Contributions à la mise en place d'une infrastructure de Cloud Computing à large échelle

Jonathan Pastor

▶ To cite this version:

Jonathan Pastor. Contributions à la mise en place d'une infrastructure de Cloud Computing à large échelle. Autre [cs.OH]. Ecole des Mines de Nantes, 2016. Français. NNT : 2016EMNA0240 . tel-01416099

HAL Id: tel-01416099
https://theses.hal.science/tel-01416099

Submitted on 14 Dec 2016

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.

UNIVERSITE BRETAGNE LOIRE

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Jonathan PASTOR

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure des mines de Nantes
Label européen
sous le sceau de l'Université Bretagne Loire

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

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

Soutenue le 18 octobre 2016
Thèse n° : 2016 EMNA 0240

Contributions à la mise en place d'une infrastructure de Cloud Computing à large échelle.

JURY

Président :	**M. Mario SÜDHOLT**, Professeur, École des Mines de Nantes
Rapporteurs :	**M. Pierre SENS**, Professeur des Universités, LIP6
	M. Stéphane GENAUD, Professeur des Universités, ENSIIE
Examinateur :	**M. Thierry COUPAYE**, Directeur de domaine de recherche, Orange Labs
Directeur de thèse :	**M. Frédéric DESPREZ**, Directeur de recherche, Université de Grenoble
Co-directeur de thèse :	**M. Adrien LEBRE**, Chargé de recherche, INRIA

Remerciements

Je commence par remercier les membres de mon jury de thèse qui m'ont fait l'honneur d'être les juges de mes travaux. Merci à Pierre Sens et Stéphane Genaud d'avoir accepté d'être les rapporteurs de ma thèse.

Ensuite, je tiens à exprimer ma reconnaissance à mes directeurs de thèse, Adrien Lebre et Frédéric Desprez, qui m'ont encadré pendant ces quatre années. Je les remercie de m'avoir donné l'opportunité de travailler sur ce sujet de thèse, ainsi que pour leur encadrement plein de patience et de bienveillance. J'ai beaucoup appris à vos cotés ! Adrien, je te suis très reconnaissant de m'avoir fait confiance, de m'avoir poussé dans mes travaux et de m'avoir donné autant de ton temps !

Pendant mes travaux de doctorat, j'ai pu prendre part à différents projets où j'ai eu le plaisir de travailler avec des gens qui m'ont beaucoup appris. Tout d'abord, merci à Flavien qui, en fin de thèse, m'a guidé alors que je débutais la mienne. Ses conseils se sont révélés fort utiles, et je le remercie de m'avoir passé le relais sur DVMS. Merci à Mario qui, pendant notre collaboration sur VMPlaceS, m'aura transmis sa passion pour un certain éditeur de texte que j'utilise dorénavant chaque jour. Je remercie aussi Laurent, qui m'a beaucoup aidé dans mes expériences sur Grid'5000, et qui a développé des outils magiques qui m'ont fait gagner un temps précieux sur Grid'5000 (*Funk* mérite d'être dans les remerciements). De plus, je n'oublierai jamais la belle expérience que nous avons présenté ensemble à Lyon, et qui nous a permis de remporter un prix lors de l'école de printemps Grid'5000 en 2014 (après quelques nuits blanches !).

Au cours de ces quatre années, j'ai eu la chance d'évoluer dans un environnement agréable, entouré de gens inspirants. Merci aux habitués des tournois de *Street Fighter*, Ronan, Florent, Walid, Alexandre, Brice, Gustavo, Ignacio, Ziad, avec qui nous avons enchaîné les quarts de cercles et autres Shoryukens (dans le vide). Merci aux collègues du département informatique, Anthony, Mario, Hervé, Rémi, Nicolas, Pierre Maël, Frédérico, Massimo, Amine, Hammzah, et Tarzan avec qui nous avons souvent eu des discussions passionnées et passionnantes.

Enfin, je termine tout naturellement en remerciant mes amis (de France, de Chicago, et d'ailleurs) et ma famille, qui m'ont soutenu dans les moments de joies et les passages à vide. Je remercie en particulier ma famille pour son soutien, ainsi que pour avoir eu la patience de relire mon manuscrit de thèse.

Table des matières

I Introduction **13**

II Contexte de la thèse **19**

1 Avènement de l'informatique utilitaire **21**
1.1 Début de l'informatique moderne 21
1.2 Premières mises en réseau d'ordinateurs 22
1.3 Préhistoire du calcul utilitaire : les grilles de calcul 23
1.4 Avènement du Cloud Computing 23
1.5 Résumé 24

2 Qu'est-ce que le Cloud Computing ? **25**
2.1 La difficulté de définir le Cloud Computing 25
2.2 SPI : un modèle en couche pour le Cloud Computing 27
2.3 IaaS : La couche basse du Cloud Computing en détail 30
2.3.1 La virtualisation : la pierre angulaire du modèle IaaS 30
2.3.2 Vers une définition d'une architecture commune pour les systèmes IaaS ? 31
2.3.3 Revue des principaux systèmes IaaS 33
2.4 Résumé 38

3 Paysage des infrastructures IaaS et Edge Computing **41**
3.1 Splendeur et décadence des mégas centres de données 41
3.2 Fédérations de nuages 44
3.2.1 *Bursting* (Cloud hybride) 45
3.2.2 *Broker* 46
3.2.3 Aggregated 47
3.2.4 *Multitier* 48
3.3 Du modèle d'Edge Computing au Fog Computing 49
3.4 Comparaison des trois modèles de Cloud Computing 51
3.5 Résumé 56

4 Infrastructures IaaS massivement distribuées avec OpenStack **57**
4.1 Passage à l'échelle d'OpenStack 57
4.2 Distribution à plat 58

4.3 Distributions hiérarchiques 59
4.3.1 *Cells* 60
4.3.2 Cascading OpenStack / Tricircle 62
4.4 Résumé 64

III Contributions de la thèse **67**

5 Discovery : vers une infrastructure IaaS massivement distribuée **69**
5.1 Un réseau de nanos centres de données dans les points de présence 70
5.2 Le LUC-OS : un gestionnaire *IaaS* massivement distribué 71
5.3 Adapter OpenStack pour utiliser les bases de données non relationnelles . 72
5.4 Résumé 74

6 Support des bases clé/valeur dans Nova **77**
6.1 Accès aux bases de données dans Nova 78
6.2 Vers un support des bases de données clé/valeur dans Nova 79
6.3 Rome : un *ORM* pour bases clé/valeur 80
6.3.1 Interfaces logicielles inspirées par *SQLAlchemy* 80
6.3.2 Représentation des données et marshalling 81
6.3.3 Support des opérations de bases : création, sélection, modification, suppression 83
6.3.4 Support des jointures et des relations 85
6.3.5 Les sessions : implémentation à base de verrous distribués 87
6.3.6 Implémentation au-dessus de REDIS 88
6.3.7 Optimisations 88
6.4 Validation sur Grid'5000 91
6.4.1 Impact de l'utilisation de REDIS par rapport à MySQL 92
6.4.2 Scénarios multisites 96
6.4.3 Compatibilité avec les fonctions avancées de Nova 99
6.4.4 Premiers essais autour de la localité dans Nova 100
6.5 Résumé 101

7 Prise en compte de la localité réseau : le cas de DVMS **103**
7.1 Contexte 103
7.1.1 Ordonnancement de machines virtuelles pair-à-pair 103
7.1.2 DVMS : ordonnancement dynamique de machines virtuelles . . . 104
7.1.3 Overlay network et prise en compte de la localité 105
7.2 Contributions 106
7.2.1 Réseau d'overlay prenant en compte la localité 107
7.2.2 PeerActor : Une brique de base pour abstraire les réseaux d'overlay 108
7.3 Validation expérimentale sur Grid'5000 109
7.4 Résumé 112

8 Contributions à VMPlaceS : un simulateur pour algorithmes de placement de machines virtuelles **115**
8.1 VMPlaceS, un simulateur s'appuyant sur SimGrid 116
8.2 Fonctionnement de VMPlaceS 116
8.3 Validation expérimentale de VMPlaceS 117
8.3.1 Précision des simulations 118
8.3.2 Comparaison de trois algorithmes de placement 119
8.4 Travaux connexes 122
8.5 Résumé 124

IV Conclusions et perspectives **125**

9 Rappel des contributions **127**

10 Perspectives **129**
10.1 Perspectives à court terme 129
10.1.1 Rome 129
10.1.2 VMPlaceS et DVMS 131
10.2 Perspectives à long terme 132

V References **135**

Liste des tableaux

3.1 Comparaison multi-critère des différents modèles de déploiement du Cloud Computing . . . 55

6.1 Temps de réponse moyens aux requêtes API pour des déploiements monosite (en ms). . . . 93
6.2 Temps utilisés pour créer 500 VMs pour des déploiements monosite (en sec.). . . . 93
6.3 Quantité de données échangées sur le réseau (en Méga-octets). . . . 95
6.4 Temps utilisés pour créer 500 VMs avec une latence inter site de 10ms (en sec.). . . . 98
6.5 Temps utilisés pour créer 500 VMs avec une latence inter site de 50ms (en sec.). . . . 99
6.6 Quantité de données échangées sur le réseau (en Méga-octets.). . . . 100

7.1 Comparaison des taux de migrations intrasite (DVMS/Chord vs DVMS/LBO) . . . 112
7.2 Latences réseau mesurées entre les sites Grid'5000 utilisés pour les expériences . . . 112
7.3 Comparaison de données sur les partitions ISP (DVMS/Chord vs DVMS/LBO) . . . 112

Table des figures

2.1 Architecture type d'un système d'exploitation pour infrastructure de Cloud Computing. 32
2.2 Architecture du système OpenStack. 34
2.3 Architecture du gestionnaire Eucalyptus (extrait de [30]). 36
2.4 Architecture du projet OpenNebula (extrait de [32]). 38

3.1 Centres de données localisés à Quincy (Washington). 43
3.2 Schématisation d'une fédération reposant sur une architecture de type *Bursting* ([25]). 45
3.3 Schématisation d'une fédération reposant sur une architecture de type *Broker* ([25]). 46
3.4 Schématisation d'une fédération reposant sur une architecture de type *Aggregated* ([25]). 47
3.5 Schématisation d'une fédération reposant sur une architecture *Multitier* ([25]). 48
3.6 Illustration du modèle de Edge Computing. 51

4.1 Déploiement du CERN utilisant les Cells (extrait de [55]). 60
4.2 Architecture de la solution Cascading OpenStack (extrait de la documentation de cascading OpenStack). 62
4.3 Architecture de la solution Cascading OpenStack (extrait de la documentation de cascading OpenStack). 63

6.1 Accès à la base de données par les sous-services de Nova. 78
6.2 Accès à la base de données après l'ajout de Rome. 81
6.3 Illustration du processus de sérialisation. 82
6.4 Illustration du processus de désérialisation. 82
6.5 Étapes du processus de jointure dans *Rome*. 86
6.6 Illustration du fonctionnnement d'une référence paresseuse non chargée dans *Rome*. Il n'y a pas de valeur locale de disponible, la valeur est chargée depuis la base de données. 89
6.7 Illustration du fonctionnnement d'une référence paresseuse chargée dans *Rome*. Une valeur locale étant disponible, le chargement de la base de données est évité. 90
6.8 Illustration du fonctionnnement des index secondaires. 91

6.9 Illustration du déploiement fait avec un seul serveur de bases de données MySQL. 92
6.10 Illustration du déploiement fait avec une base de données REDIS distribuée. 93
6.11 Illustration du déploiement fait avec une base de données MySQL distribuée avec Galera. 94
6.12 Distribution statistique des temps de réponse à l'API de Nova (en ms.). . . 95
6.13 Quantité de données échangées en fonction du type de nœuds dans une configuration avec un unique nœud MySQL. 96
6.14 Quantité de données échangées en fonction du type de nœuds dans une configuration avec un cluster REDIS de 4 nœuds sans réplication de données. 97
6.15 Quantité de données échangées en fonction du type de nœuds dans une configuration avec un cluster REDIS de 4 nœuds avec réplication de données. 98

7.1 Illustration du fonctionnement de DVMS avec un cas comprenant 3 clusters (extrait de [9]). 105
7.2 Mécanisme d'apprentissage : (a) La vue initiale de chaque nœud est représentée par les lignes pointillées. En considérant ces vues, la spirale obtenue à partir du nœud node A est représentée par la ligne épaisse. Dans cet exemple, la spirale a permis à A et B de se découvrir. (b) Si B commence à construire une spirale, il va commencer par contacter A. La construction de la spirale permet aussi à E et B de se découvrir. (c) Si A se voit demandé de construire une autre spirale, celui-ci va montrer un degré plus important de conscience de la localité. 108
7.3 DVMS s'appuyant sur l'architecture en Peer actor. 109
7.4 Déploiement Grid'5000 utilisé pour les expériences. 111

8.1 Illustration du fonctionnement de VMPlaceS. 117
8.2 Comparaison des exécutions simulées (haut) et *in vivo* (bas). 119
8.3 Évolution de la charge de travail imposée à l'infrastructure. 121
8.4 Cumul des temps de surcharge des hôtes physiques, en fonction des algorithmes. 122
8.5 Illustration du détail des informations récoltées sur les surcharges des hôtes physiques (Hiérarchique déployé sur 1024 nœuds). 123

Introduction

L'essor d'Internet et le développement de la vision où les ressources informatiques sont considérées comme des ressources utilitaires ont conduit au succès du modèle de *Cloud Computing*, dans lequel des fournisseurs hébergent des services informatiques demandeurs en ressources de calcul et destinés à être consommés via Internet. La démocratisation d'Internet a permis à certains services web de connaître un succès planétaire et ces services ont rapidement nécessité de très grandes quantités de ressources de calculs produites au sein d'infrastructures de Cloud Computing, pour traiter les requêtes de leurs utilisateurs. Ces infrastructures de Cloud Computing sont majoritairement hébergées dans des centres de données toujours plus grands afin de réaliser des économies d'échelle. Ainsi, des dizaines de milliers d'ordinateurs (serveurs) sont regroupés dans de gros centres de données, dont les besoins en énergie et en refroidissement sont considérables.

Ces gros centres de données bénéficient de leurs propres installations électriques ainsi que d'infrastructures de climatisation. En plus de l'impact sur l'environnement, la consommation de ces centres de données en électricité et en refroidissement représente sur le long terme le poste de dépense le plus important, devant le coût d'achat de matériel informatique [1]. En conséquence, l'augmentation de la taille de ces gros centres de données a conduit à une croissance de ces des coûts, mais aussi à la création de nouvelles infrastructures pour permettre cet approvisionnement. Certains États américains ont misé sur la création de nouvelles centrales électriques, souvent polluantes (charbon), tandis que chez les gros acteurs du Cloud Computing la tendance est plutôt à approvisionner les centres de données en énergies renouvelables, au moyen de l'installation de fermes de panneaux solaires à proximité directe des centres de données (comme pour le centre de données d'Apple en Caroline du Nord).

La viabilité d'un tel modèle, où la production de ressources de calcul est concentrée dans un nombre restreint de centres de données toujours plus grands, peut être débattue sur certains aspects comme la tolérance aux pannes, notamment lors de catastrophes naturelles ou d'erreurs humaines. Ainsi, pendant les fêtes de Noël 2012, un des composants de la plate-forme de Cloud Computing d'Amazon (*Elastic Load Balancer*) a souffert d'un comportement erratique [2], conduisant à l'indisponibilité d'une partie des centres de données et de tous les services hébergés au sein de ces centres. En sus de l'impact sur Amazon, cet incident a aussi impacté ses clients tels que Netflix, un service de vidéo à la demande hébergé sur Amazon. En réponse à la perte de connectivité entre le réseau Internet et une partie de l'infrastructure Amazon, le service Netflix a vu sa qualité de service considérablement dégradée au cours de cette période. Une deuxième limitation majeure de ces mégas centres de données est qu'ils ne peuvent garantir aux services qui y sont hébergés d'être proches de tous les utilisateurs finaux. Ce phénomène est particulièrement vrai pour les services possédant une audience mondiale, où les utilisateurs sont géographiquement distribués sur toute la planète, comme c'est le cas pour les réseaux sociaux. On observe alors une sur-sollicitation des infrastructures réseaux Internet qui font la connexion entre les centres de données et les utilisateurs finaux [3].

Pour répondre aux problèmes inhérents à ce modèle de production des ressources de calcul concentrée dans quelques gros centres éloignés des utilisateurs finaux, la décentralisation d'un tel modèle doit être étudiée. Greenberg *et al.* [1] ont ainsi proposé de mettre en place un modèle où les ressources de calcul sont produites au sein d'un réseau

de micros centres de données éparpillés à travers un territoire. Cette approche permet de résoudre le problème de la distance avec les utilisateurs finaux, sans toutefois développer précisément la manière de mettre en place une telle infrastructure. Reprenant ce concept de micros centres de données géo-distribués, l'initiative *Discovery* [1] a proposé de s'appuyer sur les réseaux des opérateurs de télécommunication formant la dorsale Internet, en développant une infrastructure de calcul utilitaire distribuée, s'appuyant sur des micros centres de données co-localisés avec les points de présence des opérateurs de télécommunication.

Cependant, cette décentralisation amène inévitablement à la question du système responsable d'assurer le fonctionnement de cette infrastructure de micros centres de données géo-distribués. Les logiciels en charge de l'exploitation des infrastructures de Cloud Computing sont majoritairement pensés pour fonctionner au sein de centres de données possédant des propriétés particulières telles qu'un réseau très rapide et des ressources hautement disponibles. Ces logiciels ne sont pas, en l'état, prévus pour fonctionner dans des conditions où le réseau est moins rapide avec une forte latence entre serveurs et où les ressources ont un taux d'attrition (*churn*) élevé, ce qui est souvent le cas dans les infrastructures massivement distribuées [4].

La thèse défendue à travers ce manuscrit et plus généralement au cours de ce doctorat est la possibilité de déployer une infrastructure de Cloud Computing en périphérie du réseau et exploitée par un même gestionnaire massivement distribué s'appuyant sur OpenStack. Ce gestionnaire a pour but de transformer un ensemble de micros centres de données géographiquement répartis en une infrastructure de calcul utilitaire aux capacités équivalentes à celles d'un gros centre de données.

La principale nouveauté de ces travaux de doctorat est la révision des mécanismes internes du gestionnaire OpenStack (en particulier le service *Nova*) afin de permettre à un même système d'exploiter un réseau de micros centres de données, sans nécessiter l'utilisation d'intergiciels supervisant des infrastructures OpenStack indépendantes (voir la section 4.3) et sans avoir à redévelopper un gestionnaire d'infrastructure depuis zéro. Nous avons notamment mis en oeuvre *Rome*, une bibliothèque qui permet de faire fonctionner le service Nova avec une base de données de type clé/valeur et dont le fonctionnement a été validé expérimentalement sur la plate-forme Grid'5000. Une collaboration est en cours avec Orange pour étendre cette stratégie au service Neutron, et ces travaux ont fait l'objet de présentations lors du séminaire *Distributed Cloud Computing* à Dagstuhl en 2015 [5], lors du colloque *virtualisation/cloud* organisé au LIP6 en septembre 2015, lors du *Paris OpenSource Summit* en novembre 2015, et lors de l'*OpenStack Summit* (Austin 2016), et ont conduit à un la rédaction d'un rapport de recherche [6]. De plus, cet effort a facilité la mise en place d'une ADT *Mercury* dans le cadre du projet *Discovery*, qui vise à valoriser les prototypes de recherche au sein de la communauté OpenStack.

En parallèle de la contribution principale, deux contributions secondaires ont été réalisées. Tout d'abord l'amélioration de DVMS [7], un mécanisme de placement de machines virtuelles fonctionnant de manière complètement distribuée qui a été développé lors d'une précédente thèse au sein de l'équipe ASCOLA. L'objectif était d'étudier la possibilité de modifier DVMS afin qu'il puisse être utilisé dans un contexte multisite. Cette contribu-

[1] http://beyondtheclouds.github.io/

tion s'est faite en trois étapes, dont la première était la participation à la validation de DVMS au moyen de simulations. Les résultats de cette étape ont donné lieu à la publication d'un article à la conférence internationale ISPA 2013 [8]. Ensuite nous avons ajouté la notion de localité réseau dans DVMS, afin de privilégier les collaborations entre les nœuds proches. Cette contribution a donné lieu à une publication à la conférence internationale Euro-Par 2014 [9] et à un prix scientifique (*Large scale Challenge* de l'École de printemps Grid'5000 en 2014). Ce travail a permis de mettre en avant le fait qu'ajouter la notion de localité à un algorithme massivement distribué était une piste prometteuse pour améliorer son efficacité et avoir un meilleur passage à l'échelle. Ce principe est intéressant, car pouvant être appliqué sur prototype développé dans le cadre de la contribution principale.

Enfin j'ai contribué au développement logiciel de VMPlaceS, un simulateur permettant de comparer de manière précise et équitable différents algorithmes de placement de machines virtuelles. Ce projet a fait l'objet d'une publication dans la conférence internationale Euro-Par 2015 [10].

Ce manuscrit de thèse est organisé en trois parties. La première partie se concentre sur la définition du paradigme de Cloud Computing. Le premier chapitre s'intéresse à l'avènement de la notion d'informatique utilitaire. Dans le chapitre 2, nous présentons le contexte de la thèse. Les éléments constituants l'informatique utilitaire seront introduits en commençant par une introduction historique de ce qui a mené à l'avènement de l'informatique utilitaire sous sa forme moderne : le Cloud Computing. Ensuite, dans le chapitre 3 nous nous intéressons à l'étude des efforts qui ont été entrepris pour le définir, pour dans un dernier temps nous intéresser à ses mises en œuvre actuelles. Le chapitre 4 continue cet effort en se focalisant sur les techniques actuelles de distribution des infrastructures de Cloud Computing sur le gestionnaire *OpenStack*.

La seconde partie présente les travaux réalisés au cours de cette thèse. Le chapitre 5 propose la mise en place d'une infrastructure de Cloud Computing et propose une architecture s'appuyant sur le gestionnaire OpenStack, tout en identifiant ses limitations et en proposant des solutions à celles-ci. Dans le chapitre 6 nous expliquons la mise en place d'une des propositions précédentes en remplaçant la base de données relationnelle utilisée par les services d'OpenStack, par une base de données non relationnelle. Nous appliquons cette approche au service Nova et étudions expérimentalement ses effets au moyen d'expériences sur Grid'5000. Le chapitre 7 s'intéresse à la prise en compte de la localité dans l'algorithme de placement de machines virtuelles DVMS, afin d'améliorer la qualité de ses décisions de placement. Le chapitre 8 présente les travaux effectués pour mettre en place le simulateur VMPlaceS permettant de réaliser des comparaisons équitables d'algorithmes de placement de machines virtuelles au moyen de simulations précises.

Enfin, dans la troisième partie, nous concluons et proposons plusieurs perspectives qui permettraient de poursuivre les travaux entrepris au cours de cette thèse. Nous rappelons d'abord les contributions de cette thèse, puis proposons des perspectives à court terme des contributions réalisées, puis enfin nous donnons des perspectives à plus long terme concernant les infrastructures de Cloud Computing massivement distribuées.

Contexte de la thèse

1

Avènement de l'informatique utilitaire

Dans ce chapitre nous donnons quelques éléments historiques qui permettront de comprendre l'apparition et l'évolution de l'informatique utilitaire. Nous commençons avec l'apparition de la notion de calcul et le passage des ordinateurs mécaniques aux premiers ordinateurs électroniques qui permettront d'ouvrir l'accès à des capacités de calcul jusqu'ici inégalées. Ensuite, nous évoquons le projet *Arpanet*, le premier réseau à grande échelle utilisant la communication par paquet, apparaissant comme le précurseur du réseau mondial Internet. Cette mise en réseau via *Arpanet* permet d'envisager que les ressources informatiques peuvent être consommées sur le même modèle que celui des ressources utilitaires que sont l'électricité, l'eau et le gaz. Nous détaillons cette étape, et évoquons l'apparition des grilles de calcul ainsi que leur essor dans les années 90. Enfin, nous terminons sur le modèle de *Cloud Computing* qui connaît son avènement grâce à la démocratisation d'Internet et à la maturité des grilles de calculs.

1.1 Début de l'informatique moderne

L'informatique moderne nait entre la Première et la Seconde Guerre mondiale, avec des travaux précurseurs comme ceux d'Alan Turing qui proposa en 1936 un modèle théorique pour une machine étant capable de résoudre tout problème exprimable sous la forme d'un algorithme [11] (système Turing complet). En parallèle, Konrad Zuse développa les mécano-ordinateurs de la série Z : le Z3 est en 1941 le premier ordinateur Turing complet reposant sur un fonctionnement mécanique. Le premier ordinateur entièrement électronique Turing complet [12] est l'ENIAC qui apparait en 1946.

Les premiers ordinateurs électroniques fonctionnaient grâce à des circuits logiques constitués d'assemblages de tubes à vide, servant d'interrupteurs électriques. Les tubes à vide offraient l'avantage de ne pas avoir de partie mécanique, ce qui permettait un fonctionnement beaucoup plus rapide que les circuits électriques contenant des interrupteurs

mécaniques. Dans son principe de fonctionnement, un tube à vide contient deux éléments : une cathode (filament) et une anode (plaque électrique) ; quand la cathode est froide, le tube à vide n'est pas conducteur ; quand la cathode est chauffée, le tube à vide fait circuler du courant. Ce principe de fonctionnement conduit cependant à de forts dégagements de chaleur, ce qui a l'inconvénient de rendre difficile la construction d'ordinateurs complexes contenant beaucoup de tubes à vide.

La découverte du transistor en 1947 a permis d'envisager une alternative à l'utilisation des tubes à vide dans la conception de circuits logiques. Le transistor va avoir un impact majeur sur l'électronique, car il est individuellement plus fiable que le tube à vide, peut être produit en masse et donc se révéler rapidement moins cher. Ces raisons font que le transistor permet d'envisager la construction de circuits logiques de plus en plus complexes [13]. Son invention va permettre à des générations d'ordinateurs centraux de se succéder, chaque nouvelle génération devenant largement plus puissante que la génération précédente. Cependant, ces superordinateurs centralisés vont rester pendant longtemps très onéreux et leurs utilisations se limiteront aux organisations gouvernementales ainsi qu'aux grandes entreprises.

1.2 Premières mises en réseau d'ordinateurs

Pendant la guerre froide opposant le bloc occidental au bloc soviétique, les recherches concernant les réseaux de communication ont connu un grand essor, étant considérées comme essentielles pour assurer le maintien des communications en cas de bombardement nucléaire (causant un blackout des hautes fréquences qui était le moyen de communication de l'époque). En réaction au lancement du premier satellite soviétique Spoutnik, l'agence ARPA (*Advanced Research Projects Agency*) est créée en 1958. En 1959, les travaux de Paul Baran et Donald Davies conduisent à la proposition de communication par échange de paquets sur un réseau, l'ARPA financera alors un projet de réseau utilisant cette technique de communication (ARPANET) où des ordinateurs constitueraient les nœuds du réseau et communiqueraient entre eux par échange de paquets, ceux-ci étant transmis au moyen du réseau téléphonique [14].

En 1969, le Network Working Group (NWG) commence à travailler à l'élaboration de standards permettant d'organiser la mise en réseau d'équipements informatiques hétérogènes. Ces derniers devraient communiquer par échange de messages transmis par un réseau de nœuds IMP (*Interface Message Processor*) qui deviendront plus tard les futurs routeurs du réseau Internet. Ce groupe de travail a aussi fourni des propositions de protocoles réseau en guise d'exemple d'application d'ARPANET : les protocoles NCP, FTP ainsi que les prémisses de Telnet seront proposés pour illustrer le potentiel d'ARPANET. En octobre 1971, les représentants de chacun des sites impliqués dans le projet font un premier test réel de l'infrastructure où sur chacun des sites, un représentant se connecte aux autres sites. Grâce au réseau ARPANET, des ordinateurs aux matériels hétérogènes souvent incompatibles ont pu être mis en commun, offrant aux utilisateurs la possibilité de lancer des calculs sur toutes les machines reliées au réseau, le tout géré par un système de temps de calcul partagé. Il était alors possible pour l'utilisateur d'un site d'accéder à plus de puissance que celle disponible sur son propre site.

1.3 Préhistoire du calcul utilitaire : les grilles de calcul

Le calcul utilitaire prend sa source au début des systèmes à temps de calcul partagé [15] qui apparaissent au début des années 60. En 1961, John McCarthy est le premier à envisager le calcul utilitaire, lors d'un discours à l'occasion du centenaire du MIT en 1961 :

> *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, 1961*

Quand le premier nœud ARPANET est mis en place en 1969, Leonard Kleinrock imagine que la mise en réseaux d'ordinateurs pourrait conduire à l'apparition d'une informatique utilitaire, où la consommation de ressources informatiques se ferait à la demande, de la même manière que des ressources telles que l'eau et l'électricité sont consommées. L'analogie va plus loin puisque de la même manière qu'il existe des réseaux de centrales électriques formant des grilles électriques, on verrait l'apparition de grilles de calcul composées de centres de production de ressources informatiques.

L'avènement de la micro-informatique au cours des années 80 permet la démocratisation du calcul informatique : le commun des mortels accède à une puissance de calcul jusque là réservée aux grandes organisations. Les micro-ordinateurs étant certes moins puissants que les superordinateurs, leur faible prix fera qu'ils se répandront à une très grande vitesse, et leur mise en réseaux ouvre la voie vers des infrastructures reposant sur des micro-ordinateurs dont le potentiel de puissance de calcul rattrape celui des superordinateurs. Le projet Condor est lancé en 1984 dans le but de fédérer des ressources de calcul éparpillées sur plusieurs micro-ordinateurs, permettant à ses utilisateurs d'exécuter des tâches à distance [16].

Au cours des années 90, les ordinateurs personnels (*Personal Computer* - PC) connaissent un immense succès et sont progressivement interconnectés via Internet, ce qui amène l'apparition du paradigme de "Desktop Grid" : celui-ci consiste en la distribution de calculs sur une fédération de micro-ordinateurs connectés à Internet, chaque ordinateur hébergeant un intergiciel (*middleware*) qui reçoit des tâches ; ces tâches représentent des calculs, et sont ensuite exécutées en exploitant la puissance de calcul inutilisée des ordinateurs faisant partie de la fédération. Ce paradigme a connu un certain succès au cours des années 90 avec des projets comme *SETI*, *BOINC*, et a inspiré le récent système *BITCOIN*.

1.4 Avènement du Cloud Computing

Les années 90 voient l'essor du réseau mondial Internet ainsi que l'apparition des premiers sites web possédant une audience mondiale. Certains d'entre eux vont voir leurs bases d'utilisateurs croître à des niveaux nécessitant la mise en place d'infrastructures capables de distribuer l'énorme charge de travail incidente sur un grand nombre de serveurs web, chargés de répondre aux requêtes des utilisateurs. Les premiers centres de données font leurs apparitions et vont regrouper des ordinateurs absorbant la charge de travail

des gros sites web. Cette concentration d'ordinateurs permettra de mettre en place des économies d'échelle sur les équipements électriques, le refroidissement et le personnel. Plus ces centres de données sont gros, plus les économies d'échelle sont conséquentes. Certaines entreprises vont mettre en place leurs propres centres de données dédiés à l'hébergement de leurs services, tandis que d'autres entreprises feront le choix de sous-traiter l'hébergement de leurs services dans des centres de données appartenant à des entreprises spécialisées dans l'hébergement. Rapidement émergera le modèle de “Cloud Computing”, qui connaitra un essor ininterrompu depuis la fin du 20e siècle.

1.5 Résumé

L'informatique moderne nait entre les deux guerres mondiales, alors que la notion de calcul se développe et les premiers ordinateurs font leur apparition. En 1947, le transistor est inventé, ce qui engendrera une croissance continue de la puissance des ordinateurs. Jusque dans les années 1960s, le modèle des super ordinateurs centralisés domine. En 1969, l'agence ARPA finance le projet ARPANET qui vise à mettre en place un réseau d'ordinateurs à l'échelle des USA, ce réseau sera inauguré en 1971. La perspective de mettre en réseau les ordinateurs, comme on a mis en réseau les centrales électriques, laisse présager que le calcul deviendrait une ressource utilitaire comme l'électricité ou le téléphone : celui-ci serait produit à distance et consommé à la demande. De cette vue de l'esprit découlera le modèle des grilles de calcul. Ainsi l'avènement de la micro-informatique bon marché va permettre de connecter un nombre élevé d'ordinateurs grâce à des réseaux, et ces ordinateurs seront mis à la disposition des utilisateurs. Ce modèle connaitra un certain succès dans les années 1990. Au début du 21e siècle, le succès des grilles de calcul et du réseau Internet inspirera le modèle de *Cloud Computing*.

2

Qu'est-ce que le Cloud Computing ?

Dans le chapitre précédent nous avons vu comment progressivement l'informatique est passée de calculs effectués sur un seul super-ordinateurs à des traitements distribués sur des réseaux de machines, permettant d'accéder à toujours plus de puissance. Dans ce chapitre nous détaillons le modèle de *Cloud Computing* qui permet à des demandeurs en ressources informatiques d'y accéder à la demande. Ce modèle a rencontré un fort succès et est utilisé par de nombreux acteurs très divers, aboutissant à une multitude de visions de ce qu'est le *Cloud Computing*. Ainsi nous commençons ce chapitre par la difficulté à donner une définition qui satisfait toutes les visions précédemment évoquées. Ensuite, nous nous intéressons aux efforts de classification des incarnations de ce modèle, en nous attardant particulièrement sur la classification *SPI* et ses dérivées. Enfin, nous terminons sur la couche basse de la classification *SPI* en mettant notamment l'accent sur les aspects architecturaux des principaux gestionnaires d'infrastructures de *Cloud Computing*, en cherchant à identifier ce qu'ils peuvent avoir en commun.

2.1 La difficulté de définir le Cloud Computing

L'essor des services à l'échelle du Web ainsi que les besoins pour traiter des données de plus en plus complexes ont conduit au succès du paradigme de Cloud Computing où des fournisseurs exposent une collection de ressources informatiques (calcul, stockage, capacité réseau, ...) au moyen d'Internet, permettant aux clients de les consommer à distance. Ces ressources informatiques sont facturées sur le mode du *Payez au fur et à mesure ce que vous consommez* (*Pay As You Go*), c'est-à-dire que l'utilisateur ne paie que les ressources qu'il utilise sur une durée donnée [17], soit le même mode de facturation des ressources utilitaires (électricité, eau, gaz, ...). Ce modèle privilégie des relations de sous-traitance où un acteur peut déléguer l'hébergement de ressources informatiques ou de services à un tiers, en s'accordant avec lui sur un niveau de service à respecter, moyen-

nant des pénalités financières en cas de non-respect. Ainsi le client peut limiter les coûts d'acquisition et d'entretien matériel et logiciel de ses services, mais aussi d'externaliser une partie des risques liés à l'infrastructure aux fournisseurs. Rapidement, des services de Cloud Computing s'appuient sur d'autres services, ce qui aboutira à des architectures de services de plus en plus complexes. Netflix en est un exemple, car ses utilisateurs consomment des vidéos qui leur sont distribuées en tirant avantage de l'infrastructure d'Amazon. Bien qu'Amazon et Netflix soient deux acteurs du Cloud Computing, ils fournissent des services de natures différentes.

Bien que le Cloud Computing s'appuie sur une vision proche de celle des grilles de calcul, ces deux modèles divergent sur un certain nombre de points. En premier, une grille de calcul mise sur l'agrégation de ressources issues de plusieurs organisations afin de proposer une infrastructure commune plus grande, tandis que le modèle de Cloud Computing fait souvent l'impasse sur cette notion de partage en privilégiant des infrastructures impliquant un seul fournisseur : ainsi l'infrastructure reste isolée des autres infrastructures appartenant à des fournisseurs tiers. Tandis que les acteurs impliqués dans les grilles de calculs ont fait un grand effort pour encourager la mise en place et l'adoption de standards pour favoriser l'interconnexion de plusieurs grilles de calcul afin de proposer l'accès à une quantité de ressources plus importante, les tentatives pour aboutir à une standardisation concertée des acteurs du Cloud Computing sont restées infructueuses, conduisant plutôt à l'adoption de normes considérées comme standard "de facto" telles que l'API de plateforme *Amazon*.

Malgré ces quelques différences, le modèle des grilles de calcul et le modèle du Cloud Computing se rejoignent sur le fait qu'ils visent à faciliter la mise à disposition pour l'utilisateur de collections de ressources informatiques issues de l'agrégation d'ordinateurs hétérogènes connectés en réseau. Ces similarités conduisent à une certaine confusion quand il s'agit de qualifier ce qui appartient à la catégorie des grilles de calcul et ce qui fait partie du domaine du Cloud Computing. De la même manière que les grilles de calcul, les infrastructures de Cloud Computing sont utilisées par des acteurs techniquement très variés de l'informatique (virtualisation, réseaux, stockage, ...) ou par des acteurs issus d'autres disciplines que l'informatique et appliquant les infrastructures à leurs disciplines (étude énergétique, économie, sociologie, ...).

Une étude [18] a mis en évidence que cette particularité expliquerait l'absence d'une définition claire et unanimement admise de ce qu'est le Cloud Computing par ses propres acteurs. Ce phénomène prend sa source dans le fait qu'il englobe un large spectre d'applications, rendant difficile une définition en termes généraux permettant de toucher tout ce qui appartient au Cloud Computing. Un acteur du Cloud Computing aura naturellement tendance à fournir une définition colorée par sa propre sensibilité, expliquant la multitude de définitions relevées par les auteurs.

Proposant de résoudre cette absence d'une définition communément acceptée du Cloud Computing, [18] a proposé de reprendre la multitude de définitions et d'en agréger les éléments qui faisaient consensus pour avoir une définition générale suffisamment large pour satisfaire toutes les sensibilités du Cloud Computing :

> *Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources*

> *can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a payper-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs – Définition fournie par les auteurs de [18].*

Cette définition est intéressante, car elle met l'accent sur le concept de "contrat de garantie de la qualité de service" (*Service Level Agreement* - SLA), ce qui souligne l'importance de la sous-traitance dans le modèle de Cloud Computing. Ainsi, tout le modèle semble reposer sur le fait qu'un client va consommer les ressources d'un fournisseur, et ce même fournisseur garantira une qualité de prestation à la hauteur des moyens investis par le client. Il est courant pour un service logiciel hébergé dans le Cloud d'impliquer plusieurs niveaux de sous-traitance sur plusieurs niveaux de sous-traitance. Dropbox en est un exemple, car ses utilisateurs délèguent à Dropbox le stockage de leurs fichiers qui à son tour a délégué l'hébergement de son infrastructure de stockage à Amazon.

Face à l'abondance et à la variété des relations de sous-traitance dans le Cloud Computing, certains ont tenté d'établir une classification étudiant les relations de dépendance qui pouvaient exister entre les fournisseurs de services et leurs clients, en rapprochant les services ayant des caractéristiques proches. De plus, il est courant que les services à haut niveau d'abstraction sous-traitent une partie des fonctions à bas niveau d'abstraction à d'autres services. Ces éléments ont conduit à un besoin de classification fournissant des critères pour classer les services de Cloud Computing, et ainsi regrouper dans un même ensemble les services partageant des caractéristiques semblables. Une telle classification pourrait permettre une meilleure compréhension du paysage du Cloud Computing (acteurs, services, technologies, ...) et ainsi de permettre la discussion entre des acteurs de contextes différents, qu'ils soient académiques ou issus de l'industrie, et quelques soient leurs points d'intérêts (scientifique, technique, économique, ...).

La classification la plus connue est la classification **SPI** qui classe un service parmi trois couches (*Software*, *Platform* et *Infrastructure*) en fonction du degré d'abstraction qu'il introduit à l'utilisateur. Bien qu'il existe d'autres classifications dérivées telles que UCSB-IBM et la classification de Jackson [19], seule la classification SPI sera détaillée dans cette partie.

2.2 SPI : un modèle en couche pour le Cloud Computing

La classification **SPI** [20] propose de situer un service de Cloud Computing au sein de trois couches *Infrastructure*, *Platform*, *Software*, cet ordre correspondant à un degré d'abstraction croissant de l'infrastructure sous-jacente. Dans ce modèle, les services appartenant aux couches supérieures font abstraction des fonctions relevant des couches inférieures : cela correspond à une observation de la réalité où il est courant qu'un service s'appuie sur d'autres services appartenant à des couches plus basses dans la classification. La suite de cette partie décrit les différentes couches constitutives du modèle **SPI**, allant des couches d'abstraction les plus hautes vers les plus basses, permettant ainsi de mettre en évidence le rôle des couches proches de l'infrastructure.

- ***Software as a Service* (SaaS)**

La couche haute *Software as a Service* (SaaS) regroupe des services rendus accessibles aux utilisateurs finaux au moyen d'un client léger (application web, application mobile, client bureau, ...). Ces services sont hébergés à distance : La majeure partie de leur code source (en particulier la partie métier), est exécutée à distance sur les infrastructures des fournisseurs de services.

Les avantages d'un tel modèle de service sont nombreux, comme le fait que la complexité liée au déploiement d'un service ainsi que sa maintenance sont masquées à l'utilisateur final. Les exigences en matière de matériel informatique sont aussi plus basses, car une partie de l'exécution du service est délocalisée sur l'infrastructure distante appartenant au fournisseur. L'utilisateur final bénéficie d'une garantie de qualité de service sans avoir à investir dans du matériel informatique spécialisé et performant, et en cas de pannes le contrat de garantie prévoit les indemnités que le fournisseur versera à l'utilisateur.

L'exécution de la partie métier du service se faisant sur l'infrastructure du fournisseur de service, il est possible de mettre à jour le code du service sans que l'utilisateur ne le voie, ce qui permet d'avoir un rythme de mises à jour du service plus important sans que cela soit préjudiciable pour l'expérience de l'utilisateur. Cependant, toutes les données des utilisateurs se trouvant sur l'infrastructure, en cas d'intrusion celles-ci se retrouvent exposées, de même qu'en cas de panne de l'infrastructure, il est fréquent que le service devienne totalement inaccessible pour les utilisateurs.

Les exemples connus de SaaS sont le service de messagerie Gmail (Google), le réseau social Facebook et l'outil de gestion de la relation client (CRM) Salesforces.

- ***Platform as a Service* (PaaS)**

La couche intermédiaire *Platform as a Service* regroupe des services qui ciblent les concepteurs de services logiciels. Les fournisseurs de *PaaS* fournissent des plates-formes logicielles aux concepteurs de service, qui sont livrées avec des interfaces de programmation dédiées (API) pouvant accueillir et exécuter des services de Cloud Computing. En utilisant les services de la couche PaaS, le travail des concepteurs est simplifié, car une grande partie de la complexité liée à la prise en main des aspects liés à l'infrastructure est masquée derrière l'API exposée par le fournisseur PaaS. De cette manière, le processus de développement d'un service logiciel est simplifié, car un concepteur qui utilise une plateforme *PaaS* ne fait que fournir du code source au fournisseur *PaaS*, qui en retour s'occupe de le déployer sous la forme d'un service, et de le rendre accessible aux utilisateurs au moyen d'un client léger.

Bien que le modèle de service *PaaS* réduise la complexité lors du développement d'un service, le code source développé doit néanmoins répondre à des contraintes dictées par les choix technologiques du fournisseur *PaaS*, telles que le support d'un nombre limité de langages de programmation, l'obligation d'utiliser certaines bibliothèques ou certains serveurs de bases de données. Ces contraintes sont souvent nécessaires pour garantir au fournisseur de *PaaS* une intégration plus facile entre les services des utilisateurs et les technologies utilisées par son infrastructure, rendant plus aisé l'hébergement de ceux-ci.

Enfin, bien que les éléments techniques composant les couches inférieures (notamment la partie *IaaS*) ne soient pas directement accessibles à un concepteur de service

utilisant une plate-forme *PaaS*, cette dernière est capable de faire des ajustements concernant les quantités de ressources qui lui sont allouées, afin de rester adaptées à la charge de travail du service qu'elle héberge. De même, il est courant que les fournisseurs *PaaS* proposent des interfaces logicielles (*APIs*) à leurs utilisateurs, permettant à ces derniers de modifier les paramètres associés à l'environnement d'exécution, notamment la puissance de calcul allouée à un service.

Parmi les principaux fournisseurs de ressources PaaS, figurent des acteurs comme Amazon (Amazon Web Service), Google (App Engine), Microsoft (Azure) et Salesforce (Heroku).

- ***Infrastructure as a Service* (IaaS)**

La couche basse *Infrastructure as a Service* (IaaS) regroupe les services qui fournissent des ressources de calcul sous la forme de machines virtuelles (VMs) aux utilisateurs. Ces machines virtuelles sont hébergées sur des serveurs physiques localisés dans des centres de données (*datacenters*) appartenant au fournisseur *IaaS*. L'utilisation des techniques de virtualisation procure notamment l'avantage de pouvoir consolider le degré d'utilisation des serveurs physiques en hébergeant plusieurs VMs sur un même serveur physique, ce qui permet d'améliorer le taux de rentabilité d'une infrastructure. De plus, les techniques de virtualisation permettent d'avoir des propriétés d'isolation empêchant les interférences entre des machines virtuelles appartenant à différents utilisateurs.

Lors de la fourniture d'une machine virtuelle, l'utilisateur a les mêmes possibilités d'utilisation que s'il avait affaire à une machine physique. Ainsi, il peut y déployer un système d'exploitation (OS) parmi ceux proposés par le fournisseur *IaaS* (certains fournisseurs laissent la possibilité d'installer un OS personnalisé), modifier les fichiers de configuration du serveur et y installer ses logiciels et les bibliothèques de son choix.

Il est courant que le fournisseur *IaaS* propose, en complément de l'approvisionnement en machines virtuelles, des services additionnels tels que la fourniture et l'hébergement d'images pour machines virtuelles, des ressources de stockage supplémentaires, des fonctions liées à la communication inter-VMs comme les réseaux virtuels, ou des garanties supplémentaires concernant la qualité de service (QoS) fournie pour l'hébergement de VMs.

Enfin, les infrastructures *IaaS* ont été popularisées sous le terme de "Cloud" (nuage), et les déploiements *IaaS* sont fréquemment classifiés en fonction de leurs degrés de souveraineté et de la nature des restrictions d'accès aux utilisateurs extérieurs au fournisseur *IaaS* :

- **Cloud public** (*Public Cloud*) : Un Cloud public est une infrastructure *IaaS* permettant l'accès à tout utilisateur, qu'il appartienne ou non à l'organisation propriétaire de l'infrastructure. Les usages de l'infrastructure sont facturés par le propriétaire. Ce type de déploiement correspond aux fournisseurs *IaaS* commerciaux tels qu'Amazon avec son service EC2 (*Elastic Compute*) ou au service de VPS (*Virtual Private Server*) proposé par OVH.

- **Cloud privé** (*Private Cloud*) : Un Cloud privé est une infrastructure *IaaS* appartenant à une organisation et dont l'accès est limité aux membres de cette organisation.

Ce type de déploiements est généralement propre à une entreprise. Dans le cas de certains *Private Cloud* où les ressources venaient à manquer pour satisfaire les besoins des utilisateurs, il y a la possibilité de déléguer le surplus de demandes à des infrastructures tierces publiques : on parle alors d'**Hybrid Cloud**.

- **Cloud communautaire** (*Community Cloud*) : Un Cloud communautaire est une infrastructure *IaaS* ouverte à une communauté d'utilisateurs : à partir du moment où un utilisateur est membre de l'organisation en charge de l'infrastructure, celui-ci peut exploiter les ressources de l'infrastructure communautaire. Ce modèle reprend de nombreux aspects du modèle de grille de calcul communautaire, et on le retrouve fréquemment dans le cadre des infrastructures *IaaS* dédiées à une utilisation scientifique. Ce modèle est à mi-chemin entre les Cloud privés et les Cloud publics.

2.3 IaaS : La couche basse du Cloud Computing en détail

La couche *Infrastructure as a Service* du modèle de Cloud Computing a connu un fort succès au cours des années 2000s : de nombreux acteurs ont développé leur propre solution et des projets de logiciels libres ont émergé permettant à des institutions de mettre en place leurs propres plates-formes IaaS. Chacune de ces solutions reposent sur les principes de programmation, qui étaient à la mode au moment où elle a émergé, comme les architectures à bases de services web (Eucalyptus), l'utilisation de communications à base d'échanges de messages sur un bus (OpenStack, CloudStack, OpenNebula) ou l'utilisation d'une structure hiérarchique pour organiser ses nœuds de calcul (Eucalyptus). Toutes les solutions *IaaS* s'accordent néanmoins sur l'usage de la virtualisation comme moyen de fournir en toute flexibilité des ressources de calculs à leurs utilisateurs.

2.3.1 La virtualisation : la pierre angulaire du modèle IaaS

La virtualisation [21] est une technique permettant de simuler le comportement d'une ressource physique au moyen d'un logiciel. Les **machines virtuelles**, apparues dans les années 1960s, sont l'application des techniques de virtualisation aux ordinateurs. Une machine virtuelle est un programme qui simule un ordinateur complet qui hébergera un système d'exploitation classique et bénéficiera des mêmes fonctionnalités (CPU, stockage, réseau, ...) qu'un ordinateur physique. Une machine virtuelle n'étant qu'un programme informatique, celle-ci est hébergée sur une machine physique (sur certains matériels, il est possible de faire fonctionner une machine virtuelle dans une autre machine virtuelle).

Un des principaux avantages des machines virtuelles est qu'elles permettent une abstraction du matériel chargé de l'hébergement, ce qui ouvre la voie à l'exécution de services informatiques sur du matériel hétérogène. De plus, elles permettent d'assurer l'isolation entre des processus exécutés au sein de plusieurs machines virtuelles étanches, ce qui est une base pour établir des propriétés de sécurité entre services appartenant à différents clients d'un même fournisseur d'infrastructure. Enfin, le fait que plusieurs machines virtuelles puissent être exécutées au sein d'une même machine physique ouvre la voie aux

mécanismes de consolidation [22] permettant d'augmenter le taux d'utilisation des machines physiques.

Bien que les machines virtuelles soient la technique de virtualisation la plus populaire au sein des systèmes *IaaS*, elles nécessitent l'installation d'un système d'exploitation complet par machine virtuelle. Des techniques alternatives, à l'empreinte sur l'hôte plus légère, ont connu un certain essor, comme les conteneurs (***Containers***) dont l'usage s'est popularisé. Les conteneurs tirent parti des mécanismes d'isolation entre les processus (via *chroot*) des systèmes d'exploitation modernes pour donner l'impression que des services sont exécutés sur des systèmes d'exploitation différents, alors qu'ils sont en fait exécutés dans des environnements différents, mais sur le même système appartenant à l'hôte. Grâce au succès du projet Docker [23], les conteneurs ont considérablement gagné en visibilité, et commencent à être pris en compte par les gestionnaires *IaaS* alors qu'ils étaient jusqu'alors utilisés par les fournisseurs *PaaS*.

2.3.2 Vers une définition d'une architecture commune pour les systèmes IaaS ?

De nombreux projets de gestionnaires *IaaS* ont émergé au début du 21e siècle. Bien que ces projets se soient reposés sur les technologies qui étaient en vogue au moment de leur démarrage (SOA, RPC, services web, ...), ceux-ci ont beaucoup de points communs architecturaux. Un premier élément expliquant ces similarités architecturales est que ces gestionnaires remplissent des fonctions qui sont très proches : exploiter une infrastructure fournissant de la puissance de calcul sous la forme de machines virtuelles. Le second élément expliquant ces similarités architecturales est que la plate-forme *Amazon EC2* dominant le marché des infrastructures de Cloud Computing, ses APIs se sont imposées comme des standards *de facto*, inspirant ainsi ses concurrents. De nombreux gestionnaires *IaaS* utilisent pour la plupart des terminologies proches (*instances*, *compute nodes*, *availability zones*) de celles d'*Amazon EC2*, principalement pour des raisons de compatibilité (pour éviter de verrouiller l'utilisateur sur *Amazon EC2* et de changer de plate-forme) et ne pas *réinventer la roue*. Ainsi, leurs architectures logicielles ont été composées autour de la gestion des mêmes ressources que celles fournies par Amazon (machines virtuelles, réseaux, stockage, ...), en gardant à l'esprit les fonctions assurées par la plate-forme *Amazon EC2*.

Sur la même lancée que la plate-forme *Amazon EC2*, les projets communautaires visant à fournir des gestionnaires d'IaaS permettant de déployer des infrastructures privées de Cloud Computing (Clouds privés) ont eux aussi connu un succès important. Leur étoffement en ce qui concerne leurs fonctions et leur rapide adoption permettent d'envisager un futur où les centres de données seront semblables à des infrastructures IaaS privées [24], et où chaque aspect d'un centre de données sera géré par le gestionnaire *IaaS*. Les systèmes *IaaS* étant appelés à devenir de plus en plus complexes, des auteurs ont proposé l'idée d'avoir une architecture de référence [25] pour les systèmes *IaaS* où les fonctions devant être assurées par un gestionnaire d'infrastructure seraient clairement identifiées, et associées chacune à un composant dédié comme présenté dans la figure 2.1.

Les auteurs de [25] séparent les services composant un *Cloud-OS* (gestionnaire *IaaS*)

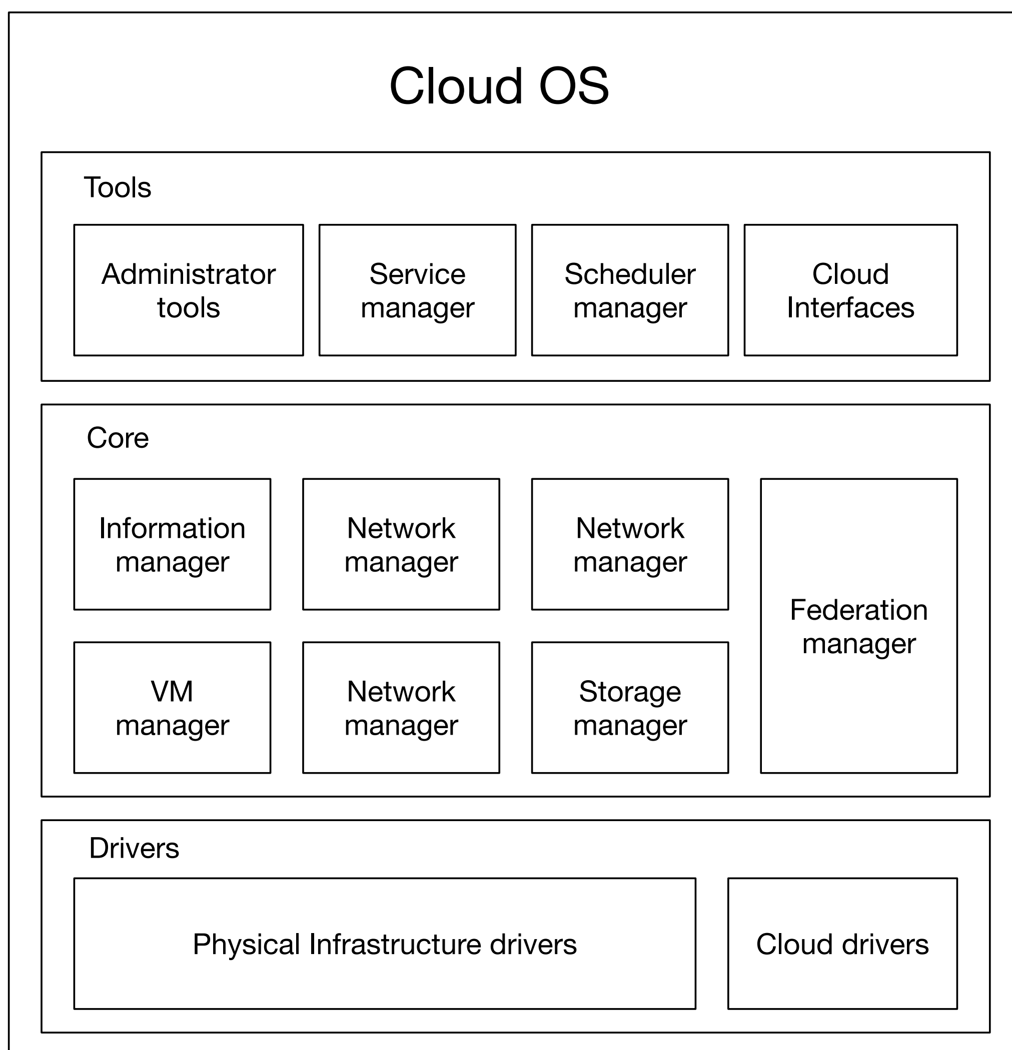

FIGURE 2.1 : Architecture type d’un système d’exploitation pour infrastructure de Cloud Computing.

en trois niveaux de composants construits au-dessus des autres :

- **Drivers** : regroupe tous les composants qui permettent au gestionnaire *IaaS* de s’interfacer avec le matériel des serveurs faisant partie de l’infrastructure. Ce niveau permet entre autres d’abstraire les spécificités matérielles (hyperviseur, stockage, réseau, . . .) permettant ainsi la mise en place d’infrastructures utilisant du matériel hétérogène. Dans la catégorie *Drivers*, on trouve aussi les mécanismes qui permettent à une infrastructure *IaaS* de s’interfacer avec des infrastructures distantes (fédération d’infrastructure).

- **Core** : contient les composants responsables de l’organisation de la production des différentes ressources informatiques (machines virtuelles, stockage, réseau, . . .) fournies par un gestionnaire *IaaS*, mais aussi tous les aspects liés à la gestion des

utilisateurs (authentification, quotas de ressources, ...) et à la facturation. Ce niveau est généralement masqué à l'utilisateur.

- **Tools** : contient des composants qui servent d'interfaces entre les utilisateurs finaux de l'infrastructure *IaaS* et les composants situés dans le niveau **core** : les requêtes des utilisateurs sont traduites en actions qui seront exécutées par les composants **core**.

Les auteurs de cette architecture de référence soulignent que l'adoption des infrastructures de Cloud Computing n'est pas optimale, et proposent deux pistes pour catalyser l'adoption de ces infrastructures :

1. Dans un premier temps, le manque d'interopérabilité entre les infrastructures de Cloud Computing est un problème qui pourrait se corriger au moyen d'une meilleure standardisation des infrastructures *IaaS* avec l'exposition de leurs fonctionnalités au travers d'interfaces logicielles communes (actuellement l'API d'Amazon est *de facto* un standard).

2. Le second axe d'amélioration consiste à faciliter la fédération d'infrastructure *IaaS* en développant notamment des couches d'adaptation logicielle (*Adapter*) qui permettraient de rendre interopérable des infrastructures appartenant à des fournisseurs distincts ou encore de reposer sur des gestionnaires *IaaS* différents.

2.3.3 Revue des principaux systèmes IaaS

OpenStack

En 2010 la Nasa et Rackspace scellent un partenariat afin de fonder le projet OpenStack, qui vise à fournir un gestionnaire *IaaS* sous une licence de logiciel libre. L'objectif est de fournir une collection de services qui peuvent être utilisés indépendamment, mais qui ensemble permettent la mise en place d'infrastructures de Cloud Computing. La première version d'OpenStack contenait deux services : le service *Nova* en charge de la gestion des machines virtuelles, dont le code venait de la plate-forme **Nebula** opérée par la Nasa, et le service *Swift* dont le code était issu du projet *Cloud File Platform* utilisé par Rackspace.

OpenStack est une collection de services, où chaque service s'occupe d'un aspect précis d'une infrastructure *IaaS*. Ces services sont relativement faiblement couplés entre eux, permettant des déploiements modulaires où seuls les services basiques sont indispensables (*Nova*, *Keystone*, *Glance*, *Cinder*). La figure 2.2 illustre le rôle des six principaux services les plus fréquemment déployés au sein d'une infrastructure gérée par OpenStack :

- **Nova** : ce service s'occupe de la gestion des machines virtuelles et de leur cycle de vie. Il sert aussi de liant avec les autres services pour ajouter, aux machines virtuelles qu'il gère, de la connectivité réseau et le support des images.

- **Glance** : ce service s'occupe de la gestion des images des machines virtuelles.

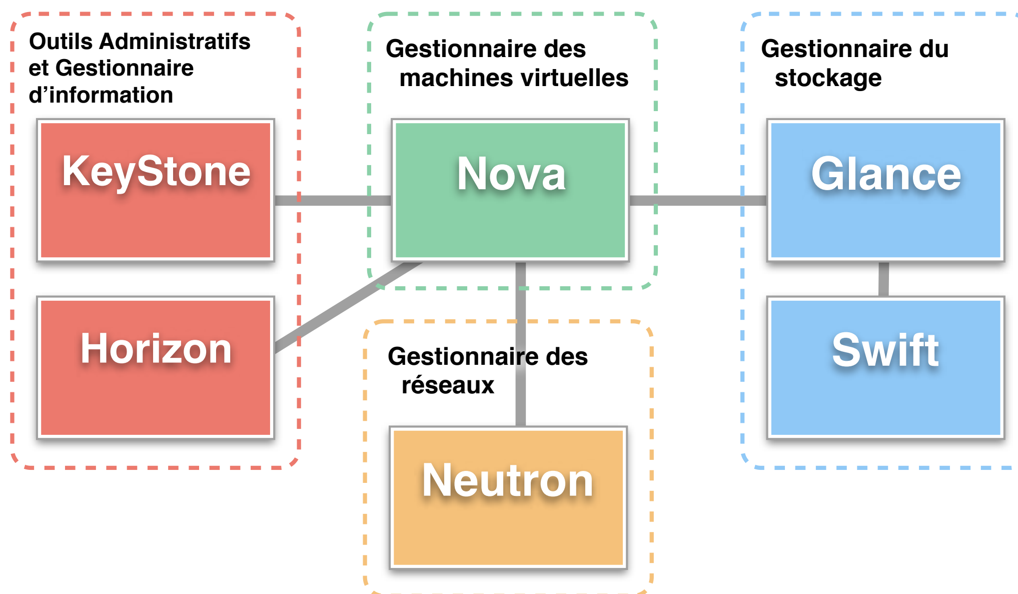

FIGURE 2.2 : Architecture du système OpenStack.

- **Cinder** : ce service s'occupe de la gestion du stockage en mode bloc (*Block Device Storage*). Cette technique permet à une machine virtuelle d'avoir accès à un disque dur virtuel distant, afin d'y stocker des données persistantes.

- **Neutron** : ce service est chargé d'assurer la connectivité entre machines virtuelles à travers la création de réseaux virtuels. Ce service utilise les techniques des réseaux définis de manière logicielle (*Software Defined Network* [26] - SDN). Il est possible de se passer de *Neutron* en utilisant le sous-service *Nova-network*, cependant ce dernier tombe en désuétude.

- **Keystone** : ce service est responsable de la sécurité et de l'authentification au sein d'un déploiement OpenStack.

- **Horizon** : ce service offre une interface graphique aux utilisateurs, qui leur permet de manipuler les ressources que ceux-ci hébergent sur une infrastructure OpenS-tack. Il est possible de se passer d'*Horizon* en utilisant l'API Rest d'openStack, ou au moyen d'une interface en ligne de commande.

Par sa volonté de fournir un gestionnaire *IaaS* qui soit modulaire et modifiable, et grâce à sa licence logicielle libre (Apache 2.0), le projet OpenStack a connu un succès rapide, comptant en janvier 2016 prés de 1800 contributeurs (tous projets confondus) [1], et est utilisé par une variété d'acteurs privés tels que Rackspace, Ebay, Yahoo, HP, Orange, ainsi que des institutions publiques ou académiques telles que le MIT, le Argonne National Laboratory, le CERN et l'Université de Strasbourg [27].

[1] http://stackalytics.com

Globus/Nimbus : de la grille de calcul vers les infrastructures Cloud

En 1995, l'université de Chicago et le laboratoire d'Argonne fondent le projet Globus [28], qui vise à faciliter le développement de services consommant des ressources informatiques (calcul, réseau, stockage, ...) produites sur les grilles de calculs. Les membres du projet Globus ont alors analysé que la plupart des services qui exploitaient des grilles de calcul possédaient des pans entiers de code très similaires, car effectuant des actions très semblables (découverte de ressources, migration de données, gestion des pannes, ...), très souvent *en réinventant la roue*. L'équipe du projet Globus a ainsi proposé une boîte à outils (*Globus Toolkit*) qui contient une implémentation pour les services qui sont souvent réécrits, permettant de gagner du temps en se focalisant sur l'écriture de code à haute valeur ajoutée, et de tirer parti des efforts de standardisation du développement de services permis par le *Globus toolkit*.

L'impact du projet Globus sur la communauté scientifique fut important : le *Globus Toolkit* fut utilisé [29] pour mettre en place les grilles de calculs nécessaires afin de fournir de grandes quantités de calculs à des projets très demandeurs en ressources de calcul comme ceux du CERN responsable de l'analyse de données collectées à la suite d'expériences menées au LHC ou par des organisations faisant des simulations scientifiques. Le *Globus Toolkit* fut aussi utilisé par des acteurs privés tels que HP, IBM, Oracle, construire les infrastructures nécessaires à l'hébergement de leurs services ou de ceux de leurs clients.

En 2005 des scientifiques du laboratoire d'Argonne et de l'Université de Chicago lancent l'initiative Nimbus qui vise à développer un gestionnaire d'infrastructure qui confine les ressources fournies aux utilisateurs dans des environnements isolés fonctionnant sur des grilles de calcul. Les utilisateurs d'une infrastructure utilisant Nimbus accèdent alors à des ressources sous la forme de machine virtuelle, et ont la possibilité de créer et de personnaliser leurs propres environnements en spécifiant les ressources matérielles nécessaires ainsi que les dépendances logicielles à installer. Un des principaux intérêts du projet Nimbus est de fournir des images d'environnements *de références*, personnalisables par les utilisateurs, ainsi qu'une série d'outils permettant de régler leurs propres environnements. Enfin, le projet Nimbus s'inscrit dans la continuité du projet Globus en s'appuyant sur certains composants de ce dernier.

Eucalyptus

En 2009 des chercheurs de l'Université de Californie publient "The Eucalyptus Open-source Cloud-computing System" [30] qui s'intéresse aux problèmes inhérents au modèle des grilles de calcul : les auteurs analysent que pour les très gros besoins en ressources de calcul, il est nécessaire d'agréger les ressources de plusieurs fournisseurs, ce qui se traduit bien souvent par une hétérogénéité matérielle. Cette hétérogénéité doit être prise en compte par les concepteurs de service, ce qui nécessite des compétences techniques avancées.

Les auteurs proposent alors de mettre en place un accès uniforme à cette collection de ressources, en les fournissant aux utilisateurs sous la forme de machines virtuelles. L'infrastructure sur laquelle sont déployées ces machines virtuelles est gérée par le sys-

tème Eucalyptus qui est défini par ses auteurs comme un cadriciel (framework) logiciel permettant de transformer des ressources de calcul et de stockage d'une organisation en ressources de Cloud Computing.

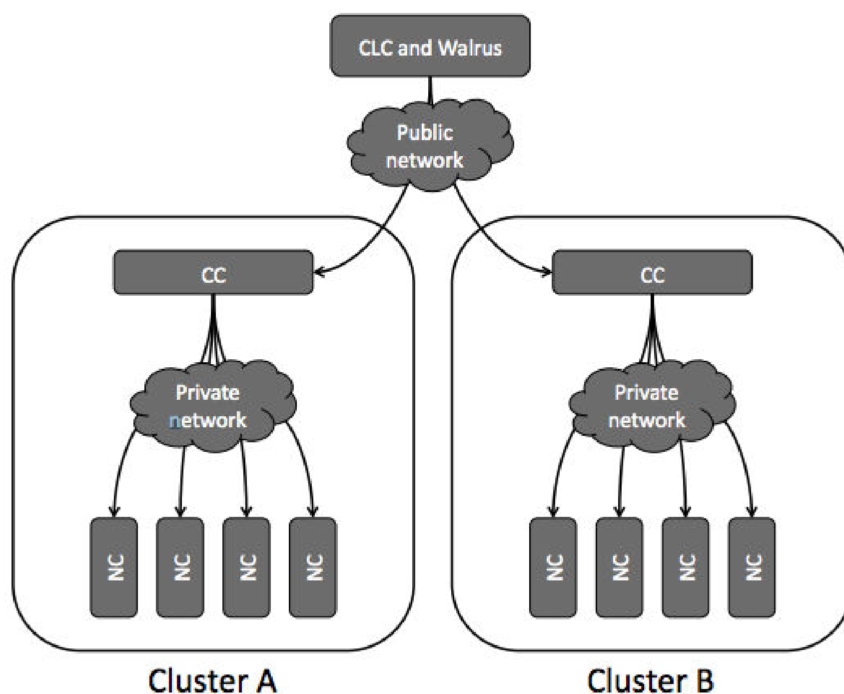

FIGURE 2.3 : Architecture du gestionnaire Eucalyptus (extrait de [30]).

Le système Eucalyptus est un gestionnaire *IaaS* conçu autour d'une architecture modulaire, où chaque composant est responsable d'une fonction précise de l'infrastructure *IaaS*, tout en restant indépendant des autres composants. Chacun des composants d'Eucalyptus est rendu accessible aux autres composants sous la forme d'un service web. Une infrastructure *IaaS*, utilisant Eucalyptus, est organisée de façon hiérarchique : une infrastructure de grande taille sera divisée en plusieurs groupes de nœuds (clusters) de tailles inférieures et où chaque serveur de l'infrastructure (c.-à-d. nœud) est spécialisé comme indiqué par la figure 2.3 :

- **Cloud Controller** : sert de point d'entrée pour communiquer avec l'infrastructure *IaaS*. Il expose une API web qui peut être utilisée pour manipuler l'infrastructure sous-jacente.
- **Cluster Controller** : est déployé dans chacun des clusters de l'infrastructure. Ce type de nœud surveille l'exécution des machines virtuelles au sein du cluster dont il est responsable. Il reçoit des ordres de création de machines virtuelles du *Cloud*

Controller et s'occupe de l'ordonnancement en choisissant le *Node Controller* qui les hébergera.

- **Node Controller** : est responsable du contrôle et de la surveillance des machines virtuelles qu'il héberge. Il reçoit des ordres de création de machines virtuelles du *Cluster Controller*.

- **Walrus (Storage Controller)** : est en charge de la fourniture d'un service de stockage utilisant les mêmes interfaces logicielles que le service *Amazon S3*. Walrus est principalement utilisé pour le stockage des images des machines virtuelles ainsi que les métadonnées des utilisateurs.

Concernant les communications entre machines virtuelles, Eucalyptus propose de connecter ces dernières à travers un système de réseau virtuel, défini de manière logicielle en se basant sur les techniques de réseau d'overlay (Overlay Network). Cela permet à des machines virtuelles qui n'appartiennent pas à la même organisation d'être isolées. Cette technique de virtualisation du réseau entrainant un léger surcoût (un saut réseau supplémentaire via l'ordinateur hébergeant la machine virtuelle.), le choix est donc laissé à l'utilisateur entre privilégier l'isolation de ses ressources de calculs ou de reposer sur un réseau plus conventionnel.

Enfin, Eucalyptus est programmable via une interface logicielle (API) qui permet à ses utilisateurs de manipuler une infrastructure déployée avec Eucalyptus. Celle-ci est calquée sur les interfaces proposées par les services phares d'Amazon EC2 et S3. Cela permet au projet Eucalyptus d'avoir l'ambition d'être utilisé sans modification par des outils reposants sur les deux services évoqués précédemment.

OpenNebula

Parallèlement à l'apparition du projet Eucalyptus, le projet OpenNebula émerge en 2009 et est le fruit de la collaboration de chercheurs issus de l'université de Chicago, de l'université de Madrid ainsi que du laboratoire d'Argonne proposent le projet OpenNebula [31]. Ce projet vise à fournir un gestionnaire d'IaaS misant sur l'ouverture à la fédération avec des infrastructures *IaaS* tierces.

Faisant le constat qu'en 2009 l'écosystème des infrastructures de Cloud Computing se partage entre les *Clouds publics* et les *Clouds privés* (et *Clouds hybrides*), et que ces derniers occupent une place de plus en plus importante, l'équipe du projet OpenNebula estime qu'il faut rendre possible et même faciliter la mise en place d'infrastructures *IaaS* qui soient exploitées par plusieurs fournisseurs. Dans cette optique, le projet OpenNebula propose un gestionnaire *IaaS* qui se place à un niveau d'abstraction plus élevé que les projets d'alors tels Nimbus et Eucalyptus, afin de faciliter la fédération d'infrastructures appartenant à plusieurs fournisseurs.

L'absence de standards ouverts pose le problème de la fédération d'infrastructures qui ne possèdent pas les mêmes APIs, voire qui ne partagent pas toutes les mêmes fonctionnalités. OpenNebula a été conçu pour être nativement ouvert à l'utilisation de ressources issues de fournisseurs externes, grâce à un système de pilotes logiciels externes (*Cloud*

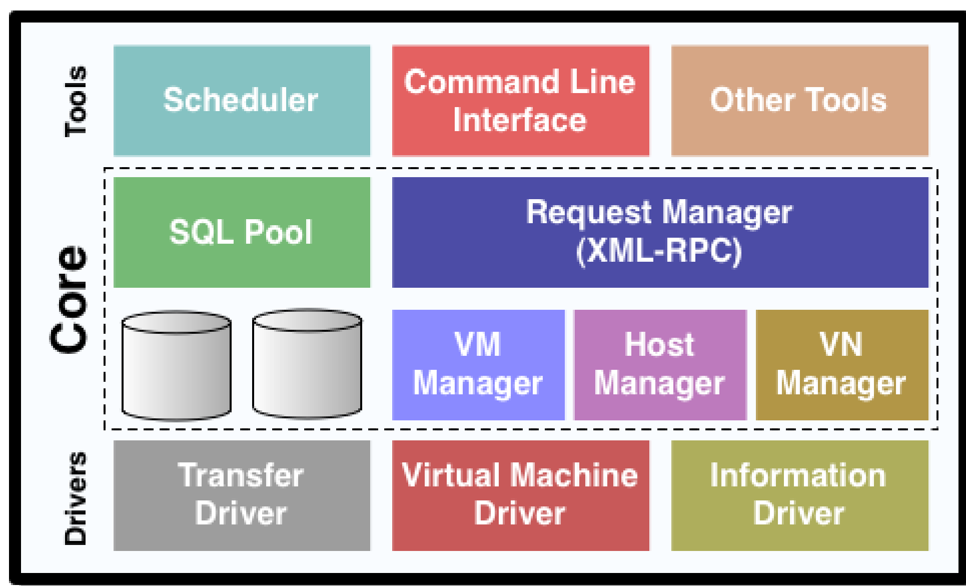

FIGURE 2.4 : Architecture du projet OpenNebula (extrait de [32]).

Drivers). Cette approche permet à OpenNebula de pouvoir piloter les ressources de calculs, qu'elles soient hébergées localement ou dans des infrastructures distantes, du moment que ces dernières sont rendues disponibles via un *Cloud Driver*. Initialement, les infrastructures externes supportées étaient Amazon EC2, Eucalyptus et ElasticHosts.

La figure 2.4 résume les différents services composant le gestionnaire OpenNebula : l'équipe en charge du développement de ce dernier impliquant les mêmes personnes que celles qui ont proposé l'architecture de référence présentée dans la section 2.3.2, les deux architectures sont donc très proches.

2.4 Résumé

Dans ce chapitre nous avons abordé la question de la définition du modèle de Cloud Computing, qui vise à fournir des ressources informatiques (calcul, réseau, stockage, . . .) à la manière utilitaire (*Pay as you go*) tout en reprenant les techniques des grilles de calcul. Comme il n'existe pas de définition formelle de ce modèle, il existe une certaine confusion quand il s'agit de qualifier ce qui relève du Cloud Computing et ce qui relève des grilles de calcul. Des auteurs [18] ont compilé et analysé les principales définitions que l'on peut trouver dans la littérature du Cloud Computing : ils ont ainsi proposé une définition synthétique suffisamment générale pour satisfaire toutes ses déclinaisons applicatives.

Malgré cette définition, le fait que le Cloud Computing possède un vaste spectre d'application pose le problème de pouvoir classifier les services assurés par les fournisseurs. La classification SPI définit trois catégories hiérarchisées de prestations :

- Le niveau bas *IaaS* regroupe les services qui délivrent de la puissance de calcul brute aux utilisateurs en leur fournissant des machines virtuelles de manière à abs-

traire le matériel responsable de l'hébergement. Ce niveau s'adresse à un public technique nécessitant d'accéder à de la puissance de calcul fournie sous la forme de machines virtuelles.

- Le niveau intermédiaire *PaaS* est construit au-dessus du niveau *IaaS* : le fournisseur propose à ses utilisateurs des environnements logiciels déjà configurés pour que ces derniers puissent y développer leurs services. Ce niveau s'adresse aux concepteurs de services informatiques ne voulant pas gérer les aspects matériels de l'hébergement de services.

- Le niveau haut *SaaS* est construit au-dessus des deux précédents niveaux et regroupe les services logiciels rendus disponibles aux utilisateurs finaux au moyen d'un client léger (généralement un navigateur web). Ce niveau s'adresse au grand public, qui se limite à la consommation de services sans voir la complexité sous-jacente.

Enfin, nous avons vu plus en détail la couche *IaaS* : celle-ci a connu un succès croissant tout au long du début du 21e siècle, en partie grâce à la démocratisation des techniques de virtualisation. Au cours de cette période, de nombreux gestionnaires pour infrastructures *IaaS* ont fait leur apparition, la plupart partageant de nombreux points architecturaux. Ces similarités ont rendu pertinente la définition d'une architecture de référence qui définirait les briques logicielles composant un gestionnaire *IaaS*, ce qui a abouti à la proposition d'une architecture pour *Cloud-OS* par l'équipe en charge du développement du projet *OpenNebula*. Parmi les gestionnaires *IaaS*, le gestionnaire *OpenStack* a connu plus de succès que les autres : celui-ci possède une large communauté d'utilisateurs et de développeurs qui contribuent activement à son amélioration.

3

Paysage des infrastructures IaaS et Edge Computing

Afin de satisfaire les besoins toujours plus importants en ressources de calcul, les fournisseurs de Cloud Computing construisent des centres de données toujours plus gros afin d'y héberger leurs infrastructures matérielles tout en réalisant des économies d'échelle. Les acteurs importants qui dominent actuellement le marché du Cloud Computing (Amazon, Google, OVH, Rackspace, ...) sont ainsi capables de financer la construction de mégas centres de données concentrant un grand nombre de serveurs (de l'ordre la centaine de milliers de serveurs d'après [33]), aboutissant à ce qu'une part non négligeable des services Internet soient hébergés sur ces infrastructures. Ce phénomène de concentration de la production des ressources de Cloud Computing dans quelques mégas centres de données n'est pas sans poser des problèmes. Ce chapitre s'intéresse dans un premier temps à ce modèle de mégas centres de données ainsi qu'aux conséquences de sa mise en place. Dans un second temps nous présentons la principale alternative à ce modèle qui consiste à déconcentrer l'hébergement des infrastructures de Cloud Computing en les distribuant, soit sur quelques centres de données éparpillés géographiquement (modèle de fédération), soit plus massivement, en déployant l'infrastructure près de l'utilisateur final, comme le propose le modèle d'*Edge/Fog Computing* [34]. Enfin nous faisons la comparaison entre ces différents modèles afin de comprendre les avantages et les inconvénients qu'ils peuvent avoir les uns par rapport aux autres.

3.1 Splendeur et décadence des mégas centres de données

Pour héberger les services de leurs clients, les fournisseurs d'infrastructures de Cloud Computing construisent des infrastructures comprenant des milliers de ordinateurs (appelés serveurs). Pour fonctionner correctement, un serveur a besoin d'être approvisionné

en électricité, et sa consommation électrique dépend de paramètres tels que la présence de composants énergivores (nombre de processeurs, nombre de cœurs par processeurs, ...) ou de son degré de sollicitation. Au cours de son fonctionnement, un serveur dégage de la chaleur qui doit être évacuée afin de limiter l'usure des composants. En effet, certains composants d'un serveur sont susceptibles de subir de manière fréquente des pannes (c'est le cas des disques durs [35]), et le remplacement d'un composant nécessite souvent une intervention humaine.

Pour réaliser des économies d'échelle, les serveurs sont regroupés dans des centres de données. Ce sont des complexes étudiés spécialement pour l'hébergement de milliers de serveurs bénéficiant d'installations électriques et de refroidissements dimensionnées pour répondre à leurs besoins. Chaque centre de données possède une équipe technique en charge d'exécuter les opérations sur les serveurs telles que la maintenance curative (serveur présentant des anomalies matérielles ou logicielles) ou maintenance préventive (changer des composants proches de la fin de vie avant qu'ils ne tombent en panne).

Concentrer la production des ressources de calcul au sein de complexes spécialisés présente ainsi l'avantage qu'il est possible de réaliser d'importantes économies d'échelle grâce à la rationalisation des coûts (en matériels et en personnel). Plus il a été possible de réaliser des économies d'échelle, plus ce phénomène de concentration a été accentué avec des centres toujours plus grands, pour aboutir aux mégas centres de données déployés aujourd'hui.

Ces mégas centres de données subissent des contraintes liées à leurs tailles, comme les énormes besoins énergétiques engendrés par la consommation électrique des matériels utilisés (serveurs, routeurs) et par les besoins en refroidissement. La facture électrique devenant un poste de dépense non négligeable, les fournisseurs de Cloud Computing ont opportunément placé leurs centres de données dans des zones géographiques où l'énergie électrique était abordable, voire dans des zones bénéficiant de capacités de refroidissement naturelles. Facebook a par exemple déployé un centre de données proche du Cercle polaire Arctique. La ville de Quincy (USA) est ainsi connue pour héberger plusieurs gros centres de données appartenant à de gros acteurs des infrastructures de Cloud Computing (c.f. figure 3.1), profitant de l'énergie bon marché issue des infrastructures hydroélectriques situées dans les environs.

La concentration géographique des centres de production de ressources de calcul pose le problème de la tolérance aux pannes en cas de désastre. Ainsi, quand une catastrophe naturelle ou une action malveillante survient au sein d'un territoire hébergeant des mégas centres de données, l'accès aux ressources informatiques qui y sont hébergées ne peut être garanti. Ce fut le cas le 29 juin 2012 où de violents orages ont éclaté en Virginie du Nord. Ces orages ont provoqué une coupure du système responsable de l'approvisionnement électrique d'un des centres de données d'Amazon situés dans cette zone [36], rendant inaccessibles les ressources qui y étaient hébergées pendant quelques heures. L'infrastructure d'Amazon étant composée de plusieurs centres de données disséminées sur plusieurs sites géographiques, cette panne n'a pas affecté le fonctionnement des autres centres de données. Cependant, l'impact a été important pour les utilisateurs de la plateforme Amazon EC2 qui avaient déployé leurs services sur ce centre de données. Ce fut le

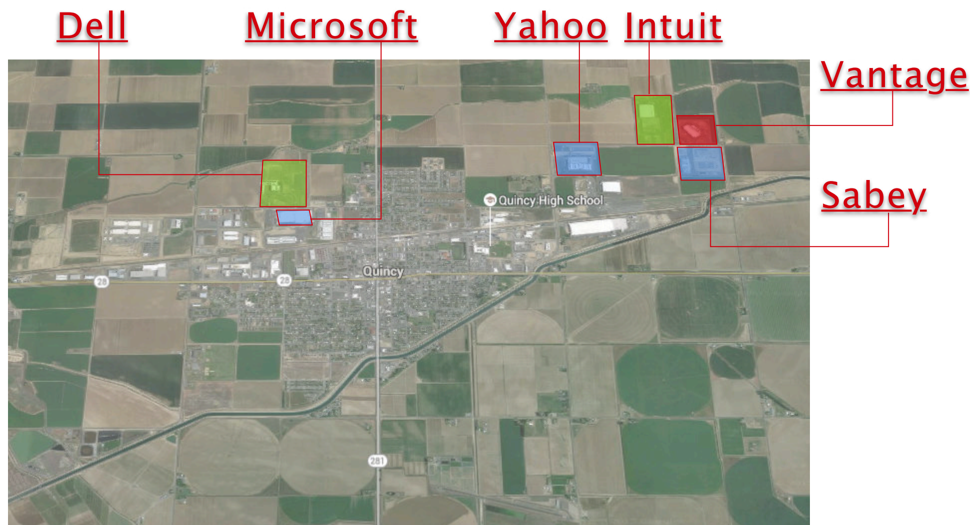

FIGURE 3.1 : Centres de données localisés à Quincy (Washington).

cas notamment pour Netflix, Instagram et Pinterest [1].

Enfin, la concentration des ressources de calcul dans quelques mégas centres de données pose aussi le problème de l'impact sur l'efficacité en termes de trafic réseau. La distance géographique pouvant séparer des utilisateurs consommant un service et l'endroit où celui-ci est hébergé est un paramètre important. La latence réseau étant proportionnelle à la distance géographique parcourue par les données, les utilisateurs situés loin des centres de données peuvent subir de fortes latences réseau. Ainsi dans des applications telles que le *Cloud Gaming* ou la vidéo à la demande, la latence doit être la plus faible possible sous peine de rendre le service inutilisable. Une étude portant sur le Cloud Gaming [37] considère qu'une latence de 80ms est considérée comme la limite pour avoir une expérience de jeu acceptable pour l'utilisateur, or pour un échantillon géographiquement représentatif de 2500 adresses IP américaines, près de 70% avaient une latence supérieure à 80ms avec la plate-forme Amazon EC2. Ces 70% d'utilisateurs montrent que la couverture des utilisateurs avec une bonne latence est trop faible, ce qui est dû principalement au fait qu' Amazon EC2 s'appuie sur de gros centres de données répartis dans un nombre restreint de lieux. Les auteurs de l'étude proposent deux pistes pour améliorer la couverture, dont la première est de construire des infrastructures possédant davantage de centres de données, les centres de données supplémentaires seraient placés dans des zones mal couvertes par les centres existants tout en étant capables de fonctionner avec ces derniers grâce à un réseau rapide. Cette première piste s'apparente au modèle de fédération de centres de données. La deuxième piste qui est proposée est de s'appuyer plus fortement sur le réseau Internet en utilisant les infrastructures CDN (Content Delivery Network) existantes pour y héberger des ressources de calcul. Les infrastructures CDN étant en matière de réseau au plus proches des utilisateurs finaux, cela permettrait de réduire la latence existant entre les utilisateurs et les infrastructures de Cloud Computing. Cette piste s'inscrit dans le mou-

[1]http://venturebeat.com/2012/06/29/amazon-outage-netflix-instagram-pinterest/

vement du *Edge/Fog Computing* qui vise à rapprocher physiquement les infrastructures de Cloud Computing qui hébergent les services des utilisateurs qui les consomment [38].

3.2 Fédérations de nuages

Dans cette partie, nous nous intéressons au modèle de fédération qui résout en partie les problèmes liés la concentration des ressources de calcul dans quelques mégas centres de données.

Dans la section 1.3, il été montré que l'informatique utilitaire venait d'une analogie entre ressources électriques et ressources de calcul : des utilisateurs consomment la demande des ressources de calcul produites par des fournisseurs d'infrastructure de Cloud Computing. Ce modèle a inspiré les grilles de calcul et les infrastructures de Cloud Computing : dans le modèle électrique, les centrales électriques sont fédérées au sein d'un réseau électrique et quand la demande devient supérieure à la capacité de production d'une seule centrale électrique, la demande non satisfaite est déléguée aux autres centrales électriques. Les grilles de calcul sont allées au bout de cette analogie : pour offrir le plus de ressources de calcul à leurs utilisateurs, celles-ci ont très tôt été fédérées. En ce qui concerne le Cloud Computing, la fédération d'infrastructures *IaaS* été un sujet de recherche qui a émergé en 2009 avec en Europe les travaux autour du projet *RESERVOIR* [39], tandis qu'en parallèle l'équipe de Nimbus proposait de faire des infrastructures de Sky computing fédérées autour de son gestionnaire *Nimbus* [40].

La fédération d'infrastructures de Cloud Computing, ou fédération de nuages, est une technique permettant à un fournisseur d'étendre son infrastructure en lui ajoutant des ressources issues d'autres fournisseurs, dont les infrastructures ne sont pas surchargées. La connectivité entre les ressources issues de différents fournisseurs est assurée au moyen d'un lien réseau privé ou du réseau Internet. Cette approche est la première étape pour offrir une gestion plus raisonnable des pics de consommation. En effet, sans technique de fédération, les fournisseurs habituellement surdimensionnent leurs infrastructures afin d'éviter de se trouver en défaut de ressources comme lors des pics de consommation [41]. Ce surdimensionnement baisse l'efficacité économique de l'infrastructure, car en dehors des pics de consommation une partie de l'infrastructure est inutilisée et donc coûte de l'argent sans en rapporter. À l'inverse, la fédération permet de garder une infrastructure de taille suffisante pour une consommation normale en dehors des pics de consommation et de déléguer à des fournisseurs externes une partie de la charge induite lors d'un pic de consommation. Enfin, la fédération d'infrastructures permet d'agréger des ressources de plusieurs infrastructures, ce qui permet aux services qui y sont hébergés d'avoir accès à une plus grande quantité et variété de ressources qui deviennent virtuellement infinies dans le cas de fédérations impliquant de gros fournisseurs tels qu'Amazon ou Microsoft. Ce mythe de la ressource infinie est un des piliers du Cloud Computing.

Le paysage du Cloud Computing est assez varié et on peut trouver de nombreuses technologies de gestionnaire *IaaS* ainsi que de nombreux acteurs de tailles diverses. Cette variété explique les nombreuses architectures de fédérations d'infrastructures de Cloud Computing qui ont émergé. Les auteurs de [25] se sont intéressés à la définition des

grandes tendances architecturales pour la fédération d'infrastructures de Cloud Computing, et ont mis en évidence que les fédérations pouvaient être classées selon deux critères :

- **Degré de couplage** entre les infrastructures composant la fédération. Le couplage peut être assimilé au degré de contrôle qu'une infrastructure a sur les autres infrastructures de la fédération. Ainsi, plus le couplage sera fort, plus le contrôle d'une infrastructure sur les autres infrastructures de la fédération pourra être intrusif. À l'inverse, un couplage faible signifie que les infrastructures n'ont qu'un accès limité sur les ressources externes.
- **Nombre d'organisations** impliquées dans la fédération. Une fédération d'infrastructures appartenant à des entreprises différentes impliquera l'établissement de contrats régulant la consommation par un des partenaires des ressources des autres partenaires, ce qui ne sera pas forcément le cas d'une fédération d'infrastructures appartenant à une unique organisation.

À partir de ces critères, quatre modèles de fédération ont été dégagés : *Bursting*, *Broker*, *Aggregated* et *Multitier*. La suite de cette partie traite de ces modes de fédération, s'intéressant à leur description ainsi que leurs caractéristiques mises en évidence par [25].

3.2.1 *Bursting* (Cloud hybride)

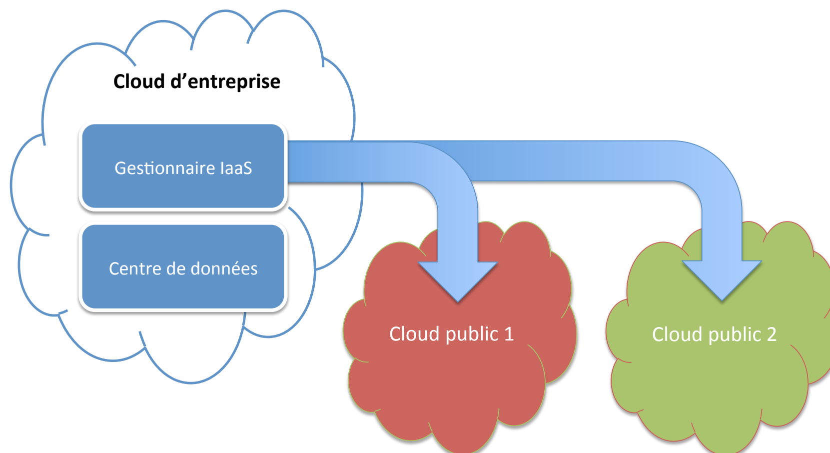

FIGURE 3.2 : Schématisation d'une fédération reposant sur une architecture de type *Bursting* ([25]).

Dans ce modèle de fédération d'infrastructures *IaaS*, un fournisseur combine son infrastructure avec des ressources issues d'autres fournisseurs, afin de proposer à ses utilisateurs de consommer une agrégation de ses ressources avec des ressources issues d'infrastructures externes. Le *Bursting* est généralement utilisé pour faire face à des pics de consommation. Ainsi plutôt que de surdimensionner son infrastructure pour faire face à de rares pics de consommation, le fournisseur va ponctuellement déléguer une partie de la charge de travail excédentaire à des fournisseurs tiers. Comme illustré dans la figure 3.2, le fournisseur initial a cependant une visibilité limitée sur les ressources appartenant aux fournisseurs externes, car il n'appartient pas à la même organisation que les autres fournisseurs et a une visibilité identique à celle des clients de ces derniers. En ce qui concerne l'organisation de la fédération, c'est l'infrastructure du fournisseur initial qui est en charge d'exploiter la fédération, les autres infrastructures n'ayant pas conscience de cette fédération.

3.2.2 *Broker*

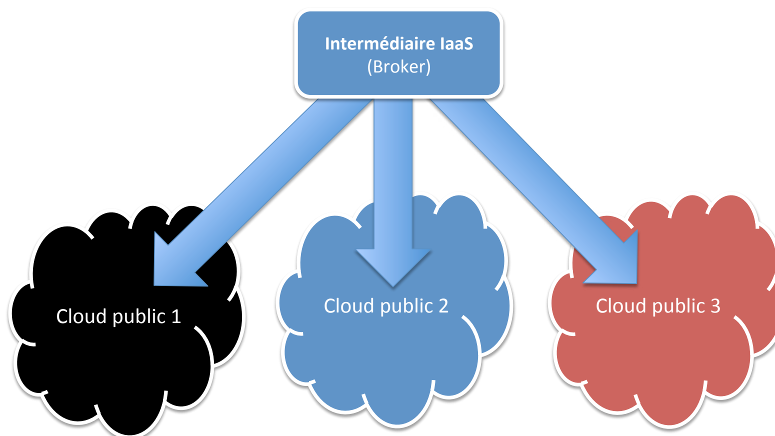

FIGURE 3.3 : Schématisation d'une fédération reposant sur une architecture de type *Broker* ([25]).

Ce modèle de fédération est le plus courant, et comme le montre la figure 3.3, un composant central (*Broker*) sert d'intermédiaire entre les utilisateurs d'une fédération d'infrastructures *IaaS* et les infrastructures qui la composent. Le *Broker* reçoit les requêtes des utilisateurs et s'occupe de les satisfaire en déployant des ressources au sein des infrastructures impliquées dans la fédération. Les Brokers les plus avancés sont capables de

prendre des décisions de placement des ressources qui optimisent les coûts financiers, énergétiques, et certains peuvent même faire varier le degré de distribution des ressources en les répartissant sur plusieurs infrastructures pour des questions de tolérance aux pannes notamment.

Cependant le *Broker* n'a qu'une vision réduite des infrastructures impliquées dans la fédération, ce qui limite les possibilités de la fédération au plus petit commun dénominateur des fonctions offertes par celles-ci. Enfin, il est courant que le *Broker* soit un composant centralisé, ce qui signifie qu'en cas de panne de ce dernier les utilisateurs perdent l'accès à l'infrastructure fédérée. L'infrastructure se met alors à fonctionner de manière dégradée. Une solution consiste alors de distribuer le *Broker* avec des mécanismes de tolérance aux pannes.

3.2.3 Aggregated

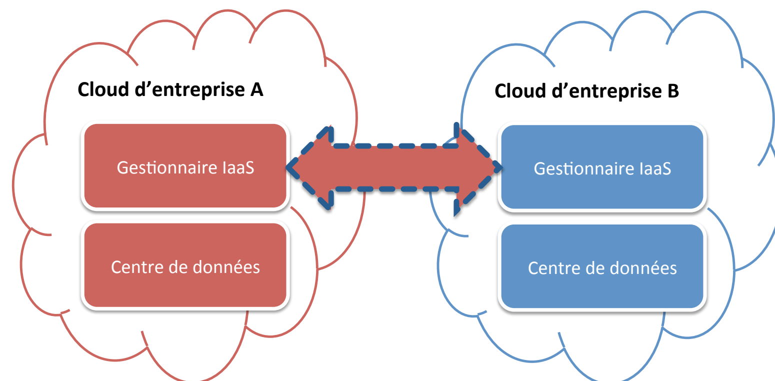

FIGURE 3.4 : Schématisation d'une fédération reposant sur une architecture de type *Aggregated* ([25]).

L'agrégation d'infrastructures de Cloud Computing est un modèle où plusieurs fournisseurs s'accordent pour combiner leurs infrastructures afin d'offrir à leurs utilisateurs une plus grande quantité de ressources de Cloud Computing. Avec ce mode de fédération, on observe un couplage variable entre les infrastructures, selon qu'elles appartiennent ou non à la même organisation :

Dans le cas où les fournisseurs appartiendraient à des organisations différentes, la coopération se fait dans le cadre d'un contrat définissant dans un premier temps les condi-

tions de l'accès que les utilisateurs d'une organisation vont avoir sur les infrastructures des autres organisations (afin de garantir qu'une infrastructure ne vampirisera pas les autres), et dans un deuxième temps le degré d'intrusion d'une organisation sur les infrastructures tierces (comme en autorisant un certain contrôle sur les politiques d'ordonnancement en forçant le placement de machines virtuelles sur le même cluster). Ce cas correspond à un cas de couplage faible entre les infrastructures.

À l'inverse, si l'agrégation se fait entre des infrastructures appartenant à une même organisation, l'intrusion d'une infrastructure sur les autres infrastructures sera plus facilement tolérée, aboutissant à une situation de couplage fort entre les infrastructures.

3.2.4 *Multitier*

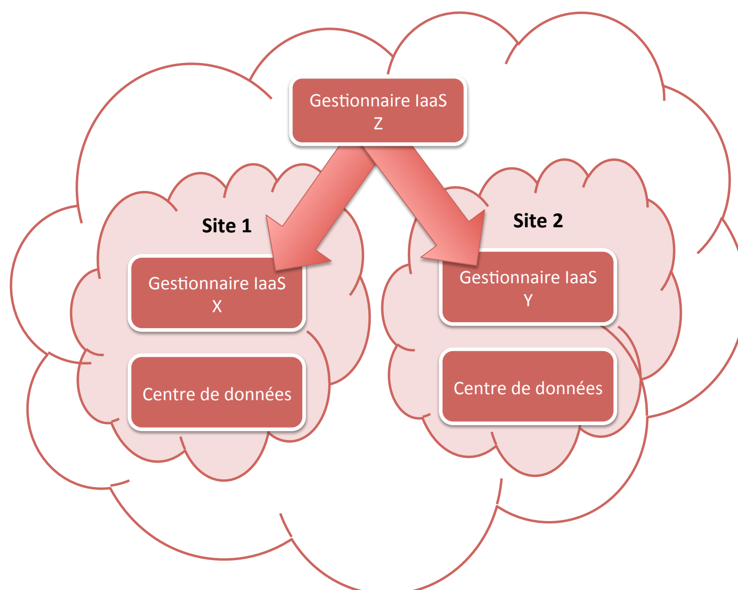

FIGURE 3.5 : Schématisation d'une fédération reposant sur une architecture *Multitier* ([25]).

Ce modèle de fédération s'intéresse à la mise en place d'une infrastructure de Cloud Computing distribuée sur plusieurs sites géographiques (multisite), où chaque site géographique héberge une infrastructure *IaaS* complète. Celles-ci sont regroupées au travers d'une infrastructure hiérarchique où au niveau supérieur se trouve un gestionnaire *IaaS* en charge de piloter les gestionnaires *IaaS* situés sur chacun des sites. Ce gestionnaire *IaaS* supérieur sert d'intermédiaire avec les utilisateurs sur un modèle proche de la fédération à base de *Broker* introduite dans la section 3.2.2.

En général, les infrastructures appartiennent toutes à une même organisation, ce qui autorise un degré d'intrusion maximal du gestionnaire IaaS supérieur dans les infrastruc-

tures sous-jacentes, mais aussi permet l'utilisation de mécanismes *IaaS* avancés.

Seul le gestionnaire situé en haut de la hiérarchie est visible de l'utilisateur, et ce dernier a ainsi l'impression d'utiliser une infrastructure *IaaS* unique (Single System Image) de grande taille, sans nécessairement être conscient du caractère distribué de la fédération.

Un tel modèle est intéressant pour mettre en place des mécanismes avancés de haute disponibilité, d'équilibrage de charge entre les sites et de tolérances aux pannes. De la même manière que pour le modèle de fédération de type *Broker*, si le niveau supérieur subit une panne, les infrastructures sous-jacentes deviennent inaccessibles pour les utilisateurs extérieurs, rendant indisponible l'infrastructure.

3.3 Du modèle d'Edge Computing au Fog Computing

Le grand succès du modèle de Cloud Computing a permis de mettre en évidence les lacunes inhérentes à la concentration de la production de ressources de calcul dans quelques grosses infrastructures de Cloud Computing, telles que le fait que la tolérance aux pannes n'était pas naturelle et qu'elle menait à un latence réseau non optimale. Les fédérations d'infrastructures *IaaS* permettent de résoudre le problème de la tolérance aux pannes en mettant en place des mécanismes de haute disponibilité, cependant elles ne répondent que partiellement au problème de la latence induite par la distance qui peut exister entre l'endroit où un service est hébergé et celui où il est consommé.

Le récent succès de l'informatique mobile, symbolisé par l'invasion des *appareils mobiles* (*Smartphones*) dans le quotidien, a ouvert la voie à un nouveau spectre de problèmes. En effet, ces appareils ayant des ressources informatiques (calcul, stockage) modestes, celles-ci limitent les capacités de traitement des appareils, les rendant incompatibles avec l'exécution de tâches complexes qui apparaissent comme énergivores. Les concepteurs d'applications mobiles se sont donc rapidement appuyés sur les plates-formes de Cloud Computing pour étendre les capacités limitées des *Smartphones*, en faisant profiter à leurs possesseurs des ressources virtuellement illimitées offertes par le Cloud Computing.

Une des particularités de l'informatique mobile est que les utilisateurs d'applications mobiles peuvent être très nombreux et géographiquement éparpillés, ils ne peuvent donc tous être proches des centres de données qui sont à l'inverse souvent concentrés en quelques endroits. Ce fait peut induire une importante latence réseau entre les utilisateurs mobiles et les centres de données, impactant les échanges entre les applications mobiles et les infrastructures où est déportée une partie de leurs calculs et stockages, baissant la réactivité de ces applications.

Des initiatives se sont développées pour amener les ressources de calcul au plus près des utilisateurs d'applications mobiles en modifiant significativement la façon dont les infrastructures de calcul sont déployées. Ainsi, Cisco propose le paradigme de *Fog Computing* [42] qui étend celui de Cloud Computing en combinant des technologies qui sont arrivées à maturité [43]. L'idée principale du *Fog Computing* est de rapprocher les ressources de calcul des utilisateurs finaux, reprenant l'idée du modèle d'*Edge Computing* [44] en plaçant les ressources de Cloud Computing au niveau des extrémités du réseau :

> *We argue that a new platform is needed to meet these requirements ; a plat-*

> *form we call Fog Computing, or briefly, Fog, simply because the fog is a cloud close to the ground. – Précisions sur l'origine du nom Fog Computing [42]*

Les infrastructures de *Fog Computing* ont été proposées dans l'objectif de permettre la construction de plate-formes capables de supporter les besoins liés au grand succès des applications mobiles pour appareils mobiles ainsi que l'émergence annoncée de l'Internet des objets (*Internet of Things - IoT*), où jusqu'à 50 milliards d'appareils connectés à Internet deviendraient demandeurs en ressources informatiques [43]. Les consommateurs de ressources de calculs étant amenés à être de plus en plus nombreux et fortement éparpillés, le *Fog Computing* propose de migrer les infrastructures de calcul à l'extrémité du réseau, afin de pouvoir permettant de mettre en place des traitements locaux limitant la congestion réseau. Cette approche n'est pas nouvelle et rappelle fortement la technique des réseaux de distribution de contenus (*Content Delivery Network* - CDN) [45], à ceci prés que l'idée est ici de combiner le modèle de Cloud Computing avec les avancées récentes en termes de "logiciélisation" des fonctions réseau (*Network Function Virtualisation* - NFV) [46] afin de proposer à tous ces nouveaux appareils des ressources de calcul de manière homogène. Pour pouvoir fonctionner au mieux, les plate-formes de *Fog Computing* sont amenées à mettre l'accent sur le support de propriétés de localité réseau, en étant géographiquement distribuées, en supportant la mobilité des utilisateurs (Churn) et en s'ouvrant à la fédération entre des plate-formes de Cloud Computing situées sur des réseaux différents.

Enfin [47] effectue une revue des infrastructures de *Fog Computing* et distingue deux concepts orthogonaux qui en les superposant, recouvrent les principes du *Fog computing* :

- **Mobile Cloud Computing** : regroupe les cas où les applications mobiles déchargent une partie de leurs calculs et de leurs stockages vers des plate-formes de Cloud Computing. Il n'y a pas vraiment de contrainte sur la localisation de ces plate-formes : elles peuvent aussi bien être situées dans des mégas centres de données qu'être plus proche de l'utilisateur mobile.

- **Mobile Edge Computing** : regroupe les infrastructures dont une partie est hébergée à l'extrémité des réseaux mobile, c'est-à-dire au plus près des antennes mobiles. Cette partie de l'infrastructure héberge des ressources de calcul, qui sont ainsi au plus proche des utilisateurs mobiles, permettant de bénéficier de faibles latences réseau lors de la communication entre les applications mobiles et les ressources de calculs, ce qui n'est pas le cas avec les infrastructures de Cloud Computing classiques.

Une grande partie des propositions qui concernent le *Fog Computing* s'attaque au problème de la distance entre les services hébergés dans les infrastructures de *Cloud Computing* classiques et les applications mobiles, en ajoutant un niveau intermédiaire qui servirait de "proxy" entre les deux. Les *Cloudlets* [34] sont la principale implémentation complète du modèle de *Fog Computing*. Ceux-ci consistent à déployer des serveurs (ou des groupes de serveurs) de manière proche des utilisateurs pour d'absorber une partie de la charge de travail créée par les applications mobiles afin d'améliorer la réactivité, de

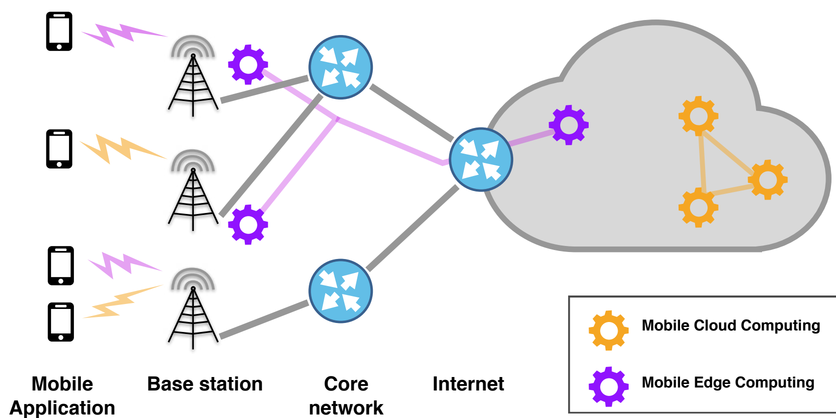

FIGURE 3.6 : Illustration du modèle de Edge Computing.

soulager l'infrastructure réseau ou de calcul, ou de proposer des services tirant avantage de la localisation des utilisateurs [48]. Bien qu'il existe plusieurs tentatives d'implémentations du modèle *Fog Computing* [49, 50], il n'existe pas de technologie de référence, chacune des implémentations s'attaquant à un problème donné, tel que le stockage d'information [51] ou le traitement au plus proche de l'utilisateur [50, 49]. Enfin, à cause de la difficulté d'écrire un système qui permette à un ensemble de ressources de fonctionner exclusivement au niveau de l'extrémité réseau sans partie centralisée [43], les propositions et implémentations actuelles s'appuient toujours en partie sur le modèle actuel de *Cloud Computing*.

3.4 Comparaison des trois modèles de Cloud Computing

Dans cette partie nous comparons les trois modèles du Cloud Computing que nous avons précédemment évoqués. Le premier critère de comparaison qui vient à l'esprit est celui du coût financier. Une étude s'est intéressée aux différents coûts financiers engendrés par la construction et le fonctionnement des centres de données [1], ces coûts serviront de premier critère de comparaison. Nous avons aussi vu que le modèle de fédération était intéressant, car il permettait de s'attaquer aux problèmes de tolérance aux pannes et d'efficacité lié au surdimmensionnement des centres de données pour répondre aux pics d'utilisation, ces deux critères sont donc à prendre en compte pour faire la comparaison entre les modèles de Cloud Computing. Enfin, la gestion de l'éloignement entre les infrastructures et les utilisateurs étant la spécialité des infrastructures d'*Edge/Fog Computing*, la latence pouvant exister entre les utilisateurs de services et le lieu d'hébergement de ces services est un critère à naturellement prendre en compte. Pour mieux comprendre ce qui sera évalué pendant la comparaison, nous donnons une description détaillée de chacun de ces critères :

- **Besoins énergétiques :** un serveur consomme de l'énergie pour permettre à ses composants de fonctionner, ainsi que pour les refroidir. En conséquence, plus un centre de données contiendra un nombre important de serveurs, plus ses besoins

énergétiques seront importants. Enfin, quand les fournisseurs d'électricité domestiques ne sont pas en mesure de fournir suffisamment d'énergie pour couvrir les besoins d'un centre de données, il faut alors faire appel à un fournisseur s'adressant aux professionnels, qui offrira des prestations plus onéreuses. Dans le cas d'un méga centre de données, l'approvisionnement en courant se fait avec du courant "haute tension" qui doit ensuite être transformé en courant "basse tension", ce qui implique des investissements dans du matériel électrique.

- **Les coûts :** les coûts d'une infrastructure de Cloud Computing ont plusieurs sources, qui sont principalement dues à l'achat de serveurs, à l'acquisition de matériel d'alimentation électrique et de refroidissement, l'approvisionnement en électricité [52] ainsi que le coût des liens permettant le raccordement du centre de données avec le réseau Internet [1]. Ces coûts vont avoir un poids plus ou moins important selon le fait que le fournisseur IaaS a construit un centre de données entièrement neuf ou s'il s'est appuyé sur des infrastructures existantes (location ou mutualisation).
- **Facilité d'entretien :** certaines opérations de maintenance dans les centres de données nécessitent des interventions humaines. Le fait que les équipes techniques ne soient pas sur place rallongeant le délai d'intervention, cela a conduit de nombreux fournisseurs d'infrastructures à faire le choix d'avoir une équipe de maintenance toujours présente au sein des infrastructures. Dans le cas d'une infrastructure distribuée, la seule façon de réaliser ces interventions dans un délai court est d'avoir plusieurs équipes de maintenance.
- **Tolérance aux pannes :** à large échelle, les pannes doivent être considérées comme étant la norme. En cas de panne d'une partie d'une infrastructure, le fournisseur d'IaaS doit avoir mis en place des mécanismes qui permettent de résoudre ces situations, sous peine de subir une chute de la qualité de service.
- **Latence avec les utilisateurs :** les utilisateurs accèdent via Internet aux services hébergés dans les infrastructures de Cloud Computing. Plus l'hébergement d'un service est éloigné géographiquement de ses utilisateurs, plus ces derniers feront l'expérience d'une latence importante. Dans ce le cas de certains services (Jeux en ligne), le seuil de latence acceptable peut être bas.

En plus des critères évoqués plus haut, nous ajoutons deux autres critères qui nous semblent importants :

- **Complexité juridique :** quand des centres de données appartenant à plusieurs fournisseurs sont fédérés pour constituer une unique infrastructure de Cloud Computing, un contrat organisant la mutualisation des ressources doit être établi [25]. Celui-ci fixe les conditions d'accès des utilisateurs d'un fournisseur sur les infrastructures des autres fournisseurs. Dans le cas d'une fédération impliquant une fédération de centres de données situées dans plusieurs pays, chaque centre de données est soumis aux lois de son pays hôte, ce qui peut poser problème dans le cas où les lois d'un pays sont contradictoires avec les lois des pays des autres centres de données.

- **Support du calcul haute performance (HPC) :** le HPC est un domaine qui s'intéresse à l'exploitation maximale des ressources de calcul d'ordinateurs mis en réseau. Contrairement aux infrastructures de Cloud Computing qui visent à satisfaire le critère de passage à l'échelle en s'appuyant sur du matériel à prix abordable utilisé en très grande quantité, le HPC cherche à satisfaire le critère de performance en se basant sur du matériel très performant et onéreux.

Le tableau 3.1 reprend les critères précédemment évoqués, et les applique aux trois solutions vues au cours de cette partie. Pour chaque critère, nous avons évalué si le degré de satisfaction que permettait chacune des solutions.

Critère	Mégas centres de données	Fédérations de centres de données	Edge/Fog Computing
Besoin énergétique	(-) Nécessite un très gros approvisionnement électrique concentré sur un unique site.	(-/+) Besoins électriques répartis sur quelques sites (l'approvisionnement par site reste important).	(+) Besoins électriques répartis sur beaucoup de sites (l'approvisionnement par site est faible).
Coût financier	(-) Investissements importants dans les infrastructures électriques et de refroidissement.	(-) Coût foncier, multiplication des dépenses en infrastructures électriques et de refroidissement. (-) Coût de la bande passante réseau si l'infrastructure est internationale (accords de peering).	(+) Besoins en infrastructures électriques et de refroidissement plus faibles, possibilité de mutualiser avec les infrastructures des points de présence déjà déployés.
Entretien	(+) Une équipe unique de maintenance doit être présente 24h/24 dans le centre de données.	(+) Chaque centres de données a sa propre équipe de maintenance 24h/24. (?) En fonction des moyens financiers, les équipes sont présentes 24h/24 ou sont d'astreinte.	(?) Dans le cas d'un opérateur de télécommunications déployant une infrastructure d'Edge Computing dans ses propres points de présence, il est possible de mutualiser avec les équipes en charge de la maintenance des points de présence.
Tolérance aux pannes	(-) Concentrer l'hébergement dans un centre de données unique expose l'infrastructure au phénomène de SPOF (Single Point Of Failure).	(+) Répartir l'infrastructure sur quelques sites permet la mise en place de mécanismes de tolérance aux pannes.	(+) Répartir l'infrastructure sur beaucoup de sites permet la mise en place de mécanismes de tolérance aux pannes.

Latence réseau avec les utilisateurs	(-) Impossibilité pour un centre de données unique d'être proche de tous les utilisateurs quand ceux-ci sont géographiquement répartis.	(+) Une infrastructure géographiquement répartie permet d'améliorer la couverture des utilisateurs en terme de faible latence.	(+) Latence minimale grâce à l'hébergement de serveurs dans les points de présence proches des utilisateurs.
Complexité juridique	(+) Concentrer l'hébergement dans un centre de données unique simplifie l'aspect juridique, car il n'y a qu'un seul fournisseur et une seule législation nationale.	(?) En fonction du nombre d'entités juridiques impliquées dans la fédération, il peut être nécessaire d'établir un contrat stipulant les rôles de chacun. Si la fédération implique des acteurs de différents pays, il y a un risque de conflit législatif.	(+) Les infrastructures de Edge/Fog Computing s'adressent aux opérateurs de télécommunications. Ces derniers organisent déjà des coopérations entre opérateurs de pays différents.
Support du HPC	(+) Optimal pour le calcul haute performance, car toutes les ressources de calcul sont situées au même endroit et bénéficient d'un réseau rapide.	(+) Dépend de la taille des centres de données impliqués dans la fédération.	(-) Les centres de données sont de petites tailles et la latence inter centres de données peut être élevée. Cela semble incompatible avec les utilisations du HPC où les ressources de calculs doivent avoir une faible latence réseau.

TABLE 3.1: Comparaison multi-critère des différents modèles de déploiement du Cloud Computing

3.5 Résumé

Dans ce chapitre, nous avons vu que le succès du Cloud Computing a poussé les fournisseurs à construire des centres de données très grands connus sous le nom de *mégas centres de données*. Pour réaliser des économies d'échelle, ces centres sont toujours plus grands, ce qui pose le problème de leur coût énergétique, de leur tolérance aux pannes, et le fait que cette concentration de grandes quantités de ressources de calcul dans un nombre restreint de lieux géographiques les éloigne inévitablement de la majorité des utilisateurs. Une alternative est alors de décentraliser ces mégas centres de données.

La première technique de décentralisation consiste à faire de la fédération d'infrastructures de Cloud Computing, qui permet à un fournisseur d'agréger son infrastructure de Cloud Computing avec celles d'autres fournisseurs. La fédération a plusieurs avantages, le premier étant que grâce à elle les utilisateurs peuvent accéder à plus de ressources de calcul sans qu'un des fournisseurs n'ait à surdimensionner son infrastructure. Le second avantage des fédérations est qu'elles permettent de répondre au problème de la tolérance aux pannes des infrastructures hébergées dans les mégas centres de données. En cas de panne d'une infrastructure, il est possible de mettre en place des mécanismes pour pouvoir utiliser celles qui sont encore opérationnelles (mécanismes de haute disponibilité).

La deuxième technique de décentralisation des infrastructures de Cloud Computing est le modèle d'*Edge Computing* ou *Fog Computing*. Ce modèle permet d'attaquer pleinement le problème de la forte latence réseau qui peut exister entre les utilisateurs d'un service et le centre de données où il est hébergé, alors que ce problème est très partiellement résolu avec les techniques de fédération. En déplaçant l'hébergement des ressources de calcul à l'extrémité du réseau Internet mobile, il est possible de soulager les appareils mobiles (Smartphones) fortement sollicités par des applications gourmandes en calcul, en leur fournissant des ressources de Cloud Computing accessibles avec une latence faible. Le développement de ce modèle devient de plus en plus intéressant, à mesure que les applications mobiles exécutées sur *Smartphone* envahissent le quotidien et que ces derniers tirent parti des réseaux mobiles rapides.

Infrastructures IaaS massivement distribuées avec OpenStack

Dans ce chapitre, nous nous intéressons aux solutions actuelles permettant de mettre en place des infrastructures OpenStack passant à l'échelle. Dans un premier temps nous évoquons l'architecture d'OpenStack ainsi que ses composants limitant son passage à l'échelle, permettant ensuite d'introduire les moyens de lui assurer un fonctionnement distribué. Nous évoquons en premier lieu la distribution dite "à plat" permettant simplement de répartir la charge soumise à une infrastructure OpenStack sur plusieurs nœuds. Nous poursuivons en évoquant les distributions hiérarchiques que sont les *Cells* et *Cascading OpenStack*, pointant aussi du doigt leurs limitations.

4.1 Passage à l'échelle d'OpenStack

OpenStack est un projet communautaire qui développe un gestionnaire IaaS, sous licence logicielle libre. Ce projet a connu un succès très important. Beaucoup d'infrastructures de Cloud Computing utilisent OpenStack. Son succès se mesure aussi à sa grande communauté de contributeurs (près de 1800 concepteurs logiciels) qui est très active, améliorant continuellement le projet, et dont le rythme de publication au sein de nouvelles versions qui sortent tous les six mois. Architecturalement, le gestionnaire OpenStack est une agrégation de projets qui peuvent être utilisés indépendamment, chacun s'occupant d'une fonction précise au sein d'une infrastructure de Cloud Computing. Cette architecture et la vivacité de la communauté font qu'à chaque nouvelle version, de nouveaux projets sont ajoutés au projet OpenStack, permettant ainsi d'étoffer et de mieux couvrir les besoins technologiques des organisations possédant une infrastructure de Cloud Computing. Naturellement, les éléments d'OpenStack qui correspondent aux modes du moment, ou qui sont les plus populaires ont attiré plus d'attention de la part des contributeurs, tandis que

d'autres composants moins attrayants sont dans un état moins avancé.

Les mécanismes permettant un fonctionnement distribué d'OpenStack semblent appartenir à la catégorie de ceux ayant reçu le moins d'attention. OpenStack utilise une architecture en service, conçue pour permettre un passage à l'échelle horizontale [53]. Danc cette architecture chaque service peut être instancié plusieurs fois sous la forme d'agents, qui ne partagent pas d'états internes entre eux, multiplier le nombre d'agents permet de répartir la charge de travail imposée à une infrastructure OpenStack, et d'ainsi améliorer sa réactivité. De plus, cette architecture organise la collaboration entre les agents sur le partage d'un bus de messagerie ainsi que d'une base de données relationnelle. Ainsi les distributions des agents des services, de la base de données relationnelle ainsi que du bus de messagerie sont nécessaires pour qu'OpenStack fonctionne pleinement de manière distribuée. Il s'agit d'une première approche pour distribuer le fonctionnement d'une infrastructure OpenStack, que l'on pourrait qualifier de distribution à plat.

Le projet OpenStack reconnait que le passage à l'échelle d'une infrastructure est en conséquence limité par le passage à l'échelle de la base de données relationnelle et du bus de messagerie. Or plus une infrastructure est grande, plus il y aura d'agents sollicitant la base de données et le bus de messagerie. Les bases de données relationnelles ayant un passage limité, une deuxième approche de distribution consiste à partitionner une infrastructure suivant un schéma hiérarchique, où chaque élément de cette hiérarchie posséderait sa propre base de données et son propre bus de messagerie. Cette approche pourrait être qualifiée de distribution hiérarchique.

Ce chapitre a pour but de détailler les deux approches *à plat* et *hiérarchiques*, qui sont actuellement utilisées pour assurer un fonctionnement distribué d'une infrastructure OpenStack.

4.2 Distribution à plat

Le projet OpenStack a adopté une architecture sous forme de services, où chaque aspect d'une infrastructure de Cloud Computing est géré par un service dédié. Chaque service d'OpenStack (Nova, Glance, Cinder, ...) est constitué de plusieurs sous-services (Nova-api, Nova-scheduler, ...) qui sont à leur tour instanciés plusieurs fois sous la forme d'agents (*worker*) sans états internes. Les agents communiquent entre eux par des appels à un système RPC (*Remote Procedure Call*) se basant sur l'échange de messages au moyen d'un bus utilisant le protocole AMQP. Les agents étant sans états internes, les états des principaux services OpenStack sont stockés dans des bases de données communes rendues accessibles à leurs agents respectifs qui butinent dans ces bases, consommant les informations qui s'y trouvent et y apportant des modifications dont la portée est globale à tous les agents. Mis à part quelques services spécifiques (comme Nova-compute), cette architecture permet de faire abstraction de la localisation des agents, et qu'importe le nœud sur lequel les agents travaillent, le partage d'une base de données leur permet d'organiser leurs collaborations.

La distribution à plat d'OpenStack se fait ainsi essentiellement de trois manières :

- **Instanciation des agents sur plusieurs nœuds** afin de répartir la charge de travail

d'un sous-service (tel que Nova-scheduler) sur plusieurs nœuds, chaque instance offrant un point d'accès au service auquel il appartient. C'est le principal de moyen de distribution du fonctionnement d'un service.

- **Synchronisation de la base de données et du bus de messages sur plusieurs (nœuds)** afin que la charge de travail soit absorbée par ceux-ci, plutôt qu'un unique nœud qui pourrait être sujet à un phénomène de saturation. En ce qui concerne la base de données relationnelle, il existe des mécanismes permettant de la répliquer sur plusieurs nœuds. Cependant, ces mécanismes sont connus pour avoir un passage à l'échelle limité [54]. Il faut noter que la synchronisation du bus se fait de la même manière, car les queues de messages sont distribuées sur plusieurs nœuds et les messages échangés sont répliqués.

- **Utilisation d'un répartiteur de charge de travail** (*load balancer*) permettant de donner l'impression à l'utilisateur qu'il n'y a qu'un point d'accès unique pour chaque service. En réalité le répartiteur de charge s'occupe de router/distribuer les requêtes des utilisateurs sur les différents points d'accès aux services. Enfin, ce composant s'occupe aussi de gérer les pannes et quand un point d'accès n'est plus disponible, les requêtes sont routées sur les autres points d'accès.

La distribution à plat d'une infrastructure OpenStack est une première solution pour permettre de répartir une charge de travail, créée par exemple par une large communauté d'utilisateurs, sur plusieurs nœuds. Étant donné qu'à large échelle les pannes doivent être considérées comme étant la norme plutôt que l'exception, en répliquant les services et les agents sur plusieurs nœuds il est possible de mieux assurer la tolérance aux pannes. Cependant, l'architecture actuelle reposant sur le partage d'une base de données relationnelle et d'un bus de messagerie, l'efficacité d'une infrastructure distribuée à plat est limitée par la distribution de ces deux éléments, utilisés comme vecteurs de collaboration entre les agents des services d'OpenStack.

4.3 Distributions hiérarchiques

Afin d'atteindre des déploiements de grandes tailles, OpenStack offre la possibilité de mettre en place des infrastructures hiérarchiques. Un déploiement OpenStack est alors divisé en plusieurs sous-déploiements de tailles plus petites, qui sont indépendants entre eux. Une structure, située en haut de la hiérarchie, est chargée de maintenir la structure hiérarchique et de distribuer la charge de travail en pilotant les sous-déploiements.

L'intérêt de ces approches hiérarchiques est qu'elles facilitent le passage à l'échelle des éléments critiques qui ne supportent pas la pression liée à la charge de travail imposée par une infrastructure de grande taille, en les distribuant dans chacune des structures de petite taille où ils seront exposés à une charge de travail locale et donc plus faible.

4.3.1 *Cells*

Les Cellules (*Cells*) permettent à une infrastructure de Cloud Computing utilisant OpenStack d'adopter une organisation hiérarchique. Au sommet se trouve une cellule mère qui sert de point d'entrée pour manipuler l'API d'OpenStack (Nova-API), masquant à l'extérieur l'organisation hiérarchique de l'infrastructure. Cette cellule mère maintient les cellules filles au sein de la hiérarchie, et organise le travail de ces dernières en routant les requêtes des utilisateurs à la cellule fille appropriée. À l'exception du service Nova-API qui est réservé à la seule cellule mère, chaque cellule fille contient une infrastructure OpenStack complète, avec sa propre base de données ainsi que son propre bus de messagerie, ce qui permet de limiter la charge de travail soumise à ces deux éléments.

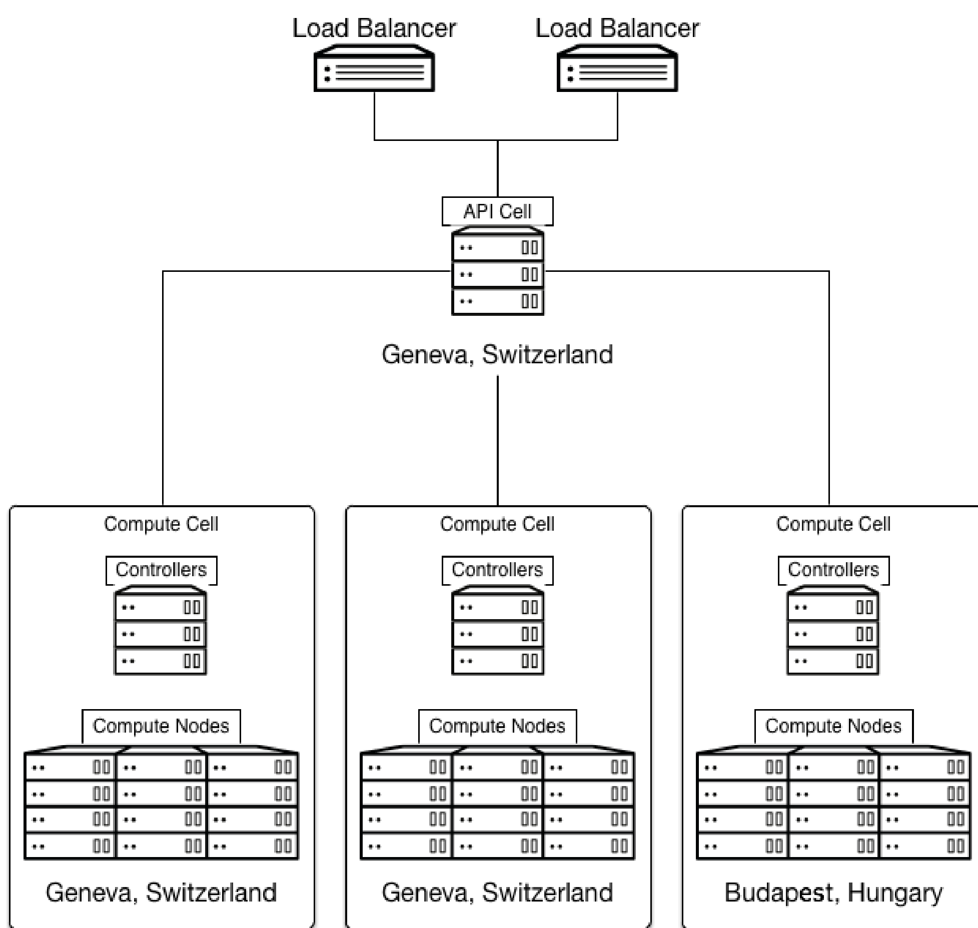

FIGURE 4.1 : Déploiement du CERN utilisant les Cells (extrait de [55]).

Au cours de l'été 2013, le CERN a déployé une infrastructure OpenStack de grande taille qui utilise les Cells [56]. Cette infrastructure contient un peu plus de 5500 nœuds de calcul (*compute nodes*) hébergeant près de 12000 machines virtuelles, réparties entre Genève (Suisse) et Budapest (Hongrie). Comme le montre la figure 4.1, l'infrastructure est divisée en cellules et chaque cellule ne contient que des nœuds appartenant au même

site géographique et une zone de disponibilité (*availability zone*) est créée pour chaque cellule. L'algorithme de placement de machines virtuelles (ordonnanceur) utilisé par le service Nova (Nova-scheduler) a été enrichi de filtres prenant en compte la localisation des nœuds et l'hétérogénéité matérielle des nœuds de calculs.

Bien que l'utilisation des Cells ait été éprouvée sur des infrastructures de tailles importantes, celle-ci impose à une infrastructure OpenStack certaines limitations :

- **La cellule mère n'est pas distribuée**, ainsi toute panne de celle-ci aboutit à une indisponibilité d'une grande partie des fonctions de l'infrastructure. Il n'y a plus d'API exposée ce qui a pour conséquence que les utilisateurs ne sont plus en mesure de transmettre leurs ordres. Les cellules filles n'étant habituellement réservées qu'à l'hébergement de machines virtuelles, tous les services (Nova-scheduler, Nova-network, ...) qui étaient exclusivement hébergés dans la cellule mère deviennent indisponibles.

- **La compatibilité des *Cells* avec les fonctions de Nova**, n'est pas totale dans la version 1 (Cells-v1). N'étant pas l'implémentation par défaut, les Cells-v1 ont nécessité un grand effort pour rendre leur implémentation compatible avec la deuxième version (Cells-v2). Dans les versions récentes d'OpenStack (Kilo), les Cells-v2 sont activées par défaut. Toute infrastructure OpenStack est une Cell, ce qui force la convergence entre les fonctions assurées par les Cells-v2 et celles de Nova.

- **Les *Cells* nécessitent des synchronisations de base de données** entre cellules mères et cellules filles. Avec les Cells-v1, la base de données de la cellule mère agrège les données des cellules filles. Ainsi, quand une image est créée sur une cellule, il faut que cette image soit visible sur toutes les cellules, ce qui implique l'utilisation de mécanismes de synchronisations haut niveau ("à la main") implémentés dans Nova entre les bases de données situées dans les cellules mère et filles. La conséquence est que le découplage entre ces bases de données n'est pas total et conduit à un surcoût d'opérations (plus complexes) et à une source potentielle d'inconsistances des états internes aux cellules si ceux-ci sont mal gérés. Avec les Cells-v2, la situation est sensiblement identique de ce point de vue.

Pour les Cells-v2, les problèmes de distribution soulignés ci-dessus pourraient être résolus en utilisant des mécanismes de réplication bas niveau de bases de données. Un effort a été fait pour séparer les tables qui ont une portée globale à l'infrastructure de celles qui sont locales à une seule cellule. Ainsi en répliquant les tables globales sur toutes les cellules, il serait possible d'éviter de synchroniser les états internes tels que les images de machines virtuelles au niveau de Nova. De plus, ce serait une première pierre vers la distribution de la cellule mère. Néanmoins, les mécanismes de réplication active de base de données ayant un passage à l'échelle limité dans un contexte multisite, cette approche risque d'être peu performante à large échelle.

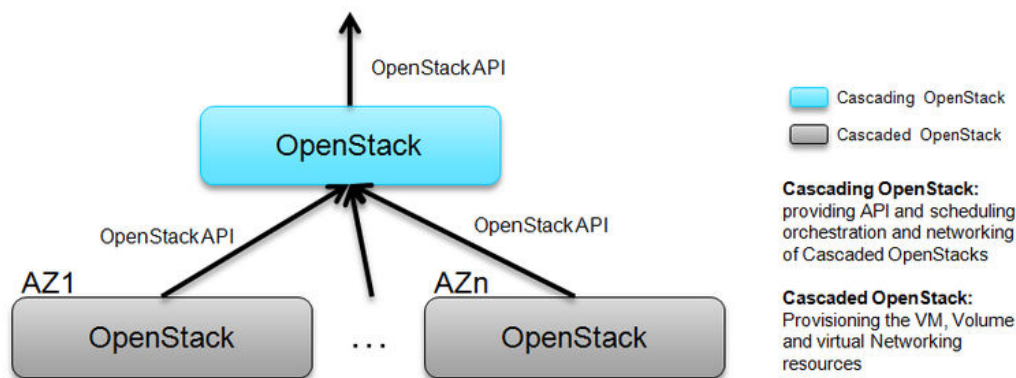

FIGURE 4.2 : Architecture de la solution Cascading OpenStack (extrait de la documentation de cascading OpenStack).

4.3.2 Cascading OpenStack / Tricircle

OpenStack Cascading [1] est une preuve de concept d'un système d'organisation hiérarchique, permettant le déploiement d'infrastructures OpenStack multisite à très large échelle, développée par Huawei dans le cadre du projet **Tricircle** [2]. Les auteurs ont fait le constat qu'une infrastructure OpenStack monolithique n'est actuellement pas adaptée pour gérer une infrastructure multisite ayant pour ordre de grandeur la centaine de milliers de nœuds de calcul et plusieurs millions de machines virtuelles, essentiellement à cause du partage d'un bus de message (RPC) et la base de données relationnelle entre tous les nœuds du déploiement.

OpenStack Cascading reprend une architecture hiérarchique qui est proche de celle utilisée pour les *Cells*. La figure 4.2 (extrait de la documentation de cascading OpenStack [3]) met en évidence qu'une infrastructure qui utilise OpenStack Cascading comprend deux niveaux. Au niveau le plus haut (c.-à-d. **Cascading OpenStack**) se trouve une infrastructure OpenStack en charge d'exposer une API globale, tandis qu'au niveau plus bas (c.-à-d. **Cascaded OpenStack**) se trouvent d'autres infrastructures OpenStack destinées à fournir des ressources informatiques (calcul, réseau, stockage, ...). Les infrastructures sont indépendantes, c'est-à-dire que chacune contient son propre bus de messages et sa base de données dédiée, ce qui permet à ces éléments d'éviter une charge de travail trop forte dans le cas d'un déploiement à l'échelle mondiale.

Quand un appel est fait à l'API globale d'une infrastructure déployée avec OpenStack Cascading, une requête API est interceptée par l'infrastructure *Cascading*, qui a pour charge de déterminer à quelle infrastructure *Cascaded* il faut acheminer la requête. De cette manière, on peut considérer que le niveau *Cascading* sert en fait de Broker (intermédiaire) qui a pour charge d'agréger les APIs exposées par les infrastructures sous-jacentes.

Le premier reproche qui peut être fait à cette approche est qu'elle brise la compatibilité avec le code existant, car elle a nécessité d'importantes modifications dans le code source

[1] https://wiki.openstack.org/wiki/OpenStack_cascading_solution
[2] https://github.com/openstack/tricircle
[3] https://wiki.openstack.org/wiki/OpenStack_cascading_solution

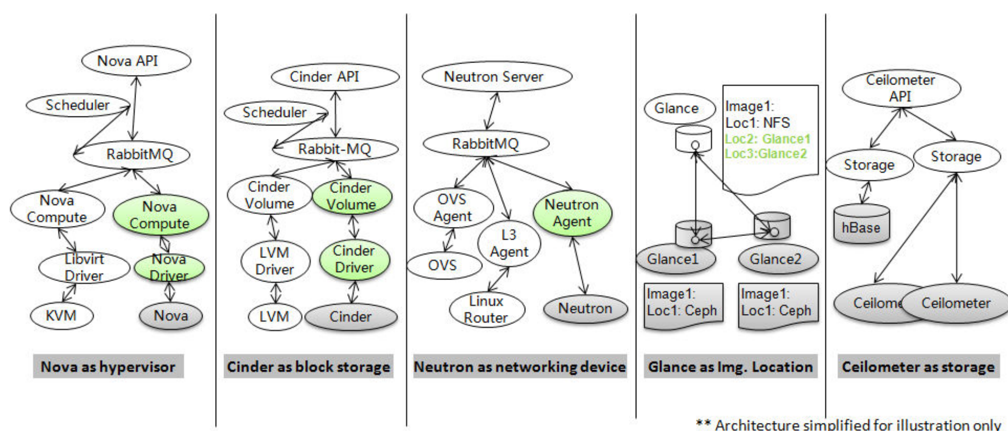

FIGURE 4.3 : Architecture de la solution Cascading OpenStack (extrait de la documentation de cascading OpenStack).

des principaux services d'une infrastructure OpenStack, ce qui est un frein potentiel à son adoption par la communauté.

Comme le présente la figure 4.3 (extrait de la documentation de cascading OpenStack[6]), une infrastructure *cascading* expose une API externe donnant l'impression que toute l'infrastructure est hébergée au sein de l'infrastructure *Cascading*. Cette approche se base sur l'écriture de drivers (pilotes logiciels) permettant à l'infrastructure *cascading* de déléguer toute manipulation d'un élément physique (hyperviseur, stockage d'images, réseaux virtuels, ...) via l'API externe, à une des infrastructures *Cascaded*. Ainsi toute manipulation sur la couche physique est en fait déléguée sur la couche physique d'une infrastructure *Cascaded*.

En reprenant la classification des fédérations d'infrastructures de Cloud Computing introduite dans la section 3.2.4, nous pourrions classer la proposition du projet *Cascading OpenStack* dans la catégorie des fédérations *multitier*, où l'infrastructure *Cascading* agit comme un Broker pour infrastructure de Cloud Computing[4]. Le broker étant une infrastructure OpenStack située sur un seul site, cela signifie que la perte de ce site rend l'infrastructure indisponible. Cette approche pose donc le même problème que pour les *Cells*.

Enfin les infrastructures *Cascaded* étant indépendantes entre elles, la modification de certains états internes d'OpenStack nécessite qu'ils soient répercutés sur les états internes des autres infrastructures *Cascaded*. Dans une discussion comparant l'approche OpenStack Cascaded avec les *Cells* [5], un des membres du projet admet que des opérations comme la création d'images de machines virtuelles nécessitent des opérations de synchronisation pour que les objets représentant des images dans les bases de données soient les mêmes, quelle que soit l'infrastructure *Cascaded*. Ceci peut se faire de différentes manières :

[4] https://wiki.openstack.org/wiki/OpenStack_cascading_solution#Motivation
[5] https://openstack.nimeyo.com/2551/openstack-dev-cascading-cells-summit-recap-and start=10#a_list_title

- **L'utilisation d'un programme de synchronisation** qui à chaque modification d'une partie des états internes d'OpenStack applique la modification sur les états des autres infrastructures. Cette approche exposerait l'infrastructure à des problèmes de consistance en cas de modifications concurrentes.
- **L'utilisation de mécanismes dédiés à la réplication de bases de données** pour que toute modification d'état interne à OpenStack se fasse de manière globale sur toutes les infrastructures. En utilisant un outil tel que Galera [57], cette approche garantirait la consistance des données au prix, limitant cependant le passage à l'échelle à cause du contexte multisite.

De ces deux approches, la première est une approche dite de "haut niveau" qui permet de sélectionner assez précisément les informations à synchroniser. En développant des mécanismes en mode pair-à-pair, il serait possible d'avoir une synchronisation rapide tout en limitant la quantité d'information échangée. Cependant la difficulté de ce genre de mécanismes fait que toute erreur de conception (bug) pourrait être la cause d'inconsistances dans les états internes, ce qui nécessite des mécanismes de réparation et pourrait fortement perturber le fonctionnement de l'infrastructure. La seconde manière consiste en l'utilisation de mécanismes de plus "bas niveau" qui s'appuient sur de la réplication de bases de données comme ceux évoqués dans la section 4.2. L'avantage d'utiliser ces mécanismes est qu'ils permettent d'éviter les inconsistances dans les états internes, au prix cependant d'un surcoût lié à la synchronisation des différentes bases relationnelles (en cas de beaucoup d'écritures parallèles sur les mêmes données les performances baissent considérablement). Le surcoût est accentué dans le cas où les bases de données sont situées sur des sites géographiques différents.

4.4 Résumé

Dans ce chapitre, nous avons vu que le gestionnaire *IaaS* OpenStack était organisé autour d'une architecture modulaire à base de services où chaque aspect d'une infrastructure de Cloud Computing est géré par un service dédié. Ces services sont eux-mêmes composés de sous-services instanciés plusieurs fois, selon un modèle d'agents sans état interne (*state less*) et travaillant sur des états stockés dans une base de données partagée. Cette architecture permet une certaine flexibilité dans l'organisation de l'infrastructure, et permet à OpenStack de fonctionner de manière distribuée. Les états des services sont stockés dans une base de données partagée en les différents agents, et les agents des différents services communiquent entre eux via un bus d'échange de messages lui aussi partagé. Cependant cette base de données pose des problèmes de passage à l'échelle quand la taille de l'infrastructure devient grande, car beaucoup d'agents accèdent et modifient les mêmes données de manière concurrente. La distribution de bases de données relationnelles au moyen de mécanismes de réplication active permet d'augmenter leur passage à l'échelle dans un contexte monosite. À l'inverse, dans un contexte de forte latence réseau, comme c'est le cas dans les infrastructures réparties sur différents sites géographiques, la réplication active de bases de données relationnelles ne permet pas à un bon passage à l'échelle, ce qui limite l'intérêt de tels mécanismes pour distribuer une infrastructure OpenStack.

Des approches alternatives (Cells-v1, Cells-v2 et OpenStack Cascading/Tricircle) ont été développées pour OpenStack, qui consistent en la réalisation de déploiements hiérarchiques. Dans un déploiement hiérarchique, l'infrastructure est alors divisée en petits groupes, chacun hébergeant sa propre base de données et son propre bus de messagerie. Ces groupes sont pilotés par un groupe situé au niveau supérieur, ayant la charge de maintenir la structure hiérarchique et de distribuer du travail aux groupes situés aux niveaux inférieurs. Bien que ces approches permettent de faire des déploiements à plus large échelle, elles requièrent de considérables modifications du code source d'OpenStack pour qu'il puisse fonctionner de manière hiérarchique, et imposent toujours des mécanismes de synchronisation entre les états des différentes bases de données. Avec la solution Cells-v2, seules certaines tables représentant les états internes sont synchronisées. Enfin pour éviter tout fonctionnement dégradé de l'infrastructure en cas de panne du groupe supérieur, le fonctionnement de ce dernier doit être réparti au moins sur plusieurs nœuds, et si possible sur plusieurs sites géographiques. Un tel fonctionnement oblige à faire appel des mécanismes de réplication de bases de données, tels que ceux évoqués plus haut, qui passent difficilement à l'échelle dans un contexte multisite.

Contributions de la thèse

5

Discovery : vers une infrastructure IaaS massivement distribuée

Nous avons vu que pour satisfaire la demande toujours plus grande en ressources de Cloud Computing, tout en réalisant des économies d'échelle, les gros fournisseurs de Cloud Computing ont construit des mégas centres de données qui concentrent la production des ressources de calcul. Ces mégas centres sont eux-mêmes concentrés dans des zones où l'énergie est bon marché, permettant de réduire les coûts liés à l'approvisionnement électrique du matériel et aux besoins en refroidissement.

En plus des problèmes de tolérance aux pannes en cas de désastres, certaines limitations inhérentes à ce modèle de méga centre de données freinent l'adoption du Cloud Computing. L'hébergement de services informatiques sur des infrastructures situées dans d'autres pays peut exposer l'utilisateur à des risques liés à des conflits de juridiction (législation européenne contre législation américaine par exemple). Le fait que beaucoup d'utilisateurs sont éloignés des centres d'hébergement des services qu'ils consomment, implique une forte latence qui peut pénaliser leur utilisation. Ces limitations peuvent repousser l'utilisation du Cloud Computing dans des domaines d'application tels que le *Cloud Gaming* et le *Big Data*.

Pour répondre au problème de la forte latence réseau liée à la concentration des gros centres de données en quelques endroits, le concept de réseau de nanos centres de données géographiquement éparpillés a été proposé par [1]. Un tel réseau permettrait d'héberger des ressources de calcul proches leurs utilisateurs, permettant d'augmenter la couverture en utilisateurs avec une faible latence. Cette proposition converge avec le modèle d'Edge Computing, présenté dans la section 3.3, qui s'intéresse lui aussi au rapprochement des services de Cloud Computing avec leurs utilisateurs. Avec ces approches, un gros centre de données serait partitionné en plusieurs petits centres de données hébergés dans des zones urbaines où la demande en ressource est importante, et sera donc éparpillé sur un territoire. Or la force du modèle de production des ressources de calcul concentrée dans

des mégas centres de données est que cela permet de faire des économies d'échelle en mutualisant les coûts d'entretien ainsi que les coûts en personnel. Ces mutualisations ne sont pas évidentes dans le cadre d'un réseau de nanos centres de données. De plus, la difficulté d'écrire un système permettant l'exploitation d'une infrastructure géographiquement éparpillée semble remettre en cause l'approche des nanos centres de données hébergés à l'extrémité du réseau.

L'initiative **Discovery**, menée conjointement par des équipes de recherche de l'INRIA et des partenaires industriels, s'est intéressée à la faisabilité d'une infrastructure de Cloud Computing déployée sur un réseau de nanos centres de données géographiquement distribués. Cette initiative s'est concentrée entre autres sur le problème de la mutualisation des coûts qui semble à première vue difficile à faire dans le cadre d'un réseau de nanos centres de données. La suite de ce chapitre développe l'approche qui a été proposée où le LUC-OS, un gestionnaire *IaaS*, est capable d'exploiter une infrastructure de Cloud Computing reposant sur un réseau de nanos centres de données déployés au-dessus du réseau d'un opérateur de télécom, ce qui est la thématique qui articule les contributions de ces travaux de doctorat.

5.1 Un réseau de nanos centres de données dans les points de présence

La mutualisation des coûts en entretien et en personnels semble difficile avec une infrastructure à base de nanos centres de données géographiquement éparpillés, car nécessitant d'acheter certains éléments matériels sur chacun des sites (alimentation électrique, refroidissement) et impliquant d'avoir plusieurs équipes techniques dans le cas où l'éloignement entre les centres de données est important.

Ce modèle devient intéressant dans le cas où cette infrastructure est déployée au-dessus du réseau existant, comme ce serait le cas si les centres étaient placés dans les points de présence d'un opérateur de télécommunication. En premier lieu il devient possible d'utiliser les locaux déjà mis en place et de tirer parti des équipements existants (réseau, électrique, refroidissement, ...), ce qui n'est pas le cas lors de la construction d'une nouvelle infrastructure réseau. Ensuite, la mutualisation peut aller encore plus loin en réutilisant les équipes techniques déjà responsables de l'entretien des points de présence, en étendant leurs compétences à l'entretien des nanos centres de données. La mutualisation est donc possible quand on envisage une infrastructure composée de nanos centres de données déployés au-dessus du réseau d'un opérateur de télécommunication.

L'hébergeant de nanos centres de données dans les points de présence des opérateurs de télécommunication permettrait d'établir une fourniture de ressources de Cloud Computing qui soit géographiquement distribuée, ouvrant la voie à une infrastructure décentralisée et plus proche de ses utilisateurs. Ainsi, quand un désastre surviendrait dans une zone géographique donnée, seuls les nanos centres de données qui y seraient localisés deviendraient inaccessibles, les autres nanos centres de données pourraient continuer à fonctionner en bénéficiant des propriétés de distribution et de résilience propres aux réseaux d'opérateurs de télécommunication. Outre, la localisation d'un nano centre de données

dans un point de présence d'opérateur permet d'y héberger des services qui auront une faible latence avec les utilisateurs reliés à ce point de présence, et donc constituer un réseau de ce genre de petits centres de données permettrait de déployer des services pouvant tirer parti de cette faible latence (*Cloud Gaming*, vidéo à la demande, ...) avec beaucoup d'utilisateurs, ce qui n'est pas vraiment possible avec les services hébergés dans les mégas centre de données concentrés à quelques endroits. Ce genre d'infrastructure touche clairement les objectifs du modèle d'*Edge Computing*, comme introduite dans la section 3.3.

Malgré les avantages indéniables que peut présenter une telle infrastructure de Cloud Computing, il reste le problème du système logiciel qui permettrait son fonctionnement. Les gestionnaires *IaaS* sont conçus pour exploiter des infrastructures hébergées dans des centres de données centralisés, ou distribuées sur quelques endroits en adoptant un schéma hiérarchique comme introduit dans la section 4.3. Dans le cas d'un réseau de nanos centres de données, une approche hiérarchique ne serait pas optimale, car il n'est pas possible à très large échelle de mapper une organisation des ressources structurée (hiérarchie) au-dessus d'une organisation des ressources non structurée (un réseau opérateur) sans perdre en efficacité. Pour garantir son efficacité, un réseau de nanos centres de données nécessite donc un gestionnaire *IaaS* nativement distribué, dont l'organisation reflète l'infrastructure réseau sous-jacente, permettant un fonctionnement optimal, en se basant par exemple sur une organisation *à plat* comme introduit dans la section 4.2.

Enfin le gestionnaire *IaaS* devrait avoir un fonctionnement capable de tirer avantage d'une grande distribution géographique. Pour améliorer son efficacité, il pourrait prendre en compte les propriétés de localité réseau, permettant ainsi de mettre en place une plateforme LUC (*Locality base Utility Computing*) déployée au-dessus d'un réseau de nanos centres de données, eux-mêmes distribués sur un réseau d'opérateurs de télécommunication, et exploités par le LUC *Operating System* (LUC-OS) [58].

5.2 Le LUC-OS : un gestionnaire *IaaS* massivement distribué

Pour permettre l'exploitation d'une infrastructure composée d'un réseau de nanos centres de données éparpillés sur un réseau d'opérateur de télécommunication, que nous appellerons maintenant infrastructure LUC (*Locality based Utility Computing* - LUC), l'initiative Discovery [58] propose de développer le LUC-OS (LUC *Operating System*), un gestionnaire *IaaS* qui serait en mesure de lancer des machines virtuelles (*VMs*) sur une infrastructure LUC et d'assurer leurs interconnexions. Le LUC-OS proposerait un ensemble de mécanismes de haut niveau permettant d'exploiter une infrastructure LUC de la même manière qu'un gestionnaire *IaaS* gère une infrastructure déployée dans un centre de données classique. Celui-ci serait en charge de répartir les ressources de calcul utilisées par des services sur l'ensemble d'une infrastructure LUC, de manière à ce qu'elles soient proches des utilisateurs de ces services. Bien que le LUC-OS fournira des APIs pour que les services puissent exploiter les caractéristiques matérielles et géographiques de l'infrastructure LUC, ces aspects seront transparents pour les utilisateurs finaux. Les

mécanismes du LUC-OS seraient semblables aux mécanismes *IaaS* évoqués dans la section 2.3.2 et permettraient de couvrir les aspects à gérer pour transformer une collection de nanos centres de données en une infrastructure LUC, c'est à dire une infrastructure de Cloud Computing aux capacités comparables à celles hébergées dans les mégas centres de données.

Une première version du LUC-OS, qui s'occuperait de l'hébergement de machines virtuelles, pourrait se contenter d'avoir les services suivants :

- **Gestionnaire de machines virtuelles** : en charge de la gestion du cycle de vie des VMs (configuration, ordonnancement, déploiement, démarrage/suspension, suppression).
- **Gestionnaire d'images** : en charge de la gestion des fichiers gabarits destinés à la création de VMs (c.-à-d. images de VM).
- **Gestionnaire réseau** : en charge de fournir et de maintenir de la connectivité entre les ressources de l'infrastructure LUC. Il s'agit principalement de la création de réseaux virtuels pour les VMs d'une même allocation (*Tenant*) et de permettre l'accès aux utilisateurs situés en dehors de l'infrastructure.
- **Gestionnaire de stockage** : fournit des dispositifs de stockage persistant aux VMs.
- **Outils administratifs** : fournissent aux utilisateurs des interfaces logicielles qui permettent d'exploiter et de paramétrer l'infrastructure LUC.
- **Gestionnaire d'information** : s'occupe de suivre les activités ayant lieu sur l'infrastructure, afin de mettre en place des mécanismes d'audit de performance et de facturation des ressources consommées par les utilisateurs.

Le principal défi dans le développement du LUC-OS provient du fait qu'il faut garantir que chacun de ces services pourra fonctionner de manière décentralisée dans un contexte fortement distribué. Réécrire ces services en partant de zéro semble être une tâche titanesque et un non-sens scientifique, et bien que les mécanismes des systèmes *IaaS* actuels n'aient pas été conçus pour exploiter une plate-forme LUC distribuée, leurs principes de fonctionnement ne seraient pas modifiés en cas d'augmentation du degré de distribution. Ainsi, une réécriture complète obligerait à redévelopper des pans entiers de code dont la logique serait identique à celle des mécanismes *IaaS* existants, ou en des termes familiers reviendrait à *réinventer la roue*.

5.3 Adapter OpenStack pour utiliser les bases de données non relationnelles

À la lumière de la section précédente, l'initiative Discovery propose de reprendre des mécanismes issus de systèmes *IaaS* qui auraient fait leurs preuves et de limiter les modifications aux parties qui ne sont pas adaptées à un fonctionnement sur une infrastructure

LUC. Le gestionnaire OpenStack présenté dans la section 2.3.3 est un bon candidat. Il utilise une architecture modulaire organisée en services. Les mises en œuvre de ces services ont été faites sur le principe *Shared-nothing*, où un service est instancié plusieurs fois sous la forme d'agents qui ne partagent pas d'états entre eux, mais plutôt travaillent à partir de données stockées dans des bases de données relationnelles partagées entre tous les agents. De plus, ces agents collaborent par échanges de messages au moyen d'un bus AMQP qui est partagé par tous les agents de l'infrastructure OpenStack. Bien que cette architecture assure en théorie un bon passage à l'échelle, celui-ci est pénalisé par le fait que la base de données relationnelle et le bus de messagerie soient partagés globalement par tous les agents de tous les principaux services d'OpenStack. Il est possible de mettre en place des mécanismes de *haute disponibilité* (High Availability - HA) pour permettre le passage à l'échelle de la base de données et du bus de messagerie, mais les techniques recommandées par OpenStack (réplication des bases de données relationnelles et du bus AMQP) ne fonctionnent bien que dans un contexte d'infrastructures monosites ou faiblement distribuées.

Le LUC-OS visant à être un gestionnaire *IaaS* décentralisé pilotant une infrastructure LUC distribuée sur un réseau d'opérateur télécom non hiérarchique, l'utilisation de la technique de distribution à plat présentée dans la section 4.2 conviendrait au LUC-OS, à condition qu'elle permette un bon passage à l'échelle. Jusqu'à présent, celle-ci se basait sur des mécanismes de haute disponibilité pour bases de données relationnelles qui ne sont pas appropriées pour le contexte multisite d'une infrastructure LUC. S'il était possible de mieux gérer le passage à l'échelle la base de données dans un contexte multisite, l'utilisation d'OpenStack comme base de départ du LUC-OS deviendrait évidente, ce qui permettrait d'envisager le début du développement d'un gestionnaire capable d'exploiter une infrastructure LUC.

En ce qui concerne le bus de messagerie partagé, OpenStack utilise RabbitMQ, un bus à base de Broker centralisé qui utilise le protocole AMQP, pour mettre en place un système RPC (*Remote Procedure Call*). Grâce à son mode *cluster*, RabbitMQ peut fonctionner de manière distribuée dans une configuration hautement disponible (High Availability - HA). Plusieurs nœuds vont héberger chacun un agent RabbitMQ et les queues de messages seront répliquées sur chacun des nœuds. Bien que ce mode ait l'avantage d'être simple, il a l'inconvénient d'être très sensible à la latence réseau ce qui pénalise son utilisation dans un contexte multisite massivement distribué comme avec une infrastructure LUC. Cette limitation est bien connue de la communauté des queues de messages distribuées [59]. En ce qui concerne OpenStack, des pistes ont été proposées afin de résoudre ce problème, comme le fait de s'appuyer sur un bus de messages sans *Broker* en s'appuyant sur ZeroMQ [60].

Suite à l'effervescence de la communauté OpenStack autour de l'ajout du support de ZeroMQ, nous estimons qu'il est préférable de concentrer nos efforts sur le problème du passage à l'échelle de la base de données utilisée par OpenStack pour stocker ses états internes. Bien que la documentation d'OpenStack conseille l'utilisation de mécanismes de réplications de bases de données permettant de distribuer la base de données sur plusieurs hôtes, ces mécanismes affichent de mauvaises performances dans les situations multisites. Les bases de données non relationnelles (*Not only SQL* - NoSQL) se basent sur des pro-

priétés qui sont plus adaptées à un contexte multisite, ont un meilleur passage à l'échelle et proposent souvent des mécanismes natifs de réplication des données. Ainsi les tables de hachage distribuées (*Distributed Hash Table* - DHT) et les systèmes clés/valeurs (*Key/-value stores* - KVS) construits au-dessus du concept de DHT, tel que Dynamo [61], ont fait la preuve de leur efficacité en ce qui concerne le passage à l'échelle et la tolérance aux pannes. Une des premières actions de l'initiative Discovery est d'étudier le remplacement des bases relationnelles par des bases non relationnelles dans le cas des services vitaux à OpenStack.

5.4 Résumé

Dans ce chapitre nous avons abordé la question de la mise en place d'une infrastructure LUC, c'est-à-dire une infrastructure de Cloud Computing massivement distribuée qui servirait d'alternative aux mégas centres de données. Une telle infrastructure, si elle correctement exploitée, présenterait l'avantage d'être tolérante aux pannes et aurait une meilleure couverture géographique des utilisateurs, ce qui aurait pour conséquence de réduire la latence réseau qui peut exister entre ces derniers et les services qu'ils consomment. L'implémentation d'une telle infrastructure s'appuierait sur des nanos centres de données éparpillés sur un territoire et reliés entre eux au moyen d'un réseau très rapide.

La réalisation d'une telle infrastructure semble à première vue casser les économies d'échelle possibles avec les mégas centres de données via la mutualisation des équipements d'approvisionnement en énergie, de refroidissement ainsi que la centralisation des équipes techniques. L'initiative Discovery estime que mettre en place une infrastructure LUC composée de nanos centres de données déployés dans un réseau d'opérateur de télécommunication permettrait réutiliser les infrastructures réseau déjà déployées, et donc de réduire les coûts en évitant l'investissement dans de nouveaux liens réseau, et en mettant en place des mutualisations (partages de locaux, moyens techniques, équipes d'entretien, ...) entre l'infrastructure de calcul et l'infrastructure réseau existante.

Il reste la question du développement du gestionnaire LUC-OS, en charge de l'exploitation d'une infrastructure LUC. Afin de ne pas réinventer la roue, l'initiative Discovery propose de s'appuyer sur les mécanismes proposés par le projet OpenStack. Les services d'OpenStack dont le comportement ne serait pas adapté à un fonctionnement distribué seraient modifiés de manière à exploiter une infrastructure LUC.

Les implémentations des services d'OpenStack se basent sur une architecture où chaque service est composé d'agents sans état interne, qui communiquent entre eux par échanges de messages sur un bus AMQP et en modifiant des états stockés dans des bases de données relationnelles partagées. Bien que théoriquement cette architecture puisse massivement passer à l'échelle, celui-ci est limité par le passage à l'échelle du bus de messages et des bases de données relationnelles, qui sont peu adaptés à un fonctionnement massivement distribué sur plusieurs sites géographiques, où la latence réseau entre les sites peu être forte. Des actions étant en cours pour adapter le bus de messagerie à un fonctionnement plus distribué, nous avons décidé de nous s'attaquer au problème de la base de données centralisée, dont la seule implémentation utilise une base de données relationnelle, en lui cherchant des alternatives qui passent à l'échelle dans un contexte fortement distribué sur

plusieurs sites. Une des pistes pourrait être l'utilisation de bases issues du mouvement NoSQL, notamment les moteurs de stockage de type clés/valeurs.

Support des bases clé/valeur dans Nova

Dans le chapitre 5, nous avions évoqué l'initiative *Discovery* qui propose d'implémenter le concept de nanos centres de données mis en réseaux en les déployant dans les points de présence appartenant à un opérateur de télécommunication. Ce réseau de nanos centres serait exploité par un système *IaaS* spécialement conçu pour ce type d'infrastructure, le LUC-OS, en s'appuyant sur des mécanismes ayant fait leurs preuves, afin de limiter l'effort de développement et donc de pouvoir se concentrer sur les mécanismes dont la valeur ajoutée est importante en ce qui concerne le fonctionnement distribué du LUC-OS. Le système OpenStack a été choisi, car il s'appuie sur une architecture à base de services, où chaque service est instancié plusieurs fois sous la forme d'agents qui travaillent sur les états internes des services, sans être responsables de leur stockage. Les états sont ainsi stockés dans une base de données relationnelle partagée et les agents collaborent entre eux par échange de messages sur un bus partagé. Enfin, la section 5.3 soulignait que bien que cette architecture à base de service puisse avoir théoriquement un bon passage à l'échelle, celui-ci était limité par le fait que le bus de messages RabbitMQ et la base de données relationnelle étaient, mis à part quelques exceptions) partagés entre tous les agents de tous les services. Étant donné que le bus de messages fait déjà l'objet de travaux pour améliorer son passage à l'échelle, nous avons décidé de nous concentrer sur la base de données, et comme OpenStack est composé d'un grand nombre de services, nous avons décidé de commencer par nous attaquer à Nova qui est le service chargé de gérer du cycle de vie des machines virtuelles. Ce chapitre présente les travaux autour du remplacement de la base de données relationnelle utilisée dans le service Nova, par une base de données non relationnelle de type clés/valeurs.

6.1 Accès aux bases de données dans Nova

Nova est le service d'OpenStack responsable du cycle de vie des machines virtuelles. Ce service contient plusieurs sous-services qui sont responsable de différents aspects (réseau, stockage, sécurité, ...) ou de certaines opérations qui interviennent au cours du cycle de vie d'une machine virtuelle (déploiement, migration, ...). Chacun de ces sous-services est instancié plusieurs fois sous la forme d'agents sans états internes, qui communiquent entre eux en se basant sur un protocole d'appel de méthodes distantes (*Remote Procedure Call* - RPC). Ce protocole RPC est implémenté au-dessus d'un bus de messages grâce à la bibliothèque **oslo.messaging**. Les méthodes qui sont appelées via le protocole RPC effectuent des actions qui modifient les états stockés dans la base de données partagée.

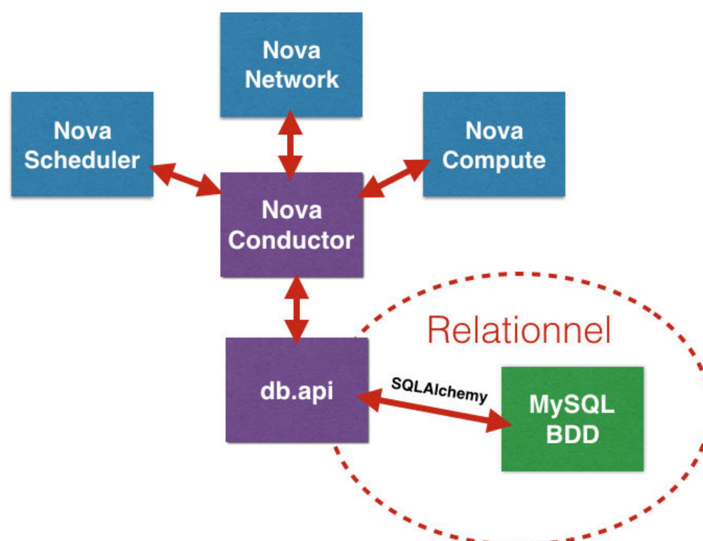

FIGURE 6.1 : Accès à la base de données par les sous-services de Nova.

En ce qui concerne l'accès aux états stockés en base de données, depuis la version *Grizzly* d'OpenStack les sous-services de Nova ne sont pas autorisés à y accéder directement. Comme le montre la figure 6.1, les sous-services de *Nova* passent par un autre sous-service ***nova-conductor***, qui sert de *Proxy* à la base de données. Ce dernier expose ses méthodes au protocole RPC de Nova, qui en étant appelées permettront aux autres sous-services de changer les états stockés en base de données. Ce fonctionnement permet de préserver un couplage faible entre la base de données et les implémentations des sous-services. Enfin, les méthodes de ***nova-conductor*** délèguent tout travail sur la base de données au composant **nova.db.api**, qui s'appuie à son tour sur l'outil de Mapping objet-relationnel (*Object-relational Mapping* - ORM) **SQLAlchemy** pour appliquer les changements sur la base de données.

6.2 Vers un support des bases de données clé/valeur dans Nova

Le service *nova-conductor* étant écrit en langage Python qui intègre le paradigme de la programmation par objet, le code source des méthodes en charge la modification des états en base de données est écrit dans un style objet. Le composant *nova.db.api* définit des interfaces logicielles (nom de méthodes, arguments, conventions sur les types de retour, ...) qui doivent être suivies pour permettre à *nova-conductor* de manipuler une base de données. Actuellement, seule une implémentation utilisant *SQLAlchemy* est proposée. Un appel aux méthodes de ce composant engendre des manipulations sur la base de données relationnelle. Ces manipulations se font au moyen de l'ORM *SQLAlchemy*, qui expose lui aussi ses méthodes et fonctionnalités dans un style objet, leurs appels appels étant par la suite traduits en requêtes SQL. *SQLAlchemy* permet ainsi aux fonctions de l'implémentation par défaut de *nova.db.api* de manipuler la base de données dans un style objet. Une étude du code de ce dernier a confirmé que le code source de l'implémentation par défaut contenait un nombre négligeable de requêtes SQL qui pouvaient être réécrites via l'ORM, et qu'ainsi ce code source était faiblement couplé avec les bases de données relationnelles.

Cette absence de couplage fort permet d'envisager l'utilisation d'autres types de bases de données. Le composant *nova.db.api* centralisant les manipulations de la base de données, c'est donc le composant à cibler pour ajouter le support d'autres types de bases de données à Nova. Pour ajouter le support des systèmes clé/valeur à Nova, il suffirait de fournir une implémentation *nova.db.api* qui suive les interfaces logicielles de ce dernier, tout en ciblant un système clé/valeur. Une première approche serait de repartir de ces interfaces logicielles et d'implémenter chacune des fonctions de façon à ce qu'elle réalise les actions spécifiées sur une base de type clé/valeur. Cette manière de procéder est la plus simple, mais exige de recoder toutes les fonctions existantes (284 fonctions), sachant que l'implémentation par défaut contient 6260 lignes de code. Cette première approche est donc fastidieuse et aboutirait en plus à une implémentation très différente de celle par défaut, qu'il faudrait faire évoluer en parallèle de cette dernière, exigeant beaucoup d'effort de maintenance de code. Avoir deux implémentations qui évoluent en parallèle n'a pas vraiment de sens, car à cause du faible couplage existant entre l'implémentation par défaut et les bases de données relationnelles, des pans entiers de cette implémentation devraient pouvoir être réutilisés.

Développer un ORM alternatif qui proposerait les mêmes interfaces logicielles que *SQLAlchemy*, mais manipulerait à la place des systèmes clé/valeur, permettrait de recycler l'implémentation par défaut. Cette approche présente de nombreux avantages, le premier étant que cela permettrait de limiter les modifications apportées au code source de Nova. Les modifications seraient limitées au clonage de l'implémentation par défaut de *nova.db.api*, ainsi que le remplacement des appels à *SQLAlchemy* par des appels à cet ORM alternatif. De cette manière, la majeure partie du développement serait fait à l'extérieur du code de Nova, ce qui pourrait en plus faciliter un transfert vers la communauté Nova car modifiant très peu l'implémentation par défaut. Le deuxième avantage réside dans le fait qu'en externalisant les manipulations des bases de données de type clé/valeur dans une bibliothèque ORM externe, indépendant de Nova, on permet à d'autres projets

de réutiliser cet ORM. Enfin, si l'ORM alternatif respecte scrupuleusement les interfaces logicielles de *SQLAlchemy*, alors il est raisonnable d'imaginer qu'à terme on puisse se passer de cette deuxième implémentation, car celle par défaut pourrait alors fonctionner avec les deux ORMs. Ce sont tous ces avantages qui ont fait que nous avons privilégié la conception d'un ORM alternatif qui permettrait à *Nova* de s'affranchir des bases de données relationnelles et de leur besoin de mécanismes de synchronisation et de réplication.

6.3 Rome : un *ORM* pour bases clé/valeur

Au cours de ces travaux de doctorat, nous avons développé ***Rome***, une bibliothèque ORM écrite en python qui facilite l'interfaçage entre les bases de données de type clé/valeur et les programmes écrits en Python. *Rome* reprend les interfaces logicielles de *SQLAlchemy* et en reproduit le comportement sur les bases de données clé/valeur. Ainsi, il est possible d'assurer une certaine compatibilité des composants écrits pour *SQLAlchemy* avec *Rome*, ce qui est particulièrement intéressant dans le cadre d'une adaptation de l'implémentation par défaut *nova.db.api* afin qu'elle puisse fonctionner avec des bases de type clé/valeur.

Les ORMs classiques traduisant les appels à leurs méthodes en opérations d'algèbre relationnelle (sous la forme de requêtes SQL), assurer le même comportement pour des bases de type clé/valeur nécessite un certain travail d'implémentation. En effet, les systèmes de type clé/valeur fournissant un ensemble plus restreint d'opérations (manipulation d'une clé, récupération de la valeur associée à une clé, ...) que les bases de données relationnelles, certaines opérations d'algèbre relationnelle ne sont pas nativement disponibles, telles que les jointures et les transactions. Afin de garantir une compatibilité entre *SQLAlchemy* et *Rome*, et pouvoir ainsi réutiliser au maximum l'implémentation par défaut de ***nova.db.api***, *Rome* doit être en mesure de proposer ces opérations manquantes. Celles-ci s'appuieront sur des mécanismes qui les émuleront au-dessus de systèmes clé/valeur, en les recodant ou en utilisant des alternatives plus adaptées à un contexte fortement distribué.

6.3.1 Interfaces logicielles inspirées par *SQLAlchemy*

SQLAlchemy contient de nombreuses classes d'objets python permettant de manipuler une base de données grâce à du code écrit dans un style objet. Chacun des aspects que l'on retrouve dans les opérations des bases de données relationnelles, telles que la création de tables, la sélection et la modification des données contenues dans les tables, se fait en appelant des méthodes exposées dans un style objet par *SQLAlchemy*. Vouloir réimplémenter dans *Rome* chacune des classes d'objets présentes dans *SQLAlchemy* exigerait un gros travail d'implémentation, il apparaît plus intéressant de n'implémenter que celles qui seraient nécessaires. Une exploration du code source de l'implémentation par défaut de *nova.db.api* a fait apparaître que seule la classe ***Query*** de *SQLAlchemy* était utilisée dans cette première. Ainsi, à chaque manipulation de la base de données, une instance de la classe *Query* est instanciée, exposant des méthodes pour ajouter, sélectionner et mettre à jour les données de la base de données. En sus de la classe *Query*, les autres classes utilisées par *nova.db.api* représentent presque exclusivement la symbolique des

requêtes SQL, c'est-à-dire les équivalents des opérateurs (*select*, *and*, *or*, ...) et des fonctions (*count*, *min*, *max*, ...). L'implémentation par défaut de *nova.db.api* ne manipulant que les classes évoquées précédemment, la stratégie optimale d'implémentation de *Rome* serait de limiter celle-ci à la fourniture d'une classe équivalente à la classe *Query* de *SQLAlchemy*, qui appliquerait les mêmes opérations, mais cette fois sur du stockage de type clé/valeur.

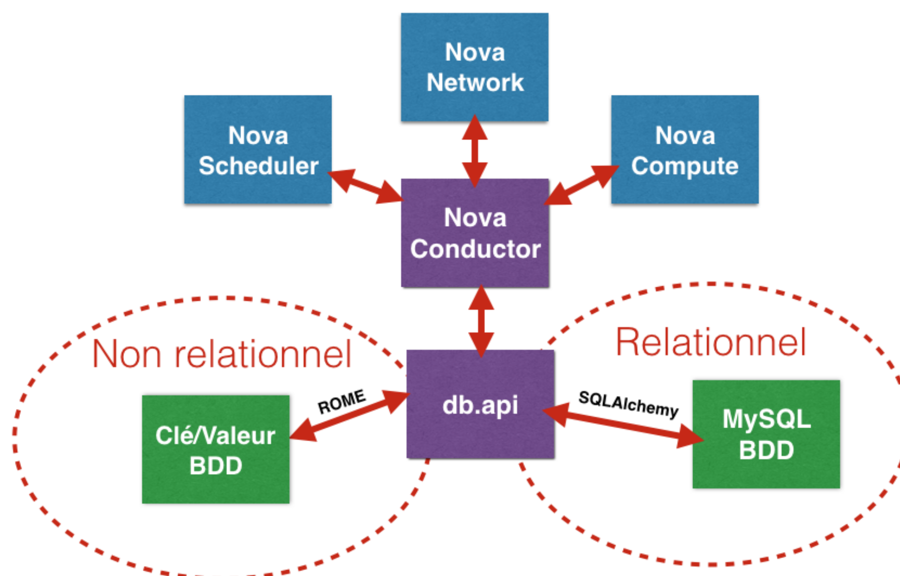

FIGURE 6.2 : Accès à la base de données après l'ajout de Rome.

Cette stratégie de réimplémentation a permis d'écrire rapidement un prototype où les sous-services de Nova utilisent une implémentation de *nova.db.api* fonctionnant pour les systèmes de type clé/valeur. Comme le montre la figure 6.2, le prototype offre une deuxième implémentation du composant *nova.db.api* qui fonctionne avec des bases de données non relationnelles, la première version restant la version par défaut qui travaille sur des bases de données relationnelles. Les modifications apportées à Nova se sont limitées au composant *nova.db.api*, ce qui a permis de ne pas casser la compatibilité avec les autres composants de Nova. De plus, le prototype fonctionnant aussi bien sur l'une ou l'autre des deux implémentations, il est ainsi possible de comparer les deux. Dans cette optique, le composant *nova.db.api* a été instrumenté pour pouvoir collecter des données permettant comprendre et évalue le fonctionnement des deux implémentations.

6.3.2 Représentation des données et marshalling

Un ORM servant de pont entre le monde de la programmation par objet et celui des bases de données relationnelles, il a notamment pour tâche de transformer des données représentant des objets en données stockables dans une base de données. Ce processus qui permet de transformer des données dans un format "objet" vers un autre format adapté au

stockage est appelé *sérialisation* (*Serialization* ou *Marshalling* [62]). Un objet contenant des valeurs dont les types sont complexes (c.-à-d. de types non primitifs) est transformé en une *représentation simplifiée* où ces valeurs sont converties en agrégats de valeurs simples (c.-à-d. de types primitifs). Ainsi, la sérialisation permet d'appliquer à des objets des opérations qui se font d'autres formats, tels que le stockage dans un fichier ou le transfert sur un réseau. Inversement, il existe une opération symétrique dite de *Désérialisation* (*Deserialization* ou *Unmarshalling*). Avec la désérialisation, une valeur produite par un processus de sérialisation d'un objet est transformée en une représentation identique à l'objet initial, au moyen de la consommation de métadonnées. La plupart des ORMs implémentent ces opérations et stockent les données qu'ils manipulent dans un format compatible avec les bases de données relationnelles. Ainsi, dans le cas de *SQLAlchemy* le stockage d'un objet se fait en insérant sa représentation simplifiée dans la table qui correspond à son type.

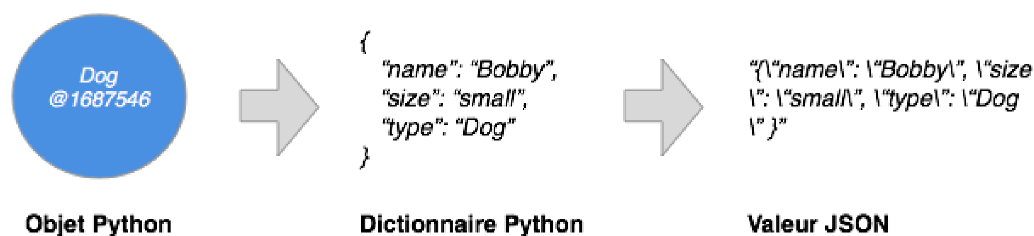

FIGURE 6.3 : Illustration du processus de sérialisation.

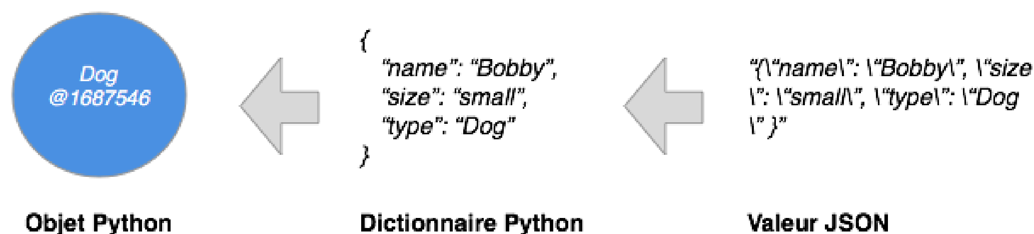

FIGURE 6.4 : Illustration du processus de désérialisation.

Dans le cas des bases de données de type clé/valeur, les valeurs stockées ne sont pas relationnelles, la clé permettant de retrouver une entrée est généralement typée sous forme d'une chaîne de caractère, tandis que les types des valeurs varient du texte brut à une représentation sous forme d'objet dans un format comme le XML ou le JSON. Pour *Rome*, il a été choisi d'utiliser le format JSON pour représenter un objet en base de données. Le processus de désérialisation, illustré par la figure 6.3, consiste en la transformation d'un un objet python en une donnée stockable ou échangeable sur un réseau. Dans le cas de *Rome*, l'objet est dans un premier temps transformé sous la forme d'un dictionnaire python contenant des métadonnées qui ont pour but de faciliter le processus inverse de désérialisation. Ensuite, pour permettre le stockage ou l'échange, le dictionnaire Python est converti en une chaîne de caractère JSON. Cette chaîne de caractère servant de valeur, la clé de stockage utilisée est elle aussi une chaîne de caractère contenant des informations sur la table et l'identifiant de l'objet stocké.

Une des conséquences qu'une opération de sérialisation a sur un objet est que la valeur simplifiée perd le type d'origine de l'objet initial. Un objet candidat au processus de sérialisation possède un type objet, or son équivalent simplifié pour être stocké en base de données est une valeur de type chaîne de caractère JSON non typée. Dans ces conditions, l'opération de désérialisation n'est pas possible sans l'ajout d'information permettant de guider le processus chargé de recréer l'objet initial, en particulier dans le cas où l'ordinateur ayant procédé à la sérialisation est différent de celui faisant la désérialisation. *Rome* ajoute automatiquement des métadonnées à toute valeur stockée en base de données, que ce soit dans le cas d'une valeur représentant une entité de la base de données ou un champ de type objet d'une de ces entités. Ces métadonnées permettent de faciliter la désérialisation en aidant *Rome* à déduire le type cible vers lequel sera traduite une valeur simplifiée dans le format JSON. Dans le cas des entités stockées en JSON, des informations additionnelles, telles que l'identifiant associé à la dernière modification en base de données ainsi que le numéro de version de l'objet (compteur du nombre de modifications faites sur un objet par *Rome*), sont ajoutées dans les métadonnées afin d'aider *Rome* à gérer les phénomènes d'écritures concurrentes.

6.3.3 Support des opérations de bases : création, sélection, modification, suppression

Définition d'une classe d'entité

La définition d'une classe d'entité se fait en définissant une classe Python qui étend les classes *Entity* et *Base* fournie par *Rome*. Celle-ci doit être précédée par l'annotation *global_scope* pour étendre sa portée au-delà du fichier courant. L'exemple ci-dessous illustre comment créer une classe représentant un objet destiné à être stocké :

```
from lib.rome.core.models import Entity, Base, global_scope
from sqlalchemy import Column, Integer, String

@global_scope
class Person(Base, Entity):
    """Represents a person."""

    __tablename__ = 'persons'

    id = Column(Integer, primary_key=True)

    name = Column(String(255))
    zip_code = Column(String(255))
```

Une classe d'entité contient des variables Python définissant les attributs d'objets qui seront stockés par *Rome*. Comme le montre l'exemple précédent, les attributs des classes d'entités se basent sur des types fournis par *SQLAlchemy*, ce qui permet de bénéficier de mécanismes déjà implémentés et donc de ne pas avoir à les réimplémenter dans *Rome*. En ce qui concerne la façon dont les données sont stockées, à chaque classe d'entité correspond une collection d'objets qui sont analogues à la notion de table issue du monde des bases de données relationnelles. Chaque instance d'une classe entité recevra, lors de son

stockage, un identifiant unique et sera stockée dans la collection d'objets correspondant à sa classe d'entité. Le stockage de collections d'objets est disponible dans de nombreuses bases de type clé/valeur (REDIS, Riak, ...) et permet de simplifier la récupération de plusieurs instances issues d'une même classe entité. Dans le cas où un système clé/valeur ne posséderait pas nativement de mécanismes pour stocker des collections d'objets, il serait possible d'émuler cette fonction en utilisant des clés contenant le nom de la table auquel on aurait ajouté l'identifiant et de mettre en place un index secondaire permettant de retrouver tous les objets issus d'une classe entité à partir du nom de la classe entité. Cette approche aurait néanmoins l'inconvénient de nécessiter plus d'opérations.

Cycle de vie d'un objet : création, modification, suppression

Rome reproduit le même processus d'instanciation que celui utilisé par *SQLAlchemy*. Ainsi, le constructeur de la classe entité est explicitement appelé pour créer un objet dont les attributs sont initialisés et publics. C'est au moment de l'appel à la méthode *save* que *Rome* fait une tentative pour stocker l'objet dans le moteur de stockage (système clé/valeur), comme le montre l'exemple suivant :

```
""" Create an instance of Person """
bob = Person()
bob.name = "Bob"
bob.zip = "44300"
bob.save()
```

Lors du premier appel à la méthode *save*, l'attribut *id* est initialisé avec une valeur unique qui lui sert d'identifiant par la suite. Dans l'exemple suivant, nous reprenons le même objet *bob* et modifions un de ses attributs. Un nouvel appel à la méthode *save* permet de propager le changement dans le moteur de stockage :

```
""" Modify bob's zip """
bob.zip = "44000"
bob.save()
```

En plus des opérations de création et de modification, *Rome* permet la suppression d'un objet au moyen la méthode *delete*, comme dans l'exemple suivant :

```
""" Delete bob """
bob.delete()
```

Après l'appel à la méthode *delete*, l'objet est entièrement effacé de la base de données. Il est aussi possible de ne supprimer que le référencement de l'objet grâce à la méthode *soft_delete*. Sa valeur reste stockée dans la base de données, mais elle devient invisible.

Sélection d'objets

L'opération de sélection permet de récupérer des objets contenus dans une base de données. Dans *Rome* cette opération se fait en créant un objet de type *Query* qui contiendra les éléments de la requête qui sera faite à la base de données. Un ensemble de classes entités ou de références à certains de leurs attributs sont passés en paramètres du constructeur de *Query* afin de spécifier les informations ciblées par la requête. L'exemple suivant illustre la création d'une requête sur une classe d'entité *Person* :

```
from lib.rome.core.orm.query import Query

query = Query(Person)
query = Query(Person.id, Person.name)
```

Il est possible d'ajouter des conditions filtrantes aux requêtes en appelant les méthodes *filter* et *filter_by* exposées par toute instance de *Query*. Ces méthodes retournent un nouvel objet *Query* qui reprend les caractéristiques de la *Query* sur laquelle se fait l'appel, tout en prenant en compte les conditions passées en paramètre. Il est ainsi possible d'enchaîner les appels à ces méthodes, comme le montre l'exemple suivant :

```
query = query.filter(Person.zip=="44300")
query = query.filter_by(name=="bob")
```

Une fois la requête définie, celle-ci peut être exécutée grâce aux fonctions *all*, *first* et *count* qui respectivement récupèrent la liste de tous les objets correspondant à la requête, le premier objet correspondant à la requête et le nombre d'objets correspondant à la requête. L'exemple suivant illustre l'utilisation de ces méthodes :

```
person = query.first()
persons = query.all()
count = query.count()
```

6.3.4 Support des jointures et des relations

Les jointures sont un ensemble d'opérations issues de l'algèbre relationnelle permettant de croiser les données issues de plusieurs ensembles. Alors que ces opérations sont nativement supportées par les bases de données relationnelles, leur support est absent des bases de données non relationnelles de type clé/valeur. Pour préserver la compatibilité avec *SQLAlchemy* les jointures ont été implémentées dans *Rome*, et peuvent être appliquées sur des classes ayant des attributs communs :

```
@global_scope
class Dog(Base, Entity):
    """Represents a dog."""

    __tablename__ = 'dogs'

    id = Column(Integer, primary_key=True)

    name = Column(String(255))
    specy_id = Column(Integer)

@global_scope
class Specy(Base, Entity):
    """Represents a specy."""

    __tablename__ = 'species'

    id = Column(Integer, primary_key=True)
    name = Column(String(255))
```

La méthode *join* permet de réaliser une jointure. Celle-ci prend en paramètre la classe d'entité sur laquelle il faut réaliser l'opération de jointure, ainsi qu'un prédicat permettant de préciser comment réaliser la jointure. L'exemple suivant illustre comment réaliser une jointure avec *Rome* :

```
query = Query(Dog).join(Specy, Dog.specy_id==Specy.id)
```

Lors de l'évaluation d'une requête impliquant une jointure, *Rome* récupère une liste contenant les objets des classes d'entités à joindre. L'association entre les objets des différentes classes d'entités est faite par le composant ***lib.rome.core.rows.tuples*** de *Rome* en s'appuyant sur la bibliothèque *Pandas* [1] qui permet de faire de l'analyse de données dans les programmes Python. La figure 6.5 illustre le processus de jointure.

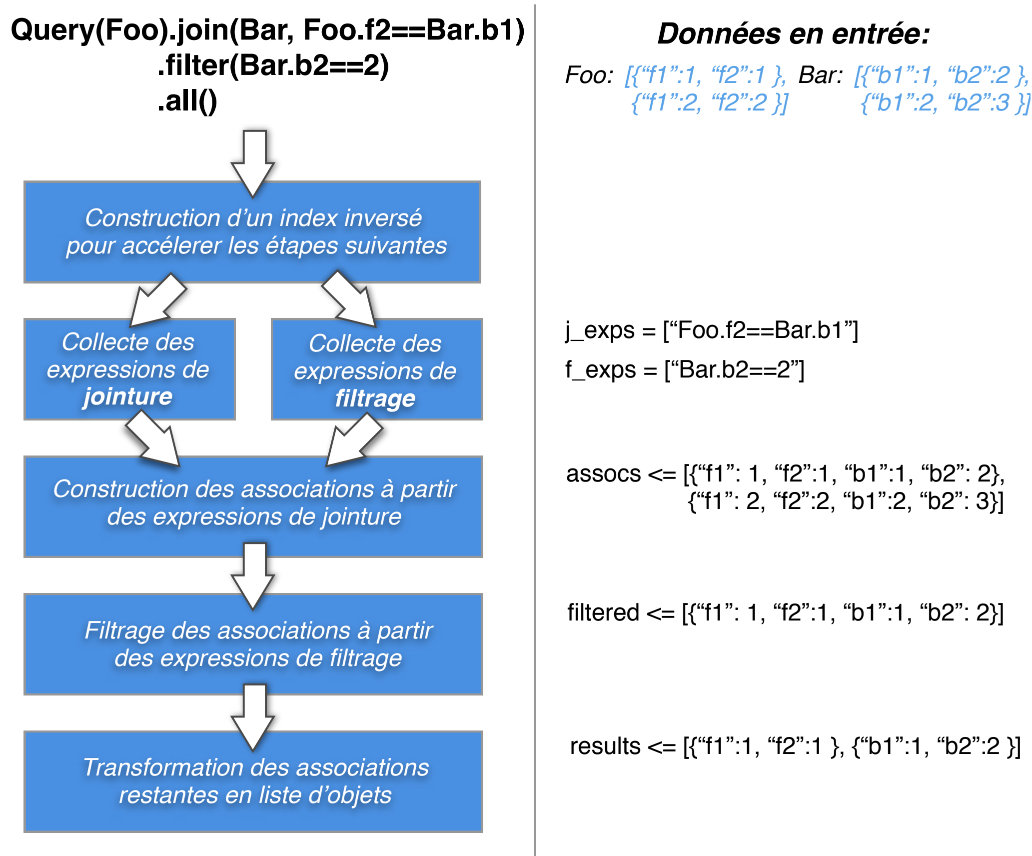

FIGURE 6.5 : Étapes du processus de jointure dans *Rome*.

Enfin, il est possible d'automatiser ces opérations de jointures en définissant un champ de type relation comme champ d'une classe entité. Ce champ relation se traduit par la création d'un objet paresseux lors de l'instanciation d'une classe entité, qui lorsqu'il sera sollicité effectuera une requête *Rome* récupérant les objets correspondant à la requête. Dans l'exemple suivant, la classe entité *Dog* possède un champ relation *specy* :

```
from sqlalchemy import orm
```

[1] http://pandas.pydata.org/

```
@global_scope
class Dog(Base, Entity):
    """Represents a dog."""

    __tablename__ = 'dogs'

    id = Column(Integer, primary_key=True)

    name = Column(String(255))
    specy_id = Column(Integer)
    specy = orm.relationship(Specy, backref="dogs",
                 foreign_keys=specy_id,
                 primaryjoin='Dog.specy_id == Specy.id')
```

De cette manière, toute évaluation d'une expression impliquant un champ relation déclenchera de manière paresseuse une requête retrouvant les objets ciblés par la relation. Ainsi, dans l'exemple précédent une instance de la classe *Dog* possédera un attribut *specy* contenant un objet paresseux qui quand il sera sollicité, chargera dynamiquement l'objet de type *Specy* correspondant à l'expression fournie lors de la définition de la relation. En spécifiant le paramètre *backref* du constructeur d'une relation, il est possible de créer la relation inverse dans la classe cible. Cela se traduit dans l'exemple précédent par le fait que toute instance de la classe *Specy* possède un objet paresseux *dogs* qui lors de son évaluation, récupère la liste des objets *Dog* répondants au critère de la relation. On peut ainsi écrire le code suivant :

```
Query(Dog).first().specy.dogs[0].specy.dogs[1].specy
```

6.3.5 Les sessions : implémentation à base de verrous distribués

Dans le contexte des bases de données relationnelles, une transaction est un mécanisme qui permet l'exécution d'actions atomiques devant être appliquées sur le mode du "tout ou rien". Quand une transaction n'est que partiellement exécutée, c'est à dire quand une de ses actions n'a pu aller jusqu'à son terme, cette première est annulée et toutes les modifications faites au cours de son exécution sont annulées. Les transactions sont supportées par la plupart des bases de données relationnelles (propriétés ACID), mais très peu de systèmes clé/valeur offrent leur support. Dans *SQLAlchemy* les transactions ne sont pas directement accessibles, il faut passer par la classe *Session* qui enrobe les transactions. Des objets peuvent être enregistrés dans une session et quand celle-ci est validée (*commit*) au moyen d'un appel à la méthode *flush* ou quand l'exécution du bloc de code de la session est finie, tous les objets enregistrés dans la session sont modifiés en base de données de manière atomique (tout ou rien). Dans le cas où plusieurs sessions ayant des objets communs sont validées en même temps, seule une des transactions réussira, les autres rencontreront une erreur qui déclenchera une exception de type *DBDeadlock*.

Rome a réimplémenté le mécanisme des *sessions* en s'inspirant de celui qui est fourni par *SQLAlchemy*. De manière à fonctionner avec des systèmes clé/valeur ne supportant pas nativement les transactions, les *Session* de *Rome* utilisent un système de verrous distribués implémentés au-dessus de REDIS, en s'appuyant sur la méthode *hsetnx* qui permet

d'associer une valeur à une clé, si cette dernière n'est pas déjà présente dans la base REDIS. Cette approche a été privilégiée par souci de simplicité, et il existe des algorithmes de verrous distribués plus évolués tels Chubby [63] ou ZooKeeper [64]. Quand une *Session* essaye de valider les changements faits sur les objets qu'elle observe, elle acquiert pour chacun de ses objets un verrou dont le nom correspond à la concaténation de la classe entité et de l'identifiant de l'objet. Si tous les verrous ont été acquis alors la validation de la session est un succès, sinon la session libère les verrous acquis et attend un léger laps de temps avant de retenter une nouvelle fois, jusqu'à épuisement des tentatives qui conduira au déclenchement d'une exception de type *DBDeadlock*. Le nombre d'essais est paramétrable dans le fichier de configuration de *Rome*. L'exemple suivant illustre l'usage d'une session dans *Rome* :

```
from lib.rome.core.session.session import Session as Session

session = Session()
with session.begin():
    floating_ip = Query(FloatingIP).filter(FloatingIP.name=="192.168.1.2").first()
    floating_ip.associated_instance_id = 1234
    floating_ip.used = True
    session.add(floating_ip)
```

6.3.6 Implémentation au-dessus de REDIS

Rome a été conçu pour fonctionner avec des systèmes clé/valeur, nous avons choisi d'utiliser *REDIS* [65] pour la première mise en œuvre. *REDIS* fournit nativement des mécanismes de distribution qui permettent de répartir les données sur plusieurs nœuds, ainsi que des mécanismes de réplications qui permettent la duplication des données sur plusieurs nœuds.

6.3.7 Optimisations

Réduction du nombre d'opérations grâce à la paresse

Dans une infrastructure OpenStack, les serveurs de bases de données sont constamment sollicités par les différents services. Dans le cas de Nova, certaines tâches périodiques chargent de nombreux objets stockés en bases de données. Les premières expériences impliquant *Rome* ont permis d'observer que certains appels aux fonctions API n'utilisaient pas tous les objets extraits de la base de données, ce qui représente une sursollicitation dispensable de la base de données, qui au final baisse les performances d'un déploiement utilisant *Rome*.

Afin de limiter la charge de travail des serveurs de bases de données, des mécanismes de chargement *paresseux* d'objets issus d'une base de données [66] ont été introduits : les références paresseuses. Ainsi une requête sans critères portant sur tous les objets d'une classe d'instance (similaire aux requêtes de type **SELECT * FROM *TABLE*** en SQL) ne va pas chercher les objets de la base de données, mais retourner à la place des références paresseuses vers ces objets. Comme l'illustre la figure 6.6, l'accès à un des champs d'une

référence paresseuse non initialisée va conduire celle-ci à récupérer la valeur référencée dans la base de données et fournir la valeur souhaitée. En plus de garantir que seuls les objets réellement utilisés seront effectivement chargés, le chargement des objets est retardé au maximum.

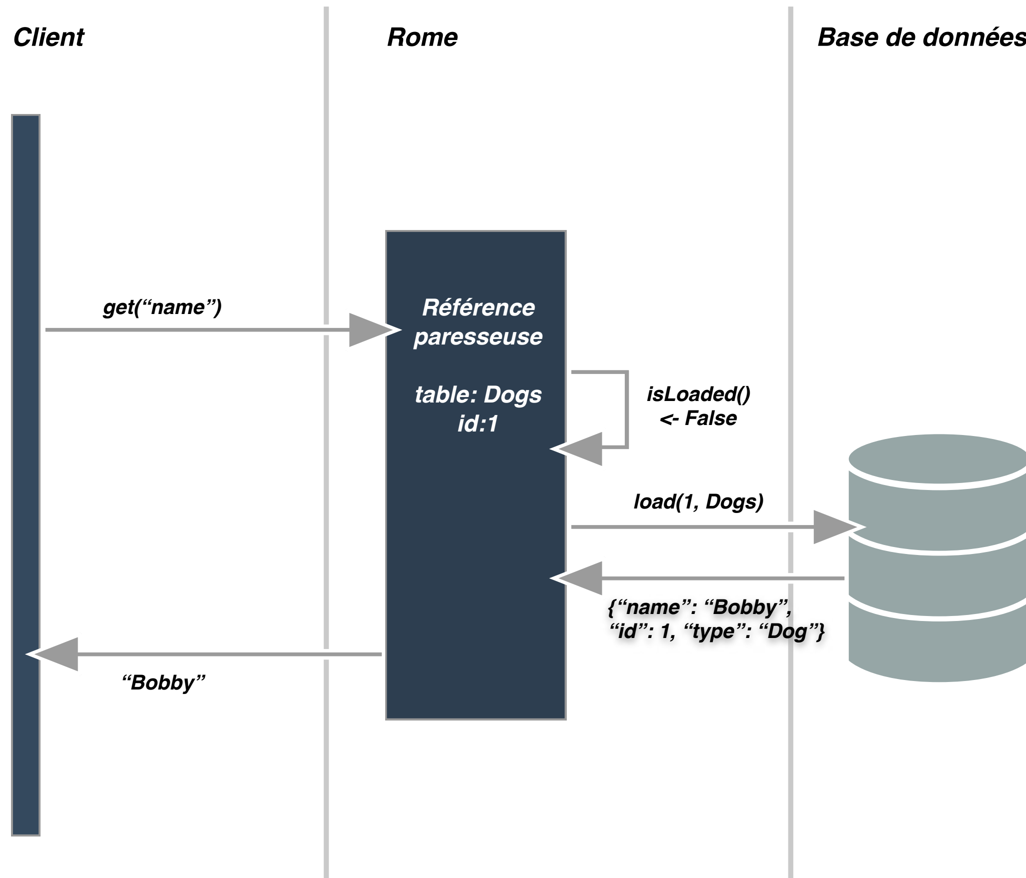

FIGURE 6.6 : Illustration du fonctionnnement d'une référence paresseuse non chargée dans *Rome*. Il n'y a pas de valeur locale de disponible, la valeur est chargée depuis la base de données.

Enfin, quand un programme accède à un attribut d'une référence paresseuse qui a déjà été initialisée, cette dernière ne fait pas de nouvelle requête à la base de données et utilise la valeur déjà chargée, comme le montre la figure 6.7.

index secondaires (ou index inversés)

En plus des clés primaires qui permettent d'identifier un enregistrement dans un système clé/valeur, un système d'index secondaires a été ajouté à *Rome*. Sans index secondaires, la recherche des objets dont un attribut (autre que la clé primaire) satisfait une valeur donnée nécessite de récupérer tous les objets, ce qui implique plus de calcul et de trafic réseau. Les index secondaires [67] permettent de s'attaquer à ce problème en maintenant une correspondance entre des valeurs d'attributs d'objets et les objets dont ils sont originaires. Cette technique permet de considérément accélérer les requêtes portant sur la

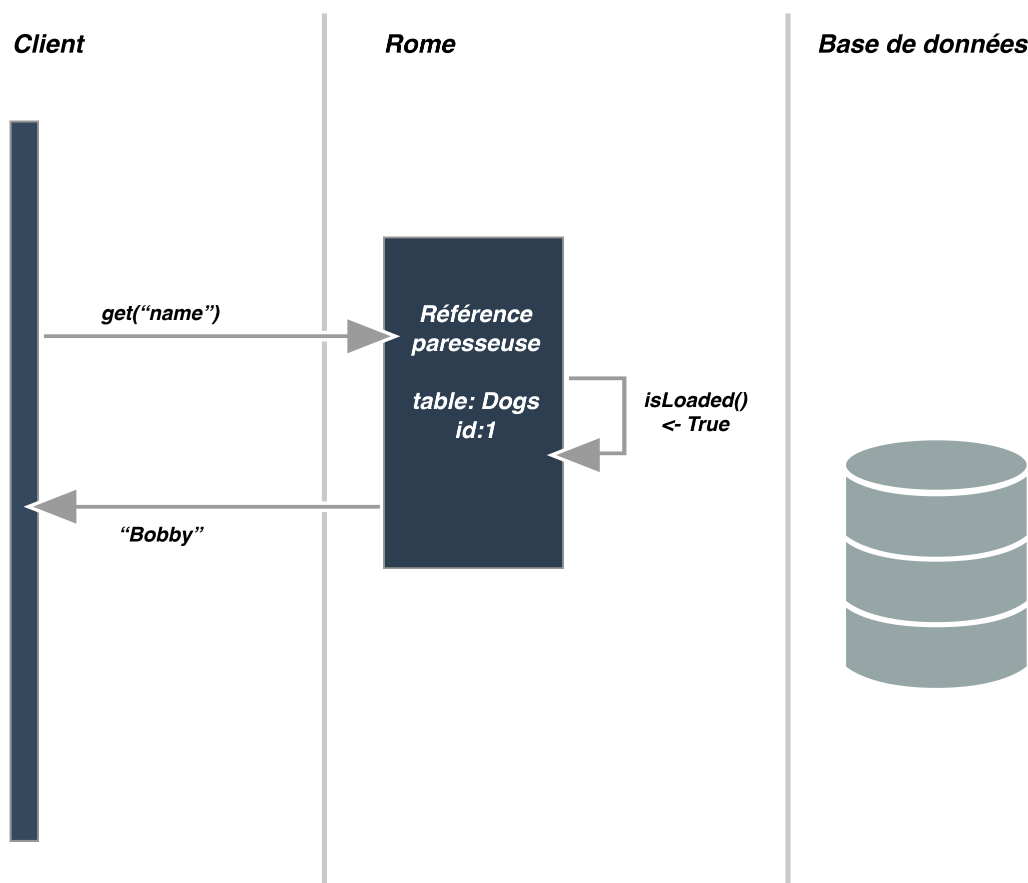

FIGURE 6.7 : Illustration du fonctionnnement d'une référence paresseuse chargée dans *Rome*. Une valeur locale étant disponible, le chargement de la base de données est évité.

recherche d'un attribut ayant une valeur précise (telles que les requêtes de type **SELECT * FROM TABLE WHERE *FIELD* = *VALEUR*** en SQL), au prix cependant d'un surcoût en matière de stockage et de complexité lors de l'ajout ou de la modification d'un objet stocké dans une base clé/valeur. Enfin, il est possible d'associer à une valeur, une liste d'identifiants d'objets. La figure 6.8 illustre le fonctionnement des index secondaires dans *Rome*.

L'utilisation des index secondaires se fait en utilisant le décorateur **secondary_index_decorator** qui permet d'indiquer au moyen d'annotations devant les définitions de classes d'entité, les attributs sur lesquels *Rome* devra construire les index secondaires :

```
from lib.rome.utils.SecondaryIndexDecorator import secondary_index_decorator

@secondary_index_decorator("address")
@secondary_index_decorator("allocated")
class FixedIp(BASE, NovaBase):
    """Represents a fixed ip for an instance."""
    __tablename__ = 'fixed_ips'

    id = Column(Integer, primary_key=True)
    address = Column(types.IPAddress())
```

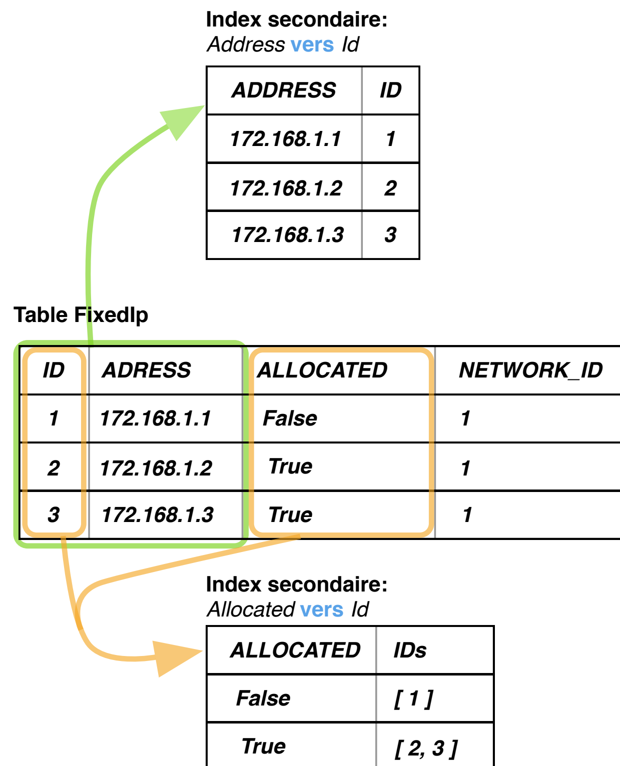

FIGURE 6.8 : Illustration du fonctionnnement des index secondaires.

```
allocated = Column(Boolean, default=False)
network_id = Column(Integer)
```

6.4 Validation sur Grid'5000

Nous avons développé un prototype de gestionnaire OpenStack où le service *Nova* utilise *Rome* pour stocker ses états internes dans un système clé/valeur distribué. Le comportement de ce prototype a été validé expérimentalement au moyen de plusieurs jeux d'expériences. Le premier jeu d'expériences avait pour objectif de mesurer l'impact de nos modifications sur le service *Nova* dans le cadre d'un déploiement OpenStack mono site. Le second jeu d'expériences s'est intéressé au comportement du prototype dans le cadre d'infrastructures multi sites, en comparant les latences des appels API mesurée avec notre version du service Nova à celle mesurée avec la version de base. Enfin, le troisième jeu d'expériences avait pour but de montrer que les mécanismes "haut-niveau" d'OpenStack ne sont pas pénalisés par nos modifications.

Toutes ces expériences ont été réalisées sur le banc d'essai Grid'5000 [68] qui permet aux communautés de la Recherche d'accéder à une grande quantité de ressources de calculs tout en gardant un contrôle précis sur les conditions expérimentales. Nous avons déployé et configuré chacun des nœuds impliqués dans une expérience pour qu'ils utilisent une pile logicielle personnalisée (Ubuntu 14.04, une version modifiée de la pile de développement logicielle d'OpenStack "Devstack" et REDIS dans sa version 3) au moyen

de scripts Python et de la bibliothèque logicielle Execo [69]. Nous insistons sur le fait que les expériences se sont appuyées sur le mécanisme réseau *nova-network* intégré à Nova (c.-à-d. sans utiliser le service Neutron), ce qui nous a permis de simplifier le processus de déploiement de nos expériences. Les autres services nécessaires au fonctionnement d'OpenStack (KeyStone, Glance, ...) ont été déployés un serveur dédié appartenant au premier site géographique (illustré comme le *master node* dans les figures 6.9, 6.10 et 6.11).

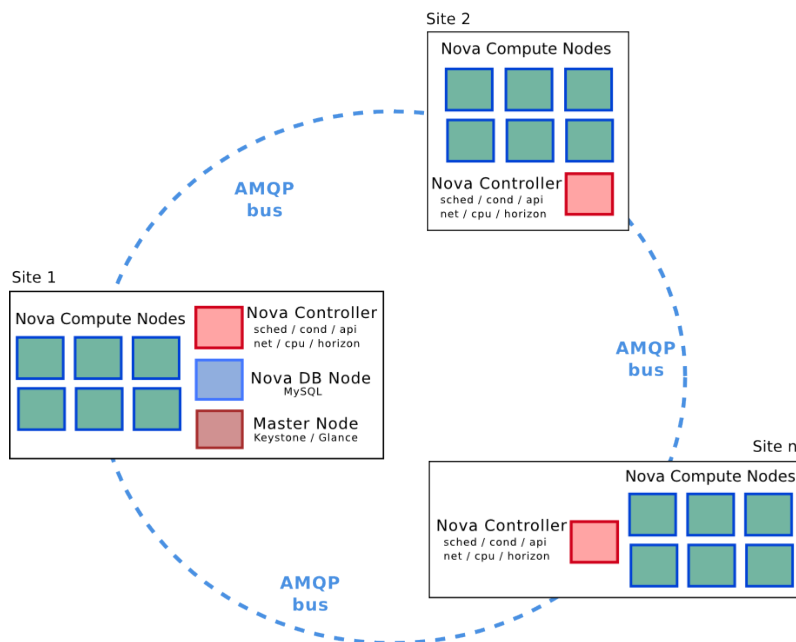

FIGURE 6.9 : Illustration du déploiement fait avec un seul serveur de bases de données MySQL.

6.4.1 Impact de l'utilisation de REDIS par rapport à MySQL

Pénalités en termes de temps de réponse

Les changements faits au code source du service Nova pour permettre le support des bases de données NoSQL, sont susceptibles de modifier sa réactivité. La première raison est qu'un système clé/valeur n'offre pas nativement le support d'opérations telles que les jointures et que le code que nous avons écrit pour ajouter de telles opérations entraine un surcoût de calculs. La seconde raison est liée aux aspects réseau. Contrairement à une base de données localisée sur un nœud unique, une base REDIS peut fonctionner nativement sur plusieurs nœuds, ce qui peut aboutir à ce qu'une seule requête soit à l'origine d'échanges réseau impliquant plusieurs nœuds de bases de données. Enfin, REDIS propose une stratégie de réplication, permettant entre autres de répondre au problème de la tolérance aux pannes, qui est aussi à l'origine d'un surcoût en matière de calcul, car nécessitant des opérations de synchronisation entre les nœuds contenant la base de données *REDIS*.

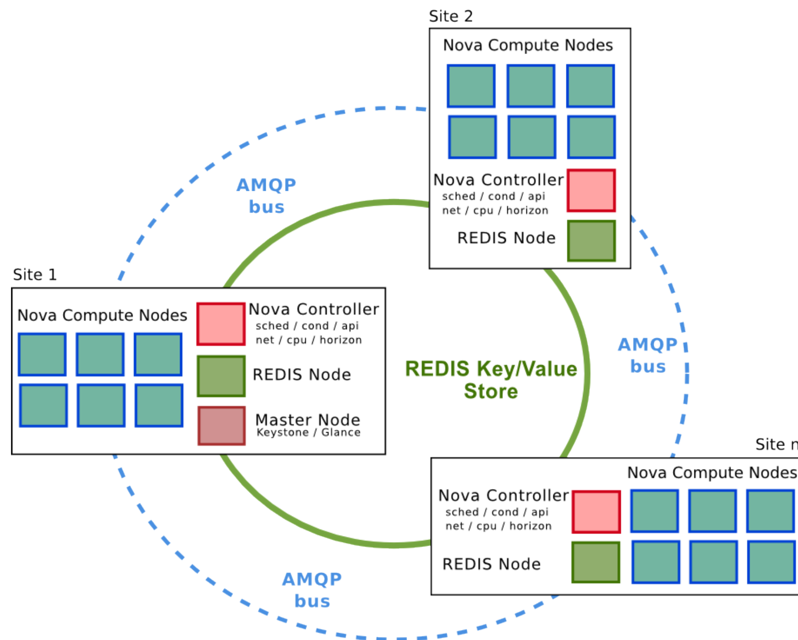

FIGURE 6.10 : Illustration du déploiement fait avec une base de données REDIS distribuée.

TABLE 6.1 : Temps de réponse moyens aux requêtes API pour des déploiements monosite (en ms).

Configuration	REDIS	MySQL
1 nœud	83	37
4 nœuds	82	-
4 nœuds + réplication	91	-

La Table 6.1 compare les temps de réponse moyens pour satisfaire les requêtes API qui sont lancées pendant la création de 500 machines virtuelles sur une infrastructure OpenStack (contenant 1 nœud contrôleur et 6 nœuds de calcul) où Nova stocke ses états dans des bases de données plus ou moins distribuées utilisant REDIS ou MySQL. Alors que la distribution de REDIS sur plusieurs nœuds et l'utilisation de ses mécanismes de réplication ne semblent pas augmenter considérablement le temps de réponse (première colonne), notre version de *Nova* semble moins réactive que celle de base, car le temps de réponse moyen est plus important à première vue (124% d'augmentation). Cependant, l'importance de ce résultat doit être mise dans la perspective de la figure 6.12 et de la Table

TABLE 6.2 : Temps utilisés pour créer 500 VMs pour des déploiements monosite (en sec.).

Configuration	REDIS	MySQL
1 nœud	322	298
4 nœuds	327	-
4 nœuds + réplication	413	-

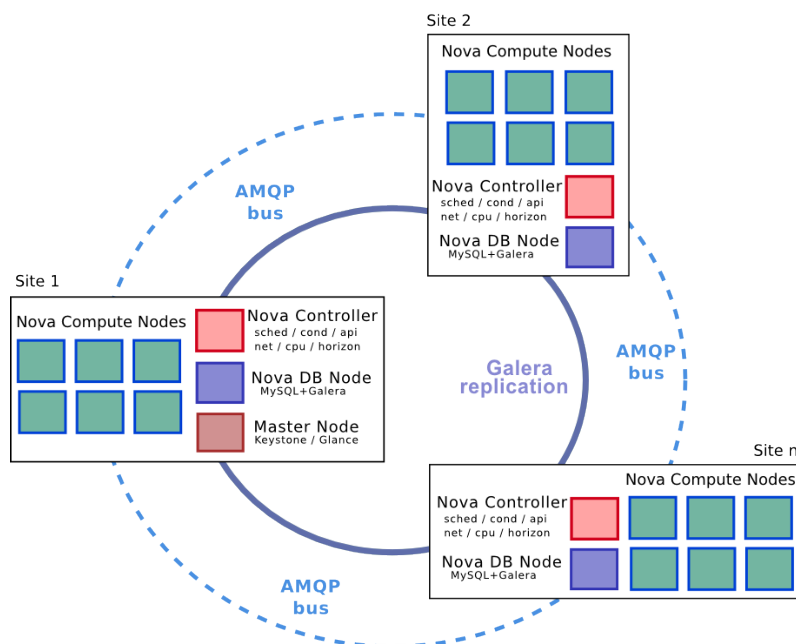

FIGURE 6.11 : Illustration du déploiement fait avec une base de données MySQL distribuée avec Galera.

6.2. La figure 6.12 montre la distribution statistique des temps de réponse de chaque appel à l'API de Nova durant la création de 500 machines virtuelles. D'une part, on observe que pour une grande partie de ces appels (environ 80%), le temps de réponse moyen de notre version de Nova reposant Rome+REDIS est inférieur à la version par défaut de Nova utilisant SQLAlchemy+MySQL. En ce qui concerne les 10% les plus lentes, leurs temps de résolution moyen est supérieur à 222ms avec notre prototype, tandis qu'il est supérieur à 86ms avec la version de base. Cette différence concernant les requêtes les plus lentes explique pourquoi les moyennes enregistrées dans la Table 6.2 sont plus élevées avec notre solution et met en évidence le besoin de pousser plus loin les analyses afin d'identifier ces requêtes et accélérer leur traitement pour qu'il soit plus efficace, ce qui n'a pu être réalisé faute de temps. De manière générale, nous pouvons voir que malgré un traitement plus lent de certaines requêtes, les temps nécessaires pour créer 500 machines virtuelles avec notre version de Nova sont très comparables à ceux par défaut, comme le montre la Table 6.2. En d'autres termes, seules quelques méthodes API semblent être décisives sur le temps de création des machines virtuelles, et elles ne semblent être pas trop pénalisées par nos modifications.

Pénalités en termes de réseau

Nos modifications du service *Nova* ayant pour but de pouvoir exploiter un réseau de micros centres de données impliquant un grand nombre de sites géographiques, la quantité de données échangées sur le réseau est un critère important qu'il faut prendre en compte dans le cadre de l'évaluation de notre approche. La principale raison est que les données que nous stockons avec *Rome* dans le système clé/valeur utilisent un format objet qui né-

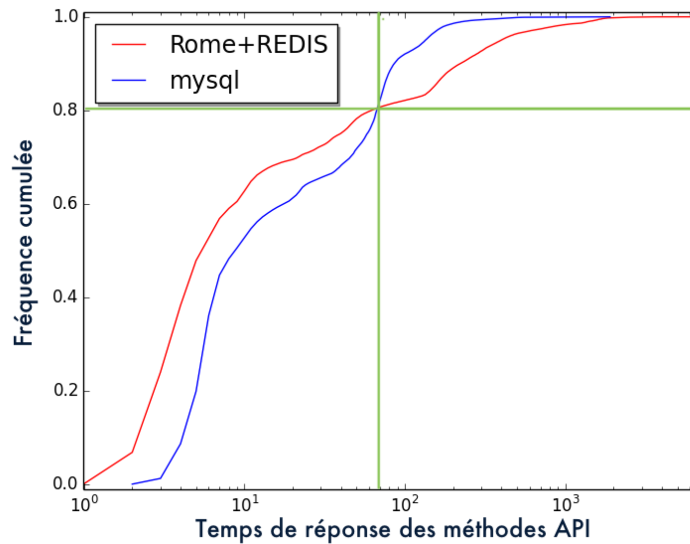

FIGURE 6.12 : Distribution statistique des temps de réponse à l'API de Nova (en ms.).

cessite une phase de sérialisation/désérialisation lors de l'accès et du stockage des valeurs. Pour permettre cette désérialisation, il a été nécessaire d'ajouter des métadonnées, ce qui augmente l'empreinte en matière de stockage et donc augmente la quantité de données échangées entre les nœuds de base de données et les nœuds hébergeant le service *Nova*.

TABLE 6.3 : Quantité de données échangées sur le réseau (en Méga-octets).

Configuration	REDIS	MySQL
1 nœud	2190	1794
4 nœuds	2382	-
4 nœuds + réplication	4186	-

Pour déterminer si le surcoût réseau est à un niveau acceptable, nous avons collecté des métriques réseau au cours des expériences précédentes. La Table 6.3 compare les quantités de données échangées sur le réseau en fonction des configurations de bases de données utilisées. Comme MySQL ne stocke pas des objets sérialisés, ceux-ci sont sérialisés par l'ORM au niveau du client, et donc seules des données brutes sont échangées sur le réseau. Pour cette raison nous considérons la configuration à base d'un nœud MySQL unique comme étant la solution optimale, avec 1764 Mo échangés. Notre solution déployée avec un simple nœud REDIS consomme 2190 Mo, ce qui signifie que le surcoût lié à l'utilisation de Rome et REDIS peut être estimé à 22%. En faisant la même expérience utilisant cette fois un cluster comprenant 4 nœuds REDIS sans avoir activé la réplication des données, nous mettons en évidence que la distribution de REDIS entraine un surcoût de 33% comparé à l'expérience à base d'un unique nœud MySQL. Enfin, quand la réplication des données est activée sur la base d'un facteur de réplication de 1, la quantité de données échangées est supérieure de 76% à celle mesurée sans réplication, ce qui est intuitivement raisonnable.

Cependant, les infrastructures qui ont été déployées contenant différents types de nœuds (contrôleurs, nœuds de calcul, nœuds de base de données, ...), les quantités de données précédemment évoquées prennent en compte tous les échanges sans distinction d'origine et donc ne permettent pas de comprendre l'origine du surcoût réseau.

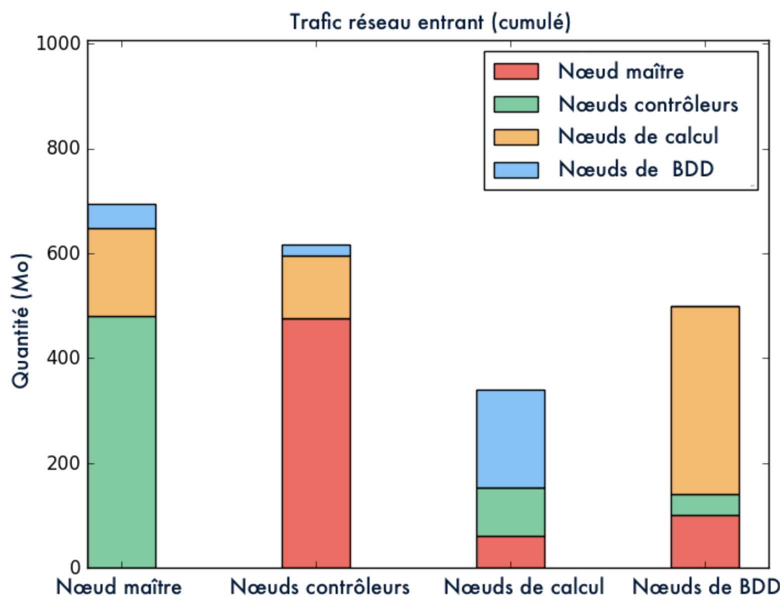

FIGURE 6.13 : Quantité de données échangées en fonction du type de nœuds dans une configuration avec un unique nœud MySQL.

En utilisant l'outil **iftop** [2], il a été possible de rassembler des informations sur l'origine et la destination des messages TCP/IP échangés pendant la création des 500 machines virtuelles, et ainsi avoir ainsi une idée précise de l'origine du surcoût réseau. Les figures 6.13, 6.14 et 6.15 illustrent les échanges de données en fonction de l'origine (axe horizontal), de la destination (barres verticales) et de la configuration de base de données utilisée par Nova. On observe qu'il n'y a pas de différence significative au niveau des flux de données entre les types de nœuds, quel que soit le type du nœud de départ et quelle que soit la configuration de base de données (que ce soit REDIS ou MySQL). Nous pouvons cependant remarquer une légère différence en ce qui concerne les nœuds de base de données, car une comparaison des figures 6.13 et 6.15 confirme que le surcoût réseau vient des nœuds de base de données. Enfin, la figure 6.15 confirme qu'une grande partie du surcoût réseau observé quand on active la réplication des données vient des données additionnelles échangées entre les nœuds de bases de données.

6.4.2 Scénarios multisites

La seconde série d'expériences que nous avons menée consistait en l'évaluation d'un déploiement OpenStack impliquant plusieurs sites géographiques. Le but était de comparer le comportement d'un déploiement OpenStack utilisant un unique nœud MySQL pour

[2] http://www.ex-parrot.com/pdw/iftop/

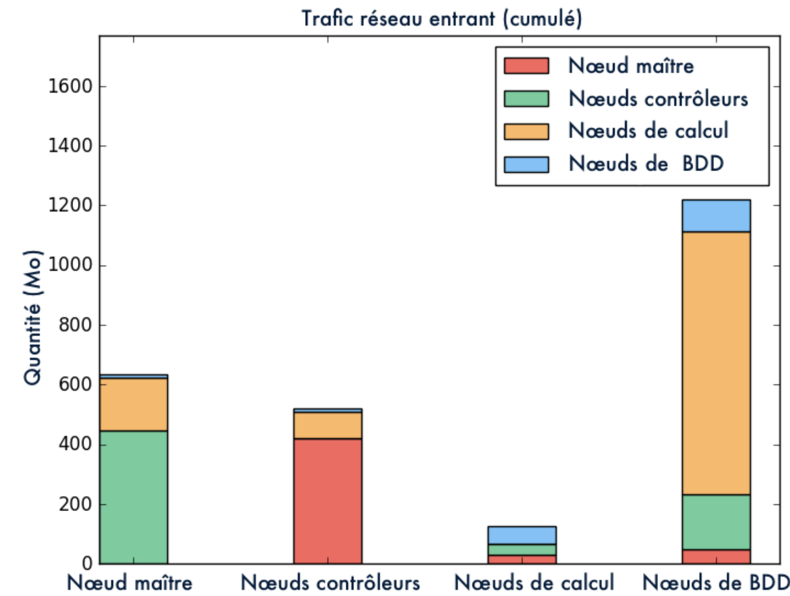

FIGURE 6.14 : Quantité de données échangées en fonction du type de nœuds dans une configuration avec un cluster REDIS de 4 nœuds sans réplication de données.

stocker ses états internes, avec la solution conseillée par OpenStack utilisant *Galera* [57], et notre solution *Rome+REDIS*. Les figures 6.9, 6.11 et 6.10 illustrent comment les nœuds ont été configurés pour chacun des scénarios précédemment cités. Alors que le scénario construit sur un unique nœud MySQL est un non-sens dans le cadre d'une utilisation *en production* comme cela a été évoqué précédemment, l'évaluation de ce scénario nous a permis d'avoir une indication sur ce qui peut être considéré comme le niveau de performances optimal que nous pouvons attendre sur du déploiement multisite. Dans ce dernier scénario, la base de données est déployée sur un serveur unique situé sur un des sites géographiques, sans qu'il n'y ait de mécanisme de synchronisation et donc aucun surcoût lié aux communications avec une base de données distante, contrairement aux scénarios impliquant une fédération de nœuds *REDIS* ou une infrastructure composée de plusieurs instances MySQL synchronisées avec le mécanisme Galera. De plus, réaliser cette expérience utilisant un unique nœud MySQL dans un contexte à large échelle nous a permis de voir les limites d'une telle approche centralisée.

Concernant le protocole expérimental, toutes les expériences multisites ont été conduites sur des serveurs situés sur le même site (Rennes) de façon a assuré la reproductibilité. L'usage de plusieurs sites géographiquement distincts a été simulé en ajoutant de la latence entre des groupes de serveurs, au moyen de l'outil Unix *TC*. Chaque site contenait 1 nœud contrôleur, 6 nœuds de calcul et 1 nœud de base de données si nécessaire. Des scénarios impliquant 2, 4, 6 et 8 sites ont été évalués, aboutissant à des infrastructures contenant un total de 8 nœuds contrôleurs et 48 nœuds de calcul. La latence intersite a été mise dans un premier temps à 10ms, puis à 50ms. Enfin, pour évaluer la capacité à distribuer la charge de travail de chacune des infrastructures déployées, et pour détecter tout problème de concurrence inhérent à l'utilisation d'une base de données non relationnelle,

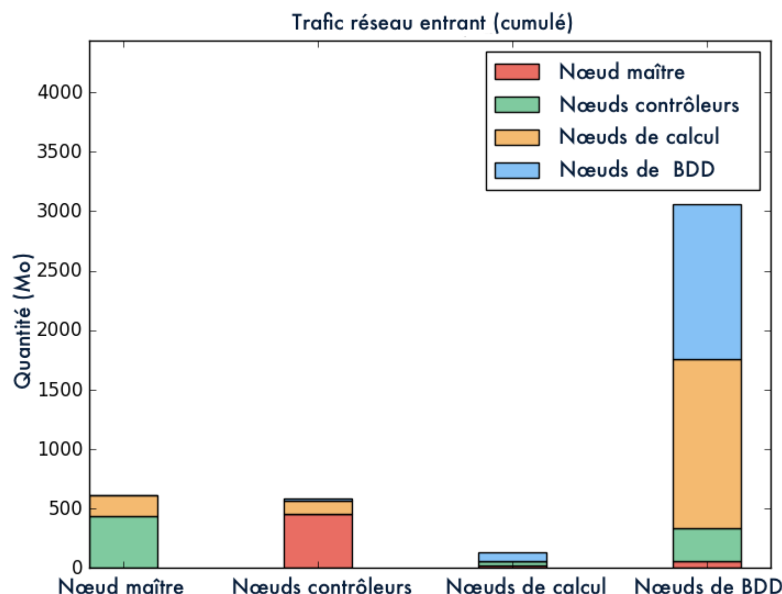

FIGURE 6.15 : Quantité de données échangées en fonction du type de nœuds dans une configuration avec un cluster REDIS de 4 nœuds avec réplication de données.

la création de 500 machines virtuelles a été distribuée équitablement, en parallèle, entre les différents nœuds contrôleurs.

TABLE 6.4 : Temps utilisés pour créer 500 VMs avec une latence inter site de 10ms (en sec.).

Nombre de sites	REDIS	MySQL	Galera
2 sites	271	209	2199
4 sites	263	139	2011
6 sites	229	123	1811
8 sites	223	422	1988

Les tables 6.4 et 6.5 montrent les temps qui ont été nécessaires pour la création de 500 machines virtuelles sur des infrastructures multisites, en faisant varier la latence intersite. Comme prévu, l'augmentation du nombre de sites aboutit à une baisse du temps nécessaire pour créer 500 machines virtuelles. Cela s'explique par le fait qu'un plus grand nombre de sites signifie un plus grand nombre de nœuds contrôleurs et de nœuds de calcul, qui absorbent en parallèle la charge de travail.

Les résultats mesurés dans le cas d'une latence inter site de 10ms, montrent que notre approche prend un temps assez constant pour créer 500 machines virtuelles, ce temps se stabilisant atour de 220 secondes. Alors que les infrastructures à base de serveur MySQL centralisé ont de meilleurs résultats jusqu'à 6 sites, on peut voir que de telles infrastructures atteignent leurs limites à partir de 8 sites. Dans ce dernier cas, l'approche utilisant un serveur MySQL centralisé devient 89% plus lente que notre approche, tandis que la solution conseillée par la documentation d'OpenStack est 891% plus lente que notre approche.

TABLE 6.5 : Temps utilisés pour créer 500 VMs avec une latence inter site de 50ms (en sec.).

Nombre de sites	REDIS	MySQL	Galera
2 sites	723	268	*
4 sites	427	203	*
6 sites	341	184	-
8 sites	302	759	-

Avec une latence inter site de 50ms, la différence de performance entre REDIS et MYSQL est accentuée dans le cas avec 8 sites, car on observe que MySQL devient 151% plus lent que notre approche reposant sur *Rome+REDIS*.

En ce qui concerne Galera, il convient de noter que d'importants problèmes liés aux modifications concurrentes de la base de données sont apparus à partir d'une latence de 50ms, ce qui empêchait la création de beaucoup des 500 machines virtuelles (plusieurs problèmes sont apparus, ce que Nova a considéré comme signe que les machines virtuelles concernées étaient en panne). Ces comportements erratiques sont dus à l'importante latence inter site et à la grosse charge de travail imposée par la création de 500 machines virtuelles en parallèle (pour information, nous avons alors demandé à ce que les machines virtuelles manquantes soient créées de manière séquentielle dans le cas des expériences impliquant Galera avec 2 et 4 sites).

En résumé, en plus de s'attaquer au problème de la difficulté de distribuer la base de données utilisée par Nova, l'association de *Rome* avec REDIS permet à OpenStack d'avoir un fonctionnement qui passe mieux à l'échelle. Ainsi plus on augmente le nombre de contrôleurs, plus on améliore les performances.

6.4.3 Compatibilité avec les fonctions avancées de Nova

Le troisième ensemble d'expériences visait à valider que les fonctions avancées de Nova avaient un comportement correct lors de l'utilisation de la combinaison *Rome+REDIS*. Ainsi, dans le but de minimiser l'intrusion dans le code source du projet OpenStack, nous avons limité les modifications au composant **nova.db.api** comme exposé dans la section 6.2. Ce composant peut être considéré comme la partie de Nova qui interagit de la manière la plus directe avec la base de données. Nous estimons que de limiter de cette façon les modifications faites à ce composant de bas niveau devrait nous permettre de préserver la compatibilité avec les mécanismes de plus haut niveau. Pour valider de manière empirique cette supposition, nous avons réalisé des expériences impliquant plusieurs sites et l'utilisation de mécanisme de " *host-aggregates* / *availability zones* " (il s'agit d'un mécanisme d'OpenStack qui permet de regrouper des nœuds d'une infrastructure OpenStack). Comme pour les expériences précédentes, le scénario consistait en la création de 500 machines virtuelles en parallèle sur une infrastructure OpenStack multisite, utilisant notre solution *Rome+REDIS* avec activation de la réplication des données (le facteur de réplication était de 1). Deux ensembles d'expériences ont été menés, avec un premier ensemble où les nœuds d'un même site virtuel font partie du même *host-aggregate* (et

de la même *availability zone*) et un second ensemble d'expériences menées sur une infrastructure OpenStack multisite utilisant une architecture plate (sans aucune *availability zone*).

Les résultats expérimentaux montrent que les mécanismes de *host-aggregates* se comportent correctement malgré l'utilisation de notre proposition autour de *Rome+REDIS*. Alors que l'utilisation des *host-aggregates* fait que les machines virtuelles sont équitablement réparties sur les différents sites géographiques, le déploiement plat sans utiliser ce mécanisme aboutit à ce que le placement des machines virtuelles n'obéisse à aucune contrainte, ce qui se traduit par une distribution non uniforme de celles-ci les sites de respectivement 26%, 20%, 22% et 32% dans le cas des expériences avec 4 sites.

6.4.4 Premiers essais autour de la localité dans Nova

Dans le cadre d'un infrastructure OpenStack multisite, les communications intersites peuvent être assujetties à d'importantes latences réseau susceptibles de perturber le fonctionnement des services d'OpenStack. Les résultats expérimentaux de la Table 6.5 présentée dans la section 6.4.2 montrent clairement que la solution *Rome+REDIS* était en mesure d'exploiter un réseau d'opérateur télécom ayant une forte latence (50ms). Bien que ce résultat soit intéressant, car indiquant que notre proposition est appropriée pour permettre l'exploitation d'une infrastructure de Cloud Computing distribuée sur plusieurs sites géographiques, il est important de pouvoir comprendre la nature du trafic réseau. La Table 6.6 montre la proportion de communications intersites qui ont été mesurées au cours de l'expérience décrite dans l'expérience de la section 6.4.3, qui impliquait une infrastructure OpenStack utilisant notre solution *Rome+REDIS*, répartie sur 4 sites avec un *host-aggregate* configuré pour chaque site. La Table 6.6 montre clairement qu'une très grande proportion (87.7%) des données échangées au cours de l'expérience se font entre des nœuds situés sur des sites distincts.

TABLE 6.6 : Quantité de données échangées sur le réseau (en Méga-octets.).

Nombre de sites	Total	Inter site	Proportion
4 sites	5326	4672	87.7%

Une analyse plus précise des résultats collectés a montré qu'un quart de ces communications intersites est causé par l'isolation des services autres que Nova (KeyStone, Glance, Cinder, ...) sur un nœud dédié (*master node*). Ainsi des opérations, telles que la fourniture d'une image de machine virtuelle, ont été une source artificielle de trafic inter site non lié à Nova. Cette observation plaide clairement en faveur de la distribution des services restants, comme nous l'avons fait pour Nova. Enfin, les trois quarts restants s'expliquent par le fait que la collaboration entre les agents de Nova se fait au moyen d'un par l'échange de messages sur bus de messagerie commun et d'une base de données qui sont tous deux partagés, à moins que ces deux éléments ne prennent en compte la localité réseau pour éviter les communications intersites, ces dernières resteront à un niveau important.

6.5 Résumé

Au cours de ce chapitre, nous avons étudié le remplacement de la base de données relationnelle centralisée utilisée par Nova pour stocker et partager ses états internes. Historiquement dans Nova, les manipulations de la base de données sont centralisées dans le service **nova-conductor** qui délègue les communications avec la base de données au composant **nova.db.api**. Ce composant bas-niveau expose une interface logicielle, dont le code d'implémentation est actuellement limité aux bases de données relationnelles. La communication entre ce composant et la base de données relationnelle se fait au moyen de *SQLAlchemy*, un outil de mapping objet-relationnel (ORM) qui permet à du code python de manipuler une base de données relationnelle dans du code de type objet.

Pour permettre l'utilisation de bases non relationnelles de type clé/valeur qui offrent un meilleur comportement dans des configurations multisites, nous avons développé *Rome*, un outil qui reprend les interfaces logicielles de *SQLAlchemy* tout en manipulant une base de données de type clé/valeur. L'avantage de cette approche est qu'il permet de reprendre une très grande partie de l'implémentation par défaut de **nova.db.api**, préservant ainsi la compatibilité avec les mécanismes de plus haut niveau de Nova, ainsi que les autres services d'OpenStack. Cette approche a été validée expérimentalement sur Grid'5000 au moyen d'expériences monosites et multisites.

Les expériences monosites ont eu pour objectif de mesurer le surcoût en matière de réactivité qui serait lié à notre implémentation. En effet, certaines opérations telles que les jointures et les transactions n'étant pas nativement supportées par les bases de données de type clé/valeur, il a fallu les recoder dans *Rome*, ce qui est source de calculs additionnels. De plus, la distribution de la base de données sur plusieurs nœuds entrainera plus de trafic réseau que l'utilisation d'une base de données relationnelle centralisée sur un nœud. Les résultats ont montré que dans le cadre de la création de 500 machines virtuelles en parallèle, 80% des requêtes faites sur l'API de Nova étaient plus rapides dans le cas d'une utilisation de *Rome+REDIS* que lors de l'utilisation de *SQLAlchemy+MySQL*. Malgré les 20% restant qui ne tirent pas parti des optimisations apportées par *Rome*, et donc ont un impact négatif sur la réactivité globale de l'API de Nova, les temps pris pour répondre à la création des 500 machines virtuelles sont très proches de ceux mesurés sur l'implémentation de référence.

Les expériences multisites ont consisté à créer 500 machines virtuelles sur des infrastructures OpenStack multisites. Ces expériences ont montré que l'implémentation reposant sur *Rome+REDIS* prenait son sens dans les configurations multisites. Dans une configuration possédant 8 sites géographiques séparés par 10ms de latence réseau, l'implémentation de Nova fonctionnant sur *Rome+REDIS* est 47% plus rapide que celle fonctionnant sur *SQLAlchemy+MYSQL*. Dans une configuration similaire où les sites sont séparés par 50ms de latence réseau, l'implémentation utilisant *Rome+REDIS* est 60% plus rapide. En plus de permettre de constamment réduire le temps utilisé pour créer les machines virtuelles en augmentant le nombre de sites, *Rome+REDIS* permet à Nova de pouvoir stocker ses états de manière géographiquement distribuée et tolérante aux pannes, sans avoir à utiliser de coûteux mécanismes tels que *Galera* qui permet de synchroniser des bases de données relationnelles sur plusieurs nœuds.

Les modifications, faites pour que Nova fonctionne avec *Rome+REDIