basé sur la théorie de files d'attente. Ce modèle est suffisamment général pour modéliser n'importe quelle application Web déployée sur le Cloud. Les résultats montrent que notre modèle produit fidèlement (avec succès) la performance et la disponibilité de service SaaS.

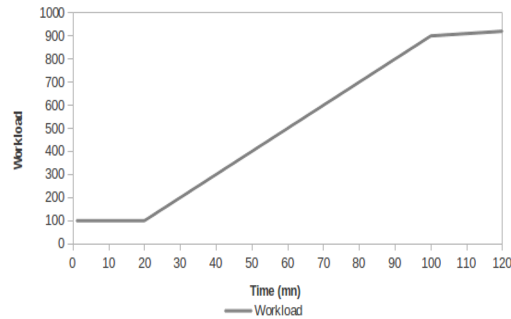
HybridScale peut être adaptée à n'importe quel type de charge pour gérer le compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients. Même face à une charge de travail imprévisible, notre solution réduit le nombre d'instance-heure à payer tout en minimisant les dépassements des seuils. Elle permet également d'avoir la meilleure QdS perçue par les clients de service. Pour une variété de workload, les résultats d'*HybridScale* ne sont jamais pires qu'une solution basée sur des règles.

ii) Réaction hybride

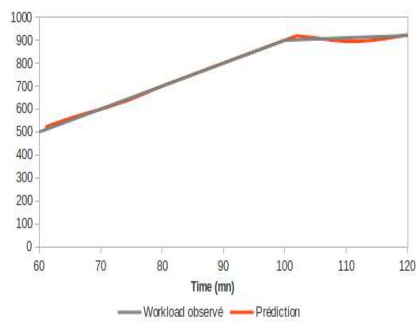
La Figure 6.13 montre l'intérêt de la prédiction dans l'objectif d'assurer la capacité toujours optimale face à une charge qui suit une tendance. Notre évaluation consiste à comparer un ajustement réactif à un ajustement réactif-proactif.

Workload prévisible : La Figure 6.13(a) présente l'allure de la charge de travail que nous appliquons au service. L'expérimentation débute avec une charge modérée correspondant à 100 clients. La charge de travail monte avec une tendance entre la 20 ^{ème} mn et la 100ème mn. La durée de l'expérimentation est égale à 120mn. Un historique d'au moins 60 observations est nécessaire pour la prédiction, c'est pourquoi nous utilisons les premières 60 mn comme fenêtre de prédiction. Les résultats présentés sont liés à la deuxième heure de l'expérimentation (61mn - 120mn).

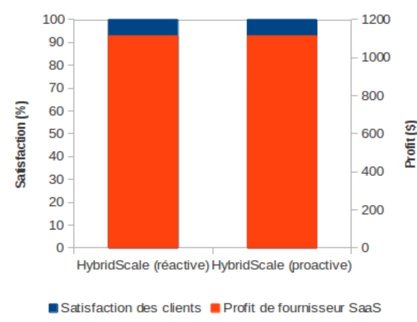
La Figure 6.13(b) illustre le comportement de notre *Forecaster* dans l'intervalle (61mn - 120mn). Les valeurs calculées par notre technique de prédiction étaient exactes avec une erreur moyenne de 0,35% (voir Figure 6.13(b)). La Figure 6.13(d) illustre le comportement de *HybridScale* avec deux types de réaction : réactive et proactive. Le premier type est déja illustré dans la section précédente. La réactive proactive consiste à utiliser le résultat de prédiction pour anticiper l'augmentation de la capacité. Notre intervalle de prédiction est égale à une minute (la valeur moyenne d'initialisation des ressources).



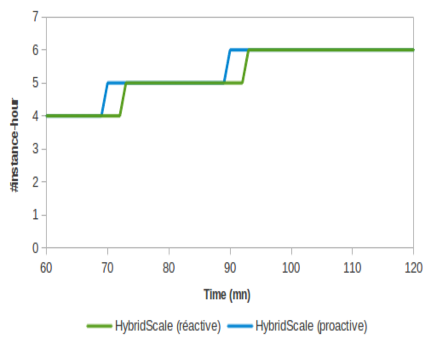
(a) Workload



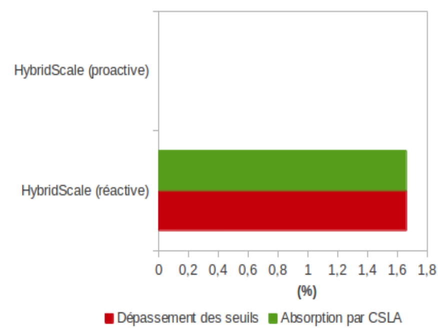
(b) Prédiction du futur workload



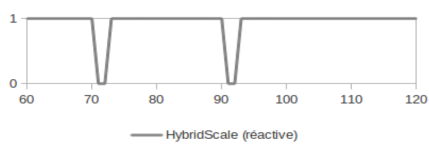
(c) Satisfaction des clients vs Profit de SaaS



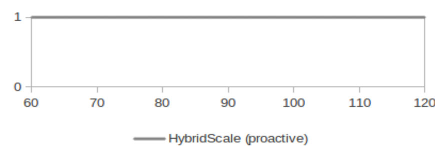
(d) Coût des instances



(e) Dépassement de seuils



(f) Dégradation de fonctionnalité



(g) Dégradation de fonctionnalité

FIGURE 6.13 – Un workload prévisible

La méthode de planification *RightCapacity* calcule la future capacité à t+1 c'est pourquoi la réaction proactive augmente la capacité avant la capacité réactive (voir Figure 6.13(d)). Cette anticipation permet d'éviter les dépassements des seuils. Alors que la réaction réactive ajoute des ressources suite à un dépassement et après avoir tenté l'utilisation de la mode dégradé pour une minute. Avec la réaction proactive, il y a zéro dépassement comme illustré dans la Figure 6.13(e).

Le profit de fournisseur SaaS est le même en utilisant *HybridScale UaP* avec une réaction réactive ou une réaction proactive. La satisfaction des clients est de 100%. Cependant, *HybridScale UaP* avec une réaction réactive produit des dépassements des seuils. De plus, avec ce type de réaction, la dégradation de fonctionnalité est utilisée 4mn pendant 60mn. La réaction hybride permet, donc, de fournir la meilleure QdS perçue par les clients.

Discussion

Nous avons utilisé une prédiction à court terme pour absorber le démarrage des ressources. Cette prédiction a un impact sur le nombre de dépassement. Afin de réduire également le nombre des ressources, une prédiction à long terme serait nécessaire.

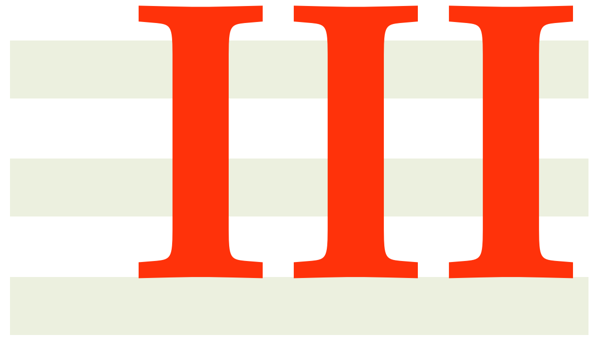
Avec une charge de travail qui suit une tendance, la réaction hybride gère bien le compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients. Elle produit la meilleure QdS perçue par les clients.

6.3 Synthèse

Ce chapitre présente notre prototype de recherche et l'évaluation de nos contributions. Nous avons proposé différentes alternatives afin de remédier aux limitations des approches existantes. La stratégie *UaP* consiste à harmoniser le dimensionnement automatique avec le modèle économique. La dégradation de fonctionnalité et la prédiction sont deux manières pour tenir compte du temps d'initialisation des ressources.

Notre contribution principale est le framework *HybridScale*. L'originalité de *HybridScale* est l'implémentation –pour la première fois– de l'élasticité en triple hybride : gestion réactive-proactive, dimensionnement vertical-horizontal et multi-couches (application-infrastructure). L'ensemble des éléments d'*HybridScale* que nous avons implémentés propose un haut niveau de réutilisabilité et d'extensibilité. Nous avons développé le prototype *AAS++* (annexe B) pour démontrer que nos politiques de gestion de l'élasticité, par exemple, peuvent être intégrées (réutilisées) sur des solutions existantes comme Amazon Auto Scaling. De plus, la conception de framework ouvre la voie à des extensions.

Les différents résultats présentés démontrent le gain significatif apporté par nos contributions. Le choix de la politique de réaction reste une question clé pour bien résoudre le compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients. Ce choix dépend fortement de la nature de workload (imprévisible, prévisible) ainsi que les caractéristiques de l'application (élastique ou non).



Conclusion

7

Conclusion et perspectives

Dans ce chapitre, nous dressons le bilan des travaux réalisés dans le cadre de cette thèse. Nous présentons, d'abord, un rappel des motivations et une synthèse des contributions. Ensuite, nous développons les perspectives à plus ou moins long terme de nos travaux.

Sommaire

7.1 Conclusion **152**

7.1.1 Contexte et problématique 152

7.1.2 Contributions 153

7.2 Perspectives **154**

7.2.1 Amélioration 155

7.2.2 Extensions 155

7.2.3 Pistes exploratoires 155

7.1 Conclusion

7.1.1 Contexte et problématique

L'informatique utilitaire est un concept qui a été proposé en 1961 par John McCarthy lors d'une conférence au MIT [GA99]. Il a expliqué que les ressources informatiques pourraient un jour être organisées comme un service public tout comme le téléphone, l'électricité ou le gaz. Ce concept a l'avantage d'avoir un coût d'investissement initial très faible pour un accès immédiat aux ressources informatiques.

Telle que l'imaginait John McCarthy, l'émergence du Cloud computing représente une suite logique dans l'histoire de l'informatique. L'informatique en nuage est une métaphore désignant un réseau de ressources informatiques (logicielles et/ou matérielles) accessibles à distance par le biais des technologies Internet [MGG11]. Ce modèle d'informatique utilitaire propose un nombre important d'offres. Un des éléments différenciateurs entre ces offres est le contrat de niveau de service proposé, nommé *Service Level Agreement* (SLA).

Le SLA est un document qui définit la qualité de service (QdS) prescrite entre un prestataire et un client. Il spécifie un ou plusieurs objectifs de niveau de service SLO (*Service Level Objective*). En cas de violation, des pénalités sont appliquées. La pénalité pénalise le prestataire de services en réduisant son profit et permet aux utilisateurs de tolérer la défaillance du service.

Le but de la gestion de capacité des ressources est l'assurance qu'un service répondra à la demande avec un niveau spécifié de qualité. L'objectif est de veiller à ce que la capacité du système puisse répondre aux objectifs de niveau de service tout en optimisant l'utilisation des ressources. Des initiatives industrielles telles que Amazon Auto-Scaling[1], RightScale[2] et Scalr[3] ainsi que des initiatives de recherche comme CloudScale [SSGW11] ou SmartScale [DGVV12] proposent des services de dimensionnement automatique. Ces solutions consistent principalement à utiliser les règles à base de seuils. Amazon Auto-Scaling, par exemple, permet d'augmenter ou diminuer automatiquement la capacité des instances Amazon EC2 selon des conditions prédéfinies. Cependant, un effort important de l'utilisateur du service de dimensionnement est nécessaire pour la planification de capacité : définir le nombre de ressources à ajouter ou à supprimer pour ajuster la capacité automatiquement selon un modèle de charge de travail nécessite une grande expertise.

Nous avons relevé d'autres lacunes au sein des approches et outils proposés pour la gestion de l'élasticité des ressources Cloud. Tout d'abord, les approches existantes ne tiennent pas compte du modèle économique de l'informatique en nuage à savoir le modèle de facturation (à l'heure, par exemple). En plus, ces solutions négligent le temps d'initialisation des instances. Ensuite, la plupart des approches ne considèrent que des critères de QdS de bas niveau tel que CPU pour la gestion de l'élasticité. Or, l'utilisation des critères de bas niveau est un bon indicateur de l'utilisation du système, mais il ne reflète pas clairement si la QdS fournie répond aux besoins des utilisateurs

[1] aws.amazon.com/autoscaling/
[2] www.rightscale.com
[3] http ://wiki.scalr.com/display/docs/Scaling

comme des critères de haut niveau (temps de réponse, par exemple). Enfin, aucune approche ne propose un SLA explicite et le modèle économique associé.

7.1.2 Contributions

Pour répondre à ces limitations, nous avons donc proposé, un langage spécifique, appelé *CSLA* pour la définition des SLAs pour tout niveau du Cloud intégrant une gestion fine des violations. Nous avons également fourni *HybridScale*, une approche hybride qui définit, dans un même prototype, toutes les étapes nécessaires pour la gestion de l'élasticité Cloud. Cette approche englobe des politiques de gestion de l'élasticité *Best-Policies* et une méthode de planification de capacité, appelée *RightCapacity*.

Définition de SLA : **CSLA** est un langage spécifique inspiré de SLA@SOI et WSLA. Ainsi, grâce à ce langage, la définition de contrat de service devient une réalité pour le Cloud et peut être présenté par n'importe quel langage dédié (XML, Java,...) pour n'importe quel service XaaS (SaaS, PaaS ou IaaS). Il permet de gérer des garanties multicritères avec et sans priorité pour refléter les préférences du client. En plus des caractéristiques classiques de définition des objectifs SLOs, *CSLA* propose un modèle économique novateur pour gérer plus finement les violations : la dégradation fonctionnelle (modes de service), la dégradation de QdS (*fuzziness* et *confidence*) et des modèles avancés de pénalité. De plus, *CSLA* suit l'architecture de référence de l'informatique en nuage de NIST et supporte l'interface OCCI.

La conséquence directe d'un tel langage est qu'il permet au fournisseur de service d'affiner ses stratégies de (re)configuration (par exemple, ordonnancement, contrôle d'admission, dimensionnement) pour arbitrer subtilement la gestion de ressources dans un contexte très dynamique.

Nous avons proposé une solution qui automatise le calibrage de SLA. Nous avons modélisé le problème de dépendances SLA comme un problème d'optimisation multicritères. Notre solution facilite le calibrage du template *CSLA* et l'aiguillage des ressources en terme de type et nombre. Cette approche est dirigée par les préférences du client, un aspect qui n'était pas considéré précédemment. Nous avons mis l'accent sur le fournisseur SaaS cependant notre approche est générique. Basée sur la programmation par contrainte, notre approche calcule à la volée (sur mesure) une solution dans un temps raisonnable. Elle permet de prendre en compte les paramètres QdS dynamiques de manière souple pour composer le meilleur processus métier vertical.

L'originalité de notre approche est de prendre appui sur les préférences des clients pour calibrer le SLA, l'énumération des k premières solutions, la flexibilité et la généricité. Notre approche est limitée à la phase de conception (design- time). Néanmoins, notre contribution est un premier effort vers la mise en lumière du problème de dépendances SLA au sein de la communauté et la nécessité d'y trouver une solution.

Gestion de SLA : **HybridScale** est un framework de dimensionnement automatique dirigé par SLA. Cette approche englobe d'autre contributions à savoir : *BestPolicies* des politiques de gestion d'élasticité, *Forecaster* une technique de prédiction de la demande et *RightCapacity*, une approche de planification de capacité dirigée par SLA.

Notre approche *HybridScale* est originale car elle implémente l'élasticité en triple hybride : gestion réactif-proactif, dimensionnement vertical-horizontal et multi-couches (application-infrastructure).

BestPolicies est un ensemble de politiques de gestion de l'élasticité. Nous distinguons trois types : i) les politiques de prix, ii) les politiques de réaction et iii) les politiques de dimensionnement. Pour ces dernières, nous avons proposé la liste de politiques suivantes : *Infrastructure Elasticity* qui consiste à contrôler le dimensionnement des ressources et *Application Elasticity* qui gère l'élasticité applicative (la dégradation de fonctionnalité). Nous proposons en particulier la stratégie de terminaison *UaP* pour harmoniser le dimensionnement automatique avec le modèle économique. Nos politiques sont enregistrées dans un fichier XML. L'originalité de cette contribution est la capacité d'absorber le démarrage des instances et des violations ainsi que l'adaptation au modèle économique.

RightCapacity est une méthode de planification de capacité. Nous avons proposé un modèle analytique, basé sur la théorie des files d'attente (MVA) pour évaluer les performances du service SaaS. Nous avons introduit l'élasticité au niveau application en la considérant comme une boîte blanche pour absorber le temps d'initialisation des ressources. L'originalité de *RightCapacity* est l'inter-couches (*cross-layer*). Elle tient en compte deux niveaux d'élasticité à savoir : l'application (SaaS) et l'infrastructure (IaaS). Cette méthode de planification est facilement paramétrable via les politiques *BestPolicies*.

HybridScale est adapté à n'importe quel type de charge pour gérer le compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients. Même face à une charge de travail imprévisible, notre solution réduit le nombre d'instance-heure à payer tout en minimisant les violations de SLA. Les resultats des expérimentations de notre contribution dans l'environnement Amazon EC2 montrent que la méthode *RightCapacity* combinée avec la stratégie *UaP* et la dégradation de fonctionnalité permet de réduire les coûts en gardant le dépassement des seuils toujours inférieur à 1%.

Pour conclure, le choix de la politique de réaction (réactive vs proactive) reste une question clé pour bien résoudre le compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients. Ce choix dépend fortement du contexte de la charge (workload imprévisible, prévisible) ainsi que les caractéristiques de l'application SaaS (élastique ou non).

7.2 Perspectives

En termes de perspectives de recherche, nous envisageons, à court terme, d'améliorer notre langage *CSLA* et d'enrichir les politiques *BestPolicies* afin d'obtenir un meilleur contrôle de dimensionnement. Ensuite, à moyen terme, nous planifions un certain nombre d'extensions pour *RightCapacity*. Enfin, pour le long terme, nous suggérons quelques pistes exploratoires.

7.2.1 Amélioration

A court terme, nous proposons des améliorations de *CSLA* et *BestPolicies*.

Nous envisageons de raffiner le langage *CSLA* de différentes manières. Nous projetons d'améliorer le modèle de pénalité en proposant d'autres fonctions en plus de la fonction linéaire. Ensuite, nous comptons développer notre solution de calibrage de template CSLA par un système d'apprentissage. Ceci permet de raffiner les propriétés *Fuzziness* et *Confidence* ainsi que le modèle économique (prix et pénalités). Enfin, nous envisageons de supporter d'autres standards pour améliorer l'interopérabilité et passer vers une version CSLA++ dans le cadre de projet OpenCloudware [Ope13].

Pour *BestPolicies*, nous envisageons d'utiliser les ressources à prix ponctuel (exemple spot instances de Amazon EC2) et de proposer par la suite une politique (modèle) dynamique des prix de service. A propos les politiques de dimensionnement, nous comptons parfaire notre approche de trois manières. Tout d'abord, nous planifions d'appliquer et d'adapter *HybridScale* à l'heure naturelle telle que proposée par Microsoft Azure. Ensuite, il est important de prendre en compte l'aspect énergétique via des politiques "vertes" pour maîtriser la consommation d'énergie qui fait l'objet d'une forte attention depuis quelques années. Enfin, il nous reste également à automatiser le calibrage de : i) la période de calme et ii) la fréquence de l'utilisation de dégradation selon le type de workload.

7.2.2 Extensions

A moyen terme, nous comptons apporter trois extensions au niveau de *RightCapacity*. Tout d'abord, nous envisageons d'étendre notre modèle analytique pour support un réseau ouvert ou un réseau mixte en s'appuyant sur la bibliothèque open source JMT (Java Modelling Tools) [4]. De plus, nous comptons optimiser la planification de la capacité applicative. Une première réflexion est d'utiliser également la théorie de file d'attente pour calculer la configuration applicative optimale. Ensuite, nous projetons d'améliorer le contrôle d'admission en offrant un ajustement des priorités des requêtes selon la demande. Enfin, nous comptons enrichir la méthode de planification pour considérer la migration entre tiers. Nous avons étudié récemment l'utilisation des ressources homogènes entre tiers. Nous comptons poursuivre cette étude car elle semble fort prometteuse pour améliorer *RightCapacity*.

7.2.3 Pistes exploratoires

A long terme, nous proposons l'exploration des pistes pour l'approche *HybridScale*. D'abord, nous planifions de fournir une coordination entre les événements notifiés et le cycle de la boucle autonomique afin d'améliorer le contrôle autonomique. Une alternative est d'utiliser les modèles de coordination et de synchronisation de plusieurs boucles de contrôle comme décrit dans [dOL12b]. De plus, nous prévoyons de fournir plus de précision sur le temps de reconfiguration de telle sorte que les décisions soient mieux maîtrisées.

[4]http ://jmt.sourceforge.net/JMVA.html

Après, nous planifions d'améliorer notre technique de prédiction pour faire la prédiction à long terme. Une telle prédiction permet de classifier à l'avance les requêtes/intervalles comme idéal, dégradé ou inadéquat pour mieux maîtriser les décisons. Également, la détection de la tendance est une piste importante à étudier afin de définir le type de réaction le plus adéquat pour mieux répondre au compromis entre le profit de fournisseur SaaS et la satisfaction de ses clients. Concernant la boucle autonomique et la méthode de planification, nous comptons nous inspirer des travaux de FScript [DLLC09] ou de VM Script pour mieux contrôler respectivement l'élasticité applicative ou l'élasticité des ressources. Ceci permet de faciliter le test, l'évaluation et l'administration des stratégies de dimensionnement. Puis, nous comptons enrichir notre approche en terme de type de Cloud (privé, public) et supporter la fédération (plusieurs fournisseurs) du Cloud. Enfin, nous projetons d'exploiter les mécanismes de synchronisation de données pour mieux maîtriser le dimensionnement horizontal des bases de données.

Bibliographie

[Aa07] A. Andrieux and al. *Web services agreement specification (ws-agreement)*. OGF, 2007. 5, 32, 34, 41, 72

[ADC11] Mohammed Alhamad, Tharam Dillon, and Elizabeth Chang. A survey on sla and performance measurement in cloud computing. In *Proceedings of the 2011th Confederated international conference on On the move to meaningful internet systems - Volume Part II*, OTM'11, pages 469–477, Berlin, Heidelberg, 2011. Springer-Verlag. 2

[AETE12] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 204–212, 2012. 52, 65

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds : A berkeley view of cloud computing. Technical report, 2009. 18

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4) :50–58, April 2010. 18

[AMCC+10] L. Al Moakar, P.K. Chrysanthis, C. Chung, S. Guirguis, A. Labrinidis, P. Neophytou, and K. Pruhs. Admission control mechanisms for continuous queries in the cloud. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 409–412, 2010. 61

[AMVBVL+13] F.J. Almeida Morais, F. Vilar Brasileiro, R. Vigolvino Lopes, R. Araujo Santos, W. Satterfield, and L. Rosa. Autoflex : Service agnostic auto-scaling framework for iaas deployment models. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 42–49, 2013. 49, 60, 61

[Azu13a] SLA Windows Azure. http ://www.windowsazure.com/en-us/support/legal/sla/. 2013. 40

[Azu13b] Windows Azure. http ://www.windowsazure.com/fr-fr/pricing/details/virtual-machines/. 2013. 20, 22, 26

[Bas12] Salman A. Baset. Cloud slas : present and future. *SIGOPS Oper. Syst. Rev.*, 46(2) :57–66, July 2012. 2, 15

[BB10] Anton Beloglazov and Rajkumar Buyya. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '10, pages 4 :1–4 :6, New York, NY, USA, 2010. ACM. 51

[BBC+03] D.F. Bantz, C. Bisdikian, D. Challener, J.P. Karidis, S. Mastrianni, A. Mohindra, D.G. Shea, and M. Vanover. Autonomic personal computing. *IBM Systems Journal*, 42(1) :165–176, 2003. 28

[BEM+10] Ivona Brandic, Vincent C. Emeakaroha, Michael Maurer, Schahram Dustdar, Sandor Acs, Attila Kertesz, and Gabor Kecskemeti. Laysi : A layered approach for sla-violation propagation in self-manageable cloud infrastructures. In *Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, COMPSACW '10, pages 365–370, Washington, DC, USA, 2010. IEEE Computer Society. 135

[BHD12] 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*, pages n/a–n/a, 2012. 53, 57

[BJ94] George Edward Pelham Box and Gwilym M. Jenkins. *Time Series Analysis : Forecasting and Control*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 1994. 6, 49, 112

[BRE13] BREIN. http ://www.eu-brein.com/. 2013. 35

[Buy09] Rajkumar Buyya. Market-oriented cloud computing : Vision, hype, and reality of delivering computing as the 5th utility. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 1–, Washington, DC, USA, 2009. IEEE Computer Society. 18

[BWRJ08] L. Bodenstaff, A. Wombacher, M. Reichert, and M.C. Jaeger. Monitoring dependencies for slas : The mode4sla approach. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 1, pages 21–29, 2008. 45

[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 Gener. Comput. Syst.*, 25(6) :599–616, June 2009. 17, 18

[cas13] Service component architecture (sca). http ://www. oasis-opencsa.org/sca. 2013. 17

[CBP+10] Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Rich Wolski. Appscale : Scalable and open appengine application development and deployment. In DimiterR. Avresky, Michel Diaz, Arndt Bode, Bruno Ciciani, and Eliezer Dekel, editors, *Cloud Computing*, volume 34 of *Lecture Notes of the Institute for*

Computer Sciences, Social-Informatics and Telecommunications Engineering, pages 57–70. Springer Berlin Heidelberg, 2010. 60

[CDM11] Eddy Caron, Frédéric Desprez, and Adrian Muresan. Pattern matching based forecast of non-periodic repetitive behavior for cloud clients. *J. Grid Comput.*, 9(1) :49–64, March 2011. 65

[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*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association. 50

[CGS03] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. *SIGMETRICS Perform. Eval. Rev.*, 31(1) :300–301, June 2003. 49

[CIL+08] Yuan Chen, Subu Iyer, Xue Liu, Dejan Milojicic, and Akhil Sahai. Translating service level objectives to lower level policies for multi-tier services. *Cluster Computing*, 11(3) :299–311, September 2008. 45, 46, 47

[Cit13] CiteSeer. citeseerx.ist.psu.edu/. 2013. 64

[Clo13a] Standards Cloud. http ://cloud-standards.org/. 2013. 23

[Clo13b] Cloud4SOA. http ://www.cloud4soa.eu/. 2013. 39

[Com13] CompatibleOne. http ://www.compatibleone.org/. 2013. 39

[Con13] Contrail. http ://contrail-project.eu/. 2013. 38, 41

[CSI+08] Yuan Chen, Akhil Sahai, Subu Iyer, Xue Liu, and Dejan Milojicic. Systematically translating service level objectives into design and operational policies for multi-tier applications. Technical report, HP Laboratories, 2008. 45, 47

[dBVB13] Ruben Van den Bossche, Kurt Vanmechelen, and Jan Broeckhove. Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds. *Future Generation Computer Systems*, 29(4) :973 – 985, 2013. <ce :title>Special Section : Utility and Cloud Computing</ce :title>. 62

[DC03] Bhavini Desai and Wendy Currie. Application service providers : a model in evolution. In *Proceedings of the 5th international conference on Electronic commerce*, ICEC '03, pages 174–180, New York, NY, USA, 2003. ACM. 15

[DGVV12] S. Dutta, S. Gera, Akshat Verma, and B. Viswanathan. Smartscale : Automatic application scaling in enterprise clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 221–228, 2012. 3, 50, 60, 152

[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, May 2011. 52, 57, 65

[DL11] D. Daniel and S.P.J. Lovesum. A novel approach for scheduling service request in cloud with trust monitor. In *Signal Processing, Communication, Computing and Networking Technologies (ICSCCN), 2011 International Conference on*, pages 509–513, 2011. 62

[DLLC09] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. Fpath and fscript : Language support for navigation and reliable reconfiguration of fractal architectures. *annals of telecommunications - annales des télécommunications*, 64(1-2) :45–63, 2009. 156

[DMRTV07] G. Di Modica, V. Regalbuto, O. Tomarchio, and L. Vita. Dynamic renegotiations of sla in service composition scenarios. In *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pages 359–366, 2007. 44

[DMV10] Jaideep Dhok, Nitesh Maheshwari, and Vasudeva Varma. Learning based opportunistic admission control algorithm for mapreduce as a service. In *Proceedings of the 3rd India software engineering conference*, ISEC '10, pages 153–160, New York, NY, USA, 2010. ACM. 61

[dOL12a] Frederico Alvares de Oliveira, Jr. and Thomas Ledoux. Self-management of cloud applications and infrastructure for energy optimization. *SIGOPS Oper. Syst. Rev.*, 46(2) :10–18, July 2012. 53

[dOL12b] Frederico Alvares de Oliveira, Jr. and Thomas Ledoux. Self-management of cloud applications and infrastructure for energy optimization. *SIGOPS Oper. Syst. Rev.*, 46(2) :10–18, July 2012. 92, 155

[drA13] Modèle de ressource Amazon. http ://awsdocs.s3.amazonaws.com/ec2/latest/ec2-api.pdf. 2013. vii, 23

[drD13] Modèle de ressource DeltaCloud. http ://deltacloud.apache.org/rest-api.html. 2013. vii, 24

[DRM+10] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck. From data center resource allocation to control theory and back. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 410–417, 2010. 50, 57

[ea11] O. Waeldrich et al. WS-Agreement Negotiation Version 1.0. *Open Grid Forum*, 2011. 35, 41

[ea12] J. Meegan et al. Practical Guide to Cloud Service Level Agreements Version 1.0. *Cloud Standards Customer Council*, 2012. 36

[ea13] K. Kritikos et al. A survey on service quality description. In *ACM Computing Surveys*, 2013. 2

[EBMD10] V.C. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar. Low level metrics to high level slas - lom2his framework : Bridging the gap between monitored metrics and sla parameters in cloud environments. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 48–54, 2010. 44, 45

[EC213a] Amazon EC2. http ://aws.amazon.com/fr/ec2/. 2013. vii, 20, 22, 26, 27

[EC213b] SLA Amazon EC2. http ://aws.amazon.com/ec2-sla/. 2013. 39

[EFN+12] Vincent C. Emeakaroha, Tiago C. Ferreto, Marco. A. S. Netto, Ivona Brandic, and Cesar A. F. De Rose. Casvid : Application level monitoring for sla violation detection in clouds. In *Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference*, COMPSAC '12, pages 499–508, Washington, DC, USA, 2012. IEEE Computer Society. 45

[Eng13a] Google App Engine. https ://cloud.google.com/products/. 2013. 20

[Eng13b] SLA Google App Engine. https ://developers.google.com/appengine/sla. 2013. 40

[FJJ+11] Liu Fang, Tong Jin, Mao Jian, Bohn Robert, Lee Badger John Messina, and Dawn Leaf. NIST Cloud Computing Reference Architecture. 2011. vii, 21

[GA99] Simson Garfinkel and Harold Abelson. *Architects of the Information Society : 35 Years of the Laboratory for Computer Science at Mit*. MIT Press, Cambridge, MA, USA, 1999. 11, 15, 17, 152

[GGW10] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press : Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16, 2010. 49, 57

[GO12] Shamsollah Ghanbari and Mohamed Othman. A priority based job scheduling algorithm in cloud computing. *Procedia Engineering*, 50(0) :778 – 785, 2012. International Conference on Advances Science and Contemporary Engineering 2012. 62

[Gol73] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, New York, NY, USA, 1973. ACM. 21

[GSLI11] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai. Exploring alternative approaches to implement an elasticity policy. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 716–723, 2011. 65

[HBS10a] Irfan Ul Haq, Ivona Brandic, and Erich Schikuta. Sla validation in layered cloud infrastructures. In *Proceedings of the 7th international conference on Economics of grids, clouds, systems, and services*, GECON'10, pages 153–164, Berlin, Heidelberg, 2010. Springer-Verlag. 43

[HBS10b] Irfan Ul Haq, Ivona Brandic, and Erich Schikuta. SLA Validation in Layered Cloud Infrastructures. pages 153–164, 2010. 43

[HGGG12] Rui Han, Li Guo, M.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, 2012. 59, 60, 65

[HGSZ10] Jinhua Hu, Jianhua Gu, Guofei Sun, and Tianhai Zhao. A scheduling strategy on load balancing of virtual machine resources in cloud computing environment. In *Parallel Architectures, Algorithms and Programming (PAAP), 2010 Third International Symposium on*, pages 89–96, 2010. 63

[HK06] Rob J. Hyndman and Anne B. Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4) :679 – 688, 2006. 113

[HKSW06] Peer Hasselmeyer, Bastian Koller, Lutz Schubert, and Philipp Wieder. Towards sla-supported resource management. In Michael Gerndt and Dieter KranzlmÃ¼ller, editors, *High Performance Computing and Communications*, volume 4208 of *Lecture Notes in Computer Science*, pages 743–752. Springer Berlin Heidelberg, 2006. 46

[HM08] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computingdegrees, models, and applications. *ACM Comput. Surv.*, 40(3) :7 :1–7 :28, August 2008. 28

[HMC+12] M.Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S.L.D. Gudreddi. Integrated and autonomic cloud resource scaling. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1327–1334, 2012. 51, 59, 65, 66

[Hor01] Paul Horn. Autonomic Computing : IBM's Perspective on the State of Information Technology. Technical report, 2001. 7, 28, 29, 67, 106, 107, 130, 183

[HPSvH03] Ian Horrocks, Peter Patel-Schneider, and Frank van Harmelen. From shiq and rdf to owl : The making of a web ontology language. *Web Semantics : Science, Services and Agents on the World Wide Web*, 1(1), 2003. 36

[HWH09] Li Hui, Theilmann Wolfgang, and Jens Happe. Sla translation in multilayered service oriented architectures : Status and challenges. Technical report, University of Karlsruhe, 2009. 43

[IDCJ11] Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Gener. Comput. Syst.*, 27(6) :871–879, June 2011. 59, 65

[IGC04] D.E. Irwin, L.E. Grit, and J.S. Chase. Balancing risk and reward in a market-based task service. In *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pages 160–169, 2004. 83

[IKLL12] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gener. Comput. Syst.*, 28(1) :155–162, January 2012. 49, 57, 58, 65, 66

[JJH+08] Gueyoung Jung, K.R. Joshi, M.A. Hiltunen, R.D. Schlichting, and C. Pu. Generating adaptation policies for multi-tier applications in consolidated server environments. In *Autonomic Computing, 2008. ICAC '08. International Conference on*, pages 23–32, 2008. 45, 46

[jJMM+02] Li jie Jin, Vijay Machiraju, Vijay Machiraju, Akhil Sahai, and Akhil Sahai. Analysis on service level agreement of web services. Technical report, HP Laboratories, 2002. 13

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, January 2003. vii, 28, 29

[KCH09] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pages 117–126, New York, NY, USA, 2009. ACM. 59, 60

[KCPV12] K. Konstanteli, T. Cucinotta, K. Psychas, and T. Varvarigou. Admission control for elastic cloud services. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 41–48, 2012. 61

[KGM10] S. Khatua, A. Ghosh, and N. Mukherjee. Optimizing the utilization of virtual resources in cloud environment. In *Virtual Environments Human-Computer Interfaces and Measurement Systems (VECIMS), 2010 IEEE International Conference on*, pages 82–87, 2010. 49

[KL03] Alexander Keller and Heiko Ludwig. The wsla framework : Specifying and monitoring service level agreements for web services. *J. Netw. Syst. Manage.*, 11(1) :57–81, March 2003. 32

[KMK09] Paul Karaenke, Andras Micsik, and Stefan Kirn. Adaptive SLA Management along Value Chains for Service Individualization. *Forum American Bar Association*, 2009 :217–228, 2009. 44

[KS07] Paul Karanke and K. I. R. N. Stefan. Service Level Agreements : An Evaluation from a Business Application Perspective. *Proceedings eChallenges e-2007 Conference*, 2007. 44

[KSI+08] Vibhore Kumar, K. Schwan, S. Iyer, Yuan Chen, and A. Sahai. A state-space approach to sla based management. In *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pages 192–199, 2008. 46

[KTK10] K.T. Kearney, F. Torelli, and C. Kotsokalis. Sla* ; : An abstract syntax for service level agreements. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 217–224, 2010. 5, 36, 37

[KVC11] K. Konstanteli, T. Varvarigou, and T. Cucinotta. Probabilistic admission control for elastic cloud computing. In *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on*, pages 1–4, 2011. 61

[KYG10] Constantinos Kotsokalis, Ramin Yahyapour, and Miguel Angel Rojas Gonzalez. SAMI The SLA Management Instance. *2010 Fifth International Conference on Internet and Web Applications and Services*, pages 303–308, 2010. 45, 46

[LBC10] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, pages 1–10, New York, NY, USA, 2010. ACM. 51, 52, 57

[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*, ACDC '09, pages 13–18, New York, NY, USA, 2009. ACM. 51

[LDD10] N. Leontiou, D. Dechouniotis, and S. Denazis. Adaptive admission control of distributed cloud services. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 318–321, 2010. 61

[LF03] Keller A. Dan A. King R. P. Ludwig, H. and R Franck. *Web service level agreement (wsla) language specification*. IBM, 2003. 5, 32, 41, 72

[LG11] Xin Lu and Zilong Gu. A load-adapative cloud resource scheduling model based on ant colony algorithm. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pages 296–300, 2011. 62

[LH05] Oussama Layaida and Daniel Hagimont. Designing self-adaptive multimedia applications through hierarchical reconfiguration. In *Proceedings of the 5th IFIP WG 6.1 international conference on Distributed Applications and Interoperable Systems*, DAIS'05, pages 95–107, 2005. 64

[LLM+10] Hao Liu, Shijun Liu, Xiangxu Meng, Chengwei Yang, and Yong Zhang. Lbvs : A load balancing strategy for virtual storage. In *Service Sciences (ICSS), 2010 International Conference on*, pages 257–262, 2010. 63, 64

[LSEO3] D.D. Lamanna, J. Skene, and W. Emmerich. SLAng A language for defining service level agreements. *The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 100–106, 20O3. 32, 36

[LXK+11] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert Greenberg. Join-idle-queue : A novel load balancing algorithm for dynamically scalable web services. *Perform. Eval.*, 68(11) :1056–1071, November 2011. 63, 64

[LZ10] P. Lama and Xiaobo Zhou. Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 151–160, 2010. 59, 60, 65

[MBS11] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Enacting slas in clouds using rules. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 455–466, Berlin, Heidelberg, 2011. Springer-Verlag. 57, 58, 65

[MC04] Daniel A. Menascá and Emiliano Casalicchio. Qos in grid computing. *IEEE Internet Computing*, 8(4) :85–87, July 2004. 15

[MDA04] Daniel A. Menasce, Lawrence W. Dowdy, and Virgilio A. F. Almeida. *Performance by Design : Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. 51, 116

[MGG11] Peter Mell, Timothy Grance, and Timothy Grance. The NIST Definition of Cloud Computing. *Nist Special Publication*, 145, 2011. 2, 18, 19, 20, 36, 152

[MH11] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 49 :1–49 :12, New York, NY, USA, 2011. ACM. 65

[MH12] Ming Mao and M. 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, 2012. 5, 27, 66

[Mic02] Sun Microsystems. Service level agreement in the data center. Technical report, 2002. 13, 14

[Mil68] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM. 4

[MMSW07] M. Michael, J.E. Moreira, D. Shiloach, and R.W. Wisniewski. Scale-up x scale-out : A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007. 50

[mOS13] mOSic. http ://www.mosaic-cloud.eu/. 2013. 39

[MyC13] MyCloud. http ://mycloud.inrialpes.fr/. 2013. 8, 39, 95

[NMNAJ12] Klaithem Al Nuaimi, Nader Mohamed, Mariam Al Nuaimi, and Jameela Al-Jaroodi. A survey of load balancing in cloud computing : Challenges and algorithms. *Network Cloud Computing and Applications, International Symposium on*, 0 :137–142, 2012. 62

[NPF10] V. Nae, R. Prodan, and T. Fahringer. Cost-efficient hosting and load balancing of massively multiplayer online games. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 9–16, 2010. 63, 64

[NRSR13] Herbst Nikolas R., Kounev Samuel, and Reussner Ralf. Elasticity in cloud computing : What it is, and what it is not. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '13, 2013. 2, 25

[NRTV07] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA, 2007. 51

[OCC10] OCCI. Open cloud computing interface - infrastructure. Technical report, 2010. vii, 24, 25

[Ont13] Web Service Modeling Ontology. http ://www.wsmo.org/TR/d2/v1.1/. 2013. 36

[Ope13] OpenCloudware. www.opencloudware.org/ ? 2013. 8, 39, 155

[Ora10] Oracle. Oracle cloud resource model api. Technical report, 2010. vii, 23, 24

[Pas05] a. Paschke. RBSLA A declarative Rule-based Service Level Agreement Language based on RuleML. *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, pages 308–314, 2005. 32, 36

[PDPBG09] Jérémy Philippe, Noël De Palma, Fabienne Boyer, and Olivier Gruber. Self-adapting service level in java enterprise edition. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 8 :1–8 :20, New York, NY, USA, 2009. Springer-Verlag New York, Inc. 64

[PH07] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented architectures : approaches, technologies and research issues. *The VLDB Journal*, 16(3) :389–415, July 2007. 16

[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*, EuroSys '09, pages 13–26, New York, NY, USA, 2009. ACM. 57, 58

[RA01] S. Ron and P. Aliko. Service level agreements. Technical report, Internet NG, 2001. 13

[Rac13] Rackspace. http ://www.rackspace.com/cloud/. 2013. 36

[RBHJ08] S. Rosario, A. Benveniste, S. Haar, and C. Jard. Probabilistic qos and soft contracts for transaction-based web services orchestrations. *Services Computing, IEEE Transactions on*, 1(4) :187–200, 2008. 44

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006. 6, 53, 98

[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*, ICAC '09, pages 137–146, New York, NY, USA, 2009. ACM. 52, 53, 57, 58

[RDG11] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507, 2011. 49, 59

[RG05] M. Rosenblum and T. Garfinkel. Virtual machine monitors : current technology and future trends. *Computer*, 38(5) :39–47, 2005. 50

[RL80] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2) :313–322, April 1980. 51, 115, 137

[SB98] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. 52, 137

[SGLI11] B. Simmons, H. Ghanbari, M. Litoiu, and G. Iszlai. Managing a saas application in the cloud using paas policy sets and a strategy-tree. In *Network and Service Management (CNSM), 2011 7th International Conference on*, pages 1–5, 2011. 65

[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, 2010. 53

[SKM09] George Spanoudakis, Christos Kloukinas, and Khaled Mahbub. The serenity runtime monitoring framework. In *Security and Dependability for Ambient Intelligence*, volume 45 of *Advances in Information Security*, pages 213–237. Springer US, 2009. 45

[SN05] J.E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5) :32–38, 2005. 21, 22

[SPU13] Praveen S Phani and Tulasi U. A study on qos challenges in cloud computing. In *International Journal Of Computers and Communications*, 2013. 2

[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*, SOCC '11, pages 5 :1–5 :14, New York, NY, USA, 2011. ACM. 3, 60, 65, 152

[ST05] L. Srinivasan and J. Treadwell. An overview of service-oriented architecture, web services and grid computing. *HP Software Global Business Unit*, 2005. 16

[ST12] Bhushan Lal Sahu and Rajesh Tiwari. A cost effective scaling approach for cloud applications. *International Journal of Advanced Research in Computer Engineering and Technology(IJARCET)*, 1(9), 2012. 48

[SW07] Xian-He Sun and Ming Wu. Quality of service of grid computing : Resource sharing. In *Proceedings of the Sixth International Conference on Grid and Cooperative Computing*, GCC '07, pages 395–402, Washington, DC, USA, 2007. IEEE Computer Society. 15

[TAP13] TAPAS. http ://uclslang.sourceforge.net/. 2013. 36

[Tay11] Sandeep Tayal. Tasks scheduling optimization for the cloud computing systems. 2011. 62

[TM10] Fei Teng and Frédéric Magoulès. A new game theoretical resource allocation algorithm for cloud computing. In Paolo Bellavista, Ruay-Shiung Chang, Han-Chieh Chao, Shin-Feng Lin, and PeterM.A. Sloot, editors, *Advances in Grid and Pervasive Computing*, volume 6104 of *Lecture Notes in Computer Science*, pages 321–330. Springer Berlin Heidelberg, 2010. 52

[TPE+02] Vladimir Tosic, Bernard Pagurek, Babak Esfandiari, Kruti Patel, and Wei Ma. Web Service Offerings Language (WSOL) and Web Service Composition Management (WSCM). *Computer*, 2002. 32, 36

[UPS+07] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. Analytic modeling of multitier internet applications. *ACM Trans. Web*, 1(1), May 2007. 45, 46, 137

[USC+08] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1) :1 :1–1 :39, March 2008. 59

[VAN08a] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pmapper : power and migration cost aware application placement in virtualized systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 243–264, New York, NY, USA, 2008. Springer-Verlag New York, Inc. 50

[VAN08b] Akshat Verma, Puneet Ahuja, and Anindya Neogi. Power-aware dynamic placement of hpc applications. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 175–184, New York, NY, USA, 2008. ACM. 50

[VdBVB10] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove. Cost-optimal scheduling in hybrid iaas clouds for deadline constrained workloads. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 228–235, 2010. 62

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages : an annotated bibliography. *SIGPLAN Not.*, 35(6) :26–36, June 2000. 137

[VRMCL08] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds : towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1) :50–55, December 2008. 18

[WB10] Linlin Wu and Rajkumar Buyya. Service Level Agreement (SLA) in Utility Computing Systems. *Architecture*, 2010. 12, 13, 15

[WCC12] Wenting Wang, Haopeng Chen, and Xi Chen. An availability-aware approach to resource placement of dynamic scaling in clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 930–931, 2012. 50

[WCSO08] H. Wada, P. Champrasert, J. Suzuki, and K. Oba. Multiobjective optimization of sla-aware service composition. In *Services - Part I, 2008. IEEE Congress on*, pages 368–375, 2008. 45, 47

[WKGB12] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. Sla-based admission control for a software-as-a-service provider in cloud computing environments. *J. Comput. Syst. Sci.*, 78(5) :1280–1299, September 2012. 61, 89

[WMZ03] Charles P. Wright, Michael C. Martino, and Erez Zadok. Ncryptfs : A secure and convenient cryptographic file system. In *In Proceedings of the Annual USENIX Technical Conference*, pages 197–210. USENIX Association, 2003. 64

[wp13a] Cloud workload patterns. http ://watdenkt.veenhof.nu/2010/07/13/workload-patterns-for-cloud-computing/. 2013. 50

[WP13b] WS-Policy. http ://www.w3.org/TR/ws-policy/. 2013. 39

[WVZX10] Guiyi Wei, AthanasiosV. Vasilakos, Yao Zheng, and Naixue Xiong. A game-theoretic method of fair resource allocation forÂ cloud computing services. *The Journal of Supercomputing*, 54(2) :252–269, 2010. 52

[WXZ+11] Lixi Wang, Jing Xu, Ming Zhao, Yicheng Tu, and Jose A. B. Fortes. Fuzzy modeling based resource management for virtualized database systems. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 32–42, Washington, DC, USA, 2011. IEEE Computer Society. 52, 57, 66

[WXZF11] Lixi Wang, Jing Xu, Ming Zhao, and José Fortes. Adaptive virtual resource management with fuzzy model predictive control. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 191–192, New York, NY, USA, 2011. ACM. 52

[WY11] Butler J. Theilmann W. Wieder, P. and R Yahyapour. *Service level agreements for cloud computing*. springer, 2011. 41, 44, 45, 46, 47, 71, 72

[YB06] Chee Shin Yeo and Rajkumar Buyya. A taxonomy of market-based resource management systems for utility-driven cluster computing. *Softw. Pract. Exper.*, 36(13) :1381–1419, November 2006. 15

[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. 18

[ZJ11] Wolfgang Ziegler and Ming Jiang. OPTIMIS SLA Framework and Term Languages for SLAs in Cloud Environment. 2011. 38, 41

[ZPL+12] Han Zhao, Miao Pan, Xinxin Liu, Xiaolin Li, and Yuguang Fang. Optimal resource rental planning for elastic applications in cloud market. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 808–819, Washington, DC, USA, 2012. IEEE Computer Society. 53

[ZWL08] Tao Zheng, M. Woodside, and M. Litoiu. Performance model estimation and tracking using optimal filters. *Software Engineering, IEEE Transactions on*, 34(3) :391–406, 2008. 46

[ZZ10] Zehua Zhang and Xuejie Zhang. A load balancing mechanism based on ant colony and complex network theory in open cloud computing federation. In *Industrial Mechatronics and Automation (ICIMA), 2010 2nd International Conference on*, volume 2, pages 240–243, 2010. 63, 64

[ZZCT12] Liang Zhou, Baoyu Zheng, Jingwu Cui, and Sulan Tang. Toward green service in cloud : From the perspective of scheduling. In *Computing, Networking and Communications (ICNC), 2012 International Conference on*, pages 939–943, 2012. 62

Annexes

Syntaxe de CSLA

```
<?xml version="1.0" encoding="UTF−8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:csla="http://www.inria.fr/cslamodel"
           targetNamespace="http://www.inria.fr/cslamodel"
           elementFormDefault="qualified">

<xs:annotation>
  <xs:documentation>
        Schema for the CSLA Language.
        Version: csla−2
        Revision: 0.3
        Author: Yousri KOUKI − Yousri.KOUKI@gmail.com
        Last revision 12/06/2013
  </xs:documentation>
</xs:annotation>

 <xs:element name="CSLA">
   <xs:complexType>
     <xs:sequence>
       <xs:element name="validity" type="csla:ValidityType" minOccurs="1" maxOccurs="1"/>
       <xs:element name="parties" type="csla:PartiesType" minOccurs="1" maxOccurs="1"/>
       <xs:element name="template" type="csla:TemplateType" minOccurs="1" maxOccurs="1"/>
     </xs:sequence>
     <xs:attribute name="id" type="xs:string"/>
     <xs:attribute name="agreedAt" type="xs:date"/>
     <xs:attribute name="template" type="xs:string"/>
   </xs:complexType>
 </xs:element>

<xs:complexType name="ValidityType">
  <xs:attribute name="effectiveFrom" type="xs:date"/>
  <xs:attribute name="effectiveUntil" type="xs:date"/>
</xs:complexType>

<xs:complexType name="PartiesType">
  <xs:sequence>
    <xs:element name="cloudProvider" type="csla:PartyType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="cloudConsumer" type="csla:PartyType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="supportingParty" type="csla:PartyType" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="PartyType">
  <xs:sequence>
    <xs:element name="name" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="contact" type="csla:ContactType" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
```

```xml
49
50 <xs:complexType name="ContactType">
51   <xs:sequence>
52     <xs:element name="address" type="xs:string" minOccurs="1" maxOccurs="1"/>
53     <xs:element name="email" type="xs:string" minOccurs="1" maxOccurs="1"/>
54     <xs:element name="phoneNumber" type="xs:string" minOccurs="1" maxOccurs="1"/>
55   </xs:sequence>
56 </xs:complexType>
57
58 <xs:complexType name="TemplateType">
59   <xs:sequence>
60     <xs:element name="cloudServices" type="csla:CloudServicesType" minOccurs="1" maxOccurs="1"/>
61     <xs:element name="parameters" type="csla:ParametersType" minOccurs="1" maxOccurs="1"/>
62     <xs:element name="guarantees" type="csla:GuaranteesType" minOccurs="1" maxOccurs="1"/>
63     <xs:element name="billing" type="csla:BillingType" minOccurs="1" maxOccurs="1"/>
64     <xs:element name="terminations" type="csla:TerminationsType" minOccurs="1" maxOccurs="1"/>
65   </xs:sequence>
66  <xs:attribute name="name" type="xs:string"/>
67  <xs:attribute name="version" type="xs:string"/>
68 </xs:complexType>
69
70 <xs:complexType name="CloudServicesType">
71   <xs:sequence>
72     <xs:element name="cloudService" type="csla:CloudServiceType" minOccurs="1" maxOccurs="unbounded"/>
73   </xs:sequence>
74 </xs:complexType>
75
76 <xs:complexType name="CloudServiceType">
77   <xs:choice>
78     <xs:element name="software" type="csla:SoftwareType" minOccurs="1" maxOccurs="1"/>
79     <xs:element name="platform" type="csla:PlatformType" minOccurs="1" maxOccurs="1"/>
80     <xs:element name="infrastructure" type="csla:InfrastructureType" minOccurs="1" maxOccurs="1"/>
81   </xs:choice>
82 </xs:complexType>
83
84 <xs:complexType name="SoftwareType">
85   <xs:sequence>
86     <xs:element name="service" type="csla:ProductType" minOccurs="1" maxOccurs="unbounded"/>
87   </xs:sequence>
88 </xs:complexType>
89
90 <xs:complexType name="ProductType">
91   <xs:sequence>
92     <xs:element name="mode" type="csla:ModeType" minOccurs="1" maxOccurs="unbounded"/>
93   </xs:sequence>
94   <xs:attribute name="id" type="xs:string"/>
95   <xs:attribute name="name" type="xs:string"/>
96   <xs:attribute name="mode" type="xs:integer"/>
97   <xs:attribute name="version" type="xs:string"/>
98   <xs:attribute name="distribution" type="xs:string"/>
99   <xs:attribute name="price" type="xs:string"/>
100  <xs:attribute name="license" type="xs:string"/>
101 </xs:complexType>
102
103 <xs:complexType name="ModeType">
104   <xs:attribute name="id" type="xs:string"/>
105   <xs:attribute name="name" type="xs:string"/>
106   <xs:attribute name="description" type="xs:string"/>
107   <xs:attribute name="price" type="xs:string"/>
108 </xs:complexType>
109
110 <xs:complexType name="PlatformType">
111  <xs:sequence>
112     <xs:element name="service" type="csla:ProductType" minOccurs="1" maxOccurs="unbounded"/>
113  </xs:sequence>
114 </xs:complexType>
115
116 <xs:complexType name="InfrastructureType">
117   <xs:choice>
118     <xs:element name="computeService" type="csla:ComputeInstancesType" minOccurs="1" maxOccurs="1"/>
119     <xs:element name="storageService" type="csla:StorageInstancesType" minOccurs="1" maxOccurs="1"/>
120   </xs:choice>
121   <xs:attribute name="id" type="xs:string"/>
122   <xs:attribute name="name" type="xs:string"/>
123 </xs:complexType>
```

```
124
125 <xs:complexType name="ComputeInstancesType">
126  <xs:sequence>
127      <xs:element name="instance" type="csla:ComputeInstanceType" minOccurs="1" maxOccurs="unbounded"/>
128  </xs:sequence>
129  <xs:attribute name="id" type="xs:string"/>
130  <xs:attribute name="name" type="xs:string"/>
131 </xs:complexType>
132
133 <xs:complexType name="ComputeInstanceType">
134   <xs:sequence>
135      <xs:element name="resource" type="csla:ResourceType" minOccurs="1" maxOccurs="1"/>
136      <xs:element name="image" type="csla:ImageType" minOccurs="0" maxOccurs="1"/>
137      <xs:element name="dataTransfer" type="csla:DataTransferType" minOccurs="0" maxOccurs="1"/>
138   </xs:sequence>
139   <xs:attribute name="id" type="xs:string"/>
140   <xs:attribute name="name" type="xs:string"/>
141 </xs:complexType>
142
143 <xs:complexType name="ResourceType">
144     <xs:sequence>
145       <xs:element name="compute" type="csla:ComputeType" minOccurs="1" maxOccurs="1"/>
146       <xs:element name="network" type="csla:NetworkType" minOccurs="1" maxOccurs="1"/>
147       <xs:element name="storage" type="csla:StorageType" minOccurs="1" maxOccurs="1"/>
148     </xs:sequence>
149     <xs:attribute name="id" type="xs:string"/>
150     <xs:attribute name="name" type="xs:string"/>
151 </xs:complexType>
152
153 <xs:complexType name="ImageType">
154   <xs:sequence>
155      <xs:element name="system" type="csla:ProductType" minOccurs="1" maxOccurs="1"/>
156      <xs:element name="package" type="csla:ProductType" minOccurs="0" maxOccurs="unbounded"/>
157   </xs:sequence>
158   <xs:attribute name="id" type="xs:string"/>
159   <xs:attribute name="name" type="xs:string"/>
160   <xs:attribute name="packages" type="xs:integer"/>
161   <xs:attribute name="price" type="xs:string"/>
162 </xs:complexType>
163
164 <xs:complexType name="StorageInstancesType">
165  <xs:sequence>
166      <xs:element name="instance" type="csla:StorageInstanceType" minOccurs="1" maxOccurs="unbounded"/>
167  </xs:sequence>
168  <xs:attribute name="id" type="xs:string"/>
169  <xs:attribute name="name" type="xs:string"/>
170 </xs:complexType>
171
172 <xs:complexType name="StorageInstanceType">
173   <xs:sequence>
174      <xs:element name="dataTransfer" type="csla:DataTransferType" minOccurs="0" maxOccurs="1"/>
175      <xs:element name="storage" type="csla:StorageType" minOccurs="1" maxOccurs="1"/>
176   </xs:sequence>
177   <xs:attribute name="id" type="xs:string"/>
178   <xs:attribute name="name" type="xs:string"/>
179 </xs:complexType>
180
181 <xs:complexType name="DataTransferType">
182   <xs:sequence>
183      <xs:element name="in" type="csla:TransferType" minOccurs="1" maxOccurs="unbounded"/>
184      <xs:element name="out" type="csla:TransferType" minOccurs="1" maxOccurs="unbounded"/>
185   </xs:sequence>
186   <xs:attribute name="id" type="xs:string"/>
187  </xs:complexType>
188
189  <xs:complexType name="TransferType">
190     <xs:attribute name="id" type="xs:string"/>
191     <xs:attribute name="size" type="xs:string"/>
192     <xs:attribute name="description" type="xs:string"/>
193     <xs:attribute name="price" type="xs:string"/>
194  </xs:complexType>
195
196  <xs:complexType name="ComputeType">
197     <xs:attribute name="id" type="xs:string"/>
198     <xs:attribute name="name" type="xs:string"/>
```

```
199      <xs:attribute name="architecture" type="xs:string"/>
200      <xs:attribute name="hostname" type="xs:string"/>
201      <xs:attribute name="cores" type="xs:string"/>
202      <xs:attribute name="speed" type="xs:string"/>
203      <xs:attribute name="memory" type="xs:string"/>
204      <xs:attribute name="price" type="xs:string"/>
205    </xs:complexType>
206
207    <xs:complexType name="StorageType">
208      <xs:attribute name="id" type="xs:string"/>
209      <xs:attribute name="name" type="xs:string"/>
210      <xs:attribute name="size" type="xs:string"/>
211      <xs:attribute name="type" type="xs:string"/>
212      <xs:attribute name="price" type="xs:string"/>
213    </xs:complexType>
214
215    <xs:complexType name="NetworkType">
216      <xs:attribute name="id" type="xs:string"/>
217      <xs:attribute name="name" type="xs:string"/>
218      <xs:attribute name="label" type="xs:string"/>
219      <xs:attribute name="vlan" type="xs:string"/>
220      <xs:attribute name="price" type="xs:string"/>
221    </xs:complexType>
222
223    <xs:complexType name="ParametersType">
224      <xs:sequence>
225        <xs:element name="metric" type="csla:MetricType" minOccurs="1" maxOccurs="unbounded"/>
226        <xs:element name="monitoring" type="csla:MonitoringType" minOccurs="1" maxOccurs="unbounded"/>
227        <xs:element name="schedule" type="csla:ScheduleType" minOccurs="1" maxOccurs="unbounded"/>
228      </xs:sequence>
229    </xs:complexType>
230
231    <xs:complexType name="MetricType">
232      <xs:sequence>
233        <xs:element name="description" type="xs:string" minOccurs="1" maxOccurs="1"/>
234      </xs:sequence>
235      <xs:attribute name="id" type="xs:string"/>
236      <xs:attribute name="name" type="xs:string"/>
237      <xs:attribute name="unit" type="xs:string"/>
238    </xs:complexType>
239
240    <xs:complexType name="MonitoringType">
241      <xs:attribute name="id" type="xs:string"/>
242      <!-- average, sum, minimum, maximum, or f() -->
243      <xs:attribute name="statistic" type="xs:string"/>
244      <xs:attribute name="window" type="xs:double"/>
245      <!-- the frequency of monitoring -->
246      <xs:attribute name="frequency" type="xs:double"/>
247    </xs:complexType>
248
249    <xs:complexType name="ScheduleType">
250      <xs:attribute name="id" type="xs:string"/>
251      <xs:attribute name="start" type="xs:string"/>
252      <xs:attribute name="end" type="xs:string"/>
253    </xs:complexType>
254
255    <xs:complexType name="GuaranteesType">
256      <xs:sequence>
257        <xs:element name="guarantee" type="csla:GuaranteeType" minOccurs="1" maxOccurs="unbounded"/>
258      </xs:sequence>
259    </xs:complexType>
260
261    <xs:complexType name="GuaranteeType">
262      <xs:sequence>
263        <xs:element name="scope" type="csla:ScopeType" minOccurs="1" maxOccurs="1"/>
264        <xs:element name="requirements" type="csla:RequirementsType" minOccurs="0" maxOccurs="1"/>
265        <xs:element name="terms" type="csla:TermsType" minOccurs="1" maxOccurs="unbounded"/>
266        <xs:element name="penalties" type="csla:PenaltiesType" minOccurs="1" maxOccurs="1"/>
267      </xs:sequence>
268      <xs:attribute name="id" type="xs:string"/>
269    </xs:complexType>
270
271    <xs:complexType name="ScopeType">
272     <xs:sequence>
273       <xs:element name="service" type="csla:ScopeServiceType" minOccurs="1" maxOccurs="unbounded"/>
```

```
274     </xs:sequence>
275       <xs:attribute name="id" type="xs:string"/>
276   </xs:complexType>
277
278   <xs:complexType name="ScopeServiceType">
279       <xs:attribute name="id" type="xs:string"/>
280       <xs:attribute name="subid" type="xs:string"/>
281   </xs:complexType>
282
283   <xs:complexType name="RequirementsType">
284     <xs:sequence>
285       <xs:element name="requirement" type="csla:RequirementType" minOccurs="0" maxOccurs="unbounded"/>
286     </xs:sequence>
287   </xs:complexType>
288
289 <xs:complexType name="RequirementType">
290   <xs:sequence>
291         <xs:element name="specification" type="csla:SpecificationType" minOccurs="1" maxOccurs="unbounded"/>
292   </xs:sequence>
293   <xs:attribute name="id" type="xs:string"/>
294 </xs:complexType>
295
296 <xs:complexType name="SpecificationType" mixed="true">
297   <xs:attribute name="id" type="xs:string"/>
298   <xs:attribute name="policy" type="xs:string"/>
299 </xs:complexType>
300
301 <xs:complexType name="TermsType">
302   <xs:sequence>
303     <xs:element name="term" type="csla:TermType" minOccurs="1" maxOccurs="unbounded"/>
304     <xs:element name="objective" type="csla:ObjectiveType" minOccurs="1" maxOccurs="unbounded"/>
305   </xs:sequence>
306 </xs:complexType>
307
308 <xs:complexType name="TermType">
309   <xs:sequence>
310     <xs:element name="item" type="csla:ItemType" minOccurs="1" maxOccurs="unbounded"/>
311   </xs:sequence>
312   <xs:attribute name="id" type="xs:string"/>
313   <xs:attribute name="operator" type="xs:string"/>
314 </xs:complexType>
315
316 <xs:complexType name="ItemType">
317   <xs:attribute name="id" type="xs:string"/>
318 </xs:complexType>
319
320 <xs:complexType name="ObjectiveType">
321   <xs:sequence>
322     <xs:element name="precondition" type="csla:ConditionType" minOccurs="0" maxOccurs="1"/>
323     <xs:element name="expression" type="csla:ConstraintType" minOccurs="1" maxOccurs="1"/>
324   </xs:sequence>
325   <xs:attribute name="id" type="xs:string"/>
326   <xs:attribute name="actor" type="xs:string"/>
327   <xs:attribute name="priority" type="xs:integer" nillable="true"/>
328 </xs:complexType>
329
330 <xs:complexType name="ConditionType">
331   <xs:sequence>
332     <xs:element name="description" type="xs:string" minOccurs="1" maxOccurs="1"/>
333   </xs:sequence>
334   <xs:attribute name="policy" type="xs:string"/>
335 </xs:complexType>
336
337 <xs:complexType name="ConstraintType">
338   <xs:attribute name="metric" type="xs:string"/>
339   <xs:attribute name="comparator" type="xs:string"/>
340   <xs:attribute name="threshold" type="xs:double"/>
341   <xs:attribute name="unit" type="xs:string"/>
342   <xs:attribute name="confidence" type="xs:double"/>
343   <xs:attribute name="fuzziness-value" type="xs:double"/>
344   <xs:attribute name="fuzziness-percentage" type="xs:double"/>
345   <xs:attribute name="schedule" type="xs:string"/>
346   <xs:attribute name="monitoring" type="xs:string"/>
347   </xs:complexType>
348
```

```
349 <xs:complexType name="PenaltiesType">
350   <xs:sequence>
351     <xs:element name="penalty" type="csla:PenaltyType" minOccurs="1" maxOccurs="unbounded"/>
352   </xs:sequence>
353 </xs:complexType>
354
355 <xs:complexType name="PenaltyType">
356   <xs:sequence>
357     <xs:choice>
358       <xs:element name="constant" type="csla:ConstantType" minOccurs="1" maxOccurs="1"/>
359       <xs:element name="function" type="csla:FunctionType" minOccurs="1" maxOccurs="1"/>
360     </xs:choice>
361     <xs:element name="procedure" type="csla:ProcedureType" minOccurs="1" maxOccurs="1"/>
362   </xs:sequence>
363   <xs:attribute name="id" type="xs:string"/>
364   <xs:attribute name="objective" type="xs:string"/>
365   <xs:attribute name="condition" type="xs:string"/>
366   <xs:attribute name="actor" type="xs:string"/>
367 </xs:complexType>
368
369 <xs:complexType name="ProcedureType">
370   <xs:sequence>
371     <xs:element name="violationDescription" type="xs:string"/>
372   </xs:sequence>
373   <xs:attribute name="actor" type="xs:string"/>
374   <xs:attribute name="notificationPeriod" type="xs:string"/>
375   <xs:attribute name="notificationMethod" type="xs:string"/>
376 </xs:complexType>
377
378 <xs:complexType name="ConstantType">
379   <xs:attribute name="value" type="xs:double"/>
380   <xs:attribute name="unit" type="xs:string"/>
381 </xs:complexType>
382
383 <xs:complexType name="FunctionType">
384   <xs:sequence>
385    <xs:element name="description" type="xs:string" minOccurs="1" maxOccurs="1"/>
386   </xs:sequence>
387   <xs:attribute name="ratio" type="xs:string"/>
388   <xs:attribute name="variable" type="xs:string"/>
389   <xs:attribute name="unit" type="xs:string"/>
390 </xs:complexType>
391
392 <xs:complexType name="BillingType">
393   <xs:choice>
394     <xs:element name="payasYouGo" type="csla:PayasYouGoType" minOccurs="1" maxOccurs="1"/>
395     <xs:element name="forfaiting" type="csla:ForfaitingType" minOccurs="1" maxOccurs="1"/>
396   </xs:choice>
397 </xs:complexType>
398
399 <xs:complexType name="PayasYouGoType">
400   <xs:sequence>
401     <xs:element name="description" type="xs:anyType" minOccurs="1" maxOccurs="1"/>
402   </xs:sequence>
403 </xs:complexType>
404
405 <xs:complexType name="ForfaitingType">
406   <xs:sequence>
407     <xs:element name="description" type="xs:anyType" minOccurs="1" maxOccurs="1"/>
408   </xs:sequence>
409 </xs:complexType>
410
411 <xs:complexType name="TerminationsType">
412   <xs:sequence>
413     <xs:element name="termination" type="csla:TerminationType" minOccurs="1" maxOccurs="unbounded"/>
414   </xs:sequence>
415 </xs:complexType>
416
417 <xs:complexType name="TerminationType">
418   <xs:sequence>
419     <xs:element name="terminationDescription" type="xs:string"/>
420   </xs:sequence>
421   <xs:attribute name="id" type="xs:string"/>
422   <xs:attribute name="terminationInitiator" type="xs:string"/>
423   <xs:attribute name="terminationType" type="xs:string"/>
```

```
424    <xs:attribute name="notificationPeriod" type="xs:string"/>
425    <xs:attribute name="notificationMethod" type="xs:string"/>
426    <xs:attribute name="fees" type="xs:double"/>
427  </xs:complexType>
428
429  </xs:schema>
```

B

Amazon Auto Scaling++

Notre objectif est de montrer la réutilisabilité de notre approche modulaire. Dans cette annexe, nous étendons le service AutoScaling de Amazon pour tenir compte des politiques de gestion (*BestPolicies*). Nous implémentons en particulier la politique de dégradation de fonctionnalité (*Application Elasticity*). Nous avons nommé cette extension *AAS*++ (Amazon Auto-Scaling++).

B.1 Modèle conceptuel

La Figure B.1 illustre le modèle conceptuel de *AAS*++. Amazon Auto Scaling est un service de dimensionnement automatique. Il permet de dimensionner les instances Amazon EC2. Amazon CloudWatch surveille les instances. Elle déclenche une alarme pour chaque condition vérifiée. Une telle condition reflète un objectif de niveau de service. Pour chaque alarme, Amazon Auto Scaling lance une action de dimensionnement (augmenter ou diminuer le nombre des instances). Nous proposons d'autres types d'actions sur l'application. Des telles actions permettent de basculer entre le mode normal et le mode dégradé.

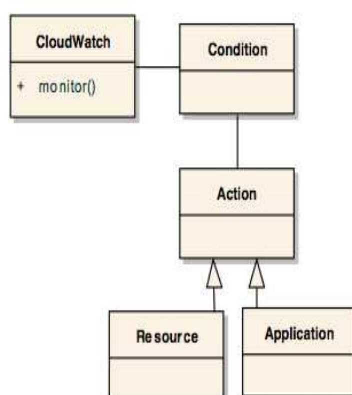


FIGURE B.1 – Méta-modèle AAS++

B.2 Elasticité de l'infrastructure

Nous rappelons que Amazon Auto Scaling est une solution basée sur les règles à base des seuils. Une règle est composée de deux parties : une condition et une action. Si la condition est vérifiée l'action sera lancée. Nous distinguons deux types de règle : les règles d'ajout et les règles de suppression. Le Tableau B.1 illustre un exemple des règles.

TABLE B.1 – Règles à base des seuils niveau IaaS

scale-out	Si m>$thr_{upper,m}$ pour $temps_{upper}$ alors ajouter k_{add} ;
scale-in	Si m<$thr_{lower,m}$ pour $temps_{lower}$ secondes alors supprimer k_{remove}

Où m est une métrique (CPU par exemple), $thr_{upper,m}$ et $thr_{lower,m}$ sont respectivement le seuil supérieur et le seuil inférieur de la métrique m, $temps_{upper}$ et $temps_{lower}$ sont respectivement deux périodes de temps qui définissent la durée pendant laquelle la condition doit être remplie pour déclencher une action de dimensionnement, k_{add} et k_{remove} sont respectivement la capacité à ajouter ou à supprimer.

B.3 Elasticité de l'architecture applicative

Nous proposons, également, des règles au niveau application pour gérer l'élasticité architecturale (dégradation de fonctionnalité). Le Tableau B.2 illustre la règle niveau application qui permet la dégradation de l'application. L'idée principale est de calibrer le paramètre $temps_{mode}$ d'une valeur inferieure à $temps_{upper}$. Après un dépassement de seuil de l'objectif de métrique m pendant $temps_{mode}$ minute, l'application passe en mode dégradé pour éviter de déclencher la règle niveau infrastructure. Il n'y a pas de règle pour le retour en mode normal. En effet, Le retour en mode normal sera assuré par les paramètres de la politique de dégradation à savoir : la durée de dégradation, la fréquence de dégradation et la fenêtre de temps.

TABLE B.2 – Règles à base des seuils niveau SaaS

scale-app	Si m>$thr_{upper,m}$ pour $temps_{mode}$ alors passer en mode degradé ;

Chaque action d'ajout en niveau infrastructure sera suivie par un déclenchement de mode dégradé. L'idée ici est d'absorber le démarrage. Une fois les instances activées, l'application passe au mode normal.

B.4 Implémentation

Le prototype *AAS++* est basé principalement sur des services Amazon à savoir principalement (voir Figure B.2) : Amazon EC2 (les instances), Amazon Cloud Watch (monitoring), Amazon Auto-scaling (service de dimensionnement automatique) et Amazon Elastic Load Balancer (répartiteur de charge).

L'extension (*App-Elasticity*) permet de gérer l'élasticité architecturale (dégradation de fonctionnalité). Elle est implémentée via une boucle de contrôle autonomique MAPE-K [Hor01]. Le gestionnaire autonome observe les informations remontées via CloudWatch. Il utilise ces données et sa connaissance interne (règle d'élasticité applicative) pour analyser , planifier et exécuter des actions (*scale-app*) pour gérer l'élastiscté au niveau application SaaS.

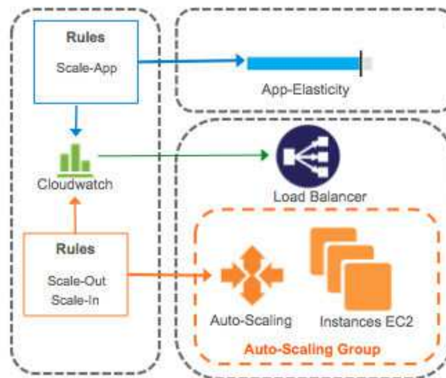


FIGURE B.2 – Prototype *AAS++*

l'université nantes angers le mans

PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes

Institut Mines-Télécom

Thèse de Doctorat

Yousri KOUKI

Approche dirigée par les contrats de niveaux de service pour la gestion de l'élasticité du "nuage"

SLA-driven Cloud Elasticity Management approach

Résumé

L'informatique en nuage révolutionne complètement la façon de gérer les ressources. Grâce à l'élasticité, les ressources peuvent être provisionnées en quelques minutes pour satisfaire un niveau de qualité de service (QdS) formalisé par un accord de niveau de service (SLA) entre les différents acteurs du nuage. Le principal défi des fournisseurs de services est de maintenir la satisfaction de leurs consommateurs tout en minimisant le coût de ces services. Du point de vue SaaS, ce défi peut être résolu d'une manière ad-hoc par l'allocation/-libération des ressources selon un ensemble de règles prédéfinies avec Amazon Auto Scaling par exemple. Cependant, implémenter finement ces règles d'élasticité n'est pas une tâche triviale. D'une part, la difficulté de profiler la performance d'un service compromet la précision de la planification des ressources. D'autre part, plusieurs paramètres doivent être pris en compte, tels que la multiplication des types de ressources, le temps non-négligeable d'initialisation de ressource et le modèle de facturation IaaS.
Cette thèse propose une solution complète pour la gestion des contrats de service du nuage. Nous introduisons CSLA (Cloud Service Level Agreement), un langage dédié à la définition de contrat de service en nuage. Il adresse finement les violations SLA via la dégradation fonctionnelle/QdS et des modèles de pénalité avancés. Nous proposons, ensuite, HybridScale un framework de dimensionnement automatique dirigé par les SLA. Il implémente l'élasticité de façon hybride : gestion réactive-proactive, dimensionnement vertical-horizontal et multi-couches (application-infrastructure). Notre solution est validée expérimentalement sur Amazon EC2.

Mots clés

Informatique en nuage, Elasticité, Contrat de niveau de service (SLA), Dimensionnement automatique, Gestion de capacité, Panification de capacité.

Abstract

Cloud computing promises to completely revolutionize the way to manage resources. Thanks to elasticity, resources can be provisioning within minutes to satisfy a required level of Quality of Service (QoS) formalized by Service Level Agreements (SLAs) between different Cloud actors. The main challenge of service providers is to maintain its consumer's satisfaction while minimizing the service costs due to resources fees. For example, from the SaaS point of view, this challenge can be achieved in ad-hoc manner by allocating/releasing resources based on a set of predefined rules as Amazon Auto Scaling implements it. However, doing it right –in a way that maintains end-users satisfaction while optimizing service cost– is not a trivial task. First, because of the difficulty to profile service performance, the accuracy of capacity planning may be compromised. Second, several parameters should be taken into account such as multiple resource types, non-ignorable resource initiation time and IaaS billing model.
For that purpose, we propose a complete solution for Cloud Service Level Management. We first introduce CSLA (Cloud Service Level Agreement), a specific language to describe SLA for Cloud services. It finely expresses SLA violations via functionality/QoS degradation and an advanced penalty model. Then, we propose HybridScale, an auto-scaling framework driven by SLA. It implements the Cloud elasticity in a triple hybrid way: reactive-proactive management, vertical-horizontal scaling at cross-layer (application-infrastructure). Our solution is experimentally validated on Amazon EC2.

Key Words

Cloud Computing, Elasticity, Service Level Agreement (SLA), Auto-Scaling, Capacity Management, Capacity Planning.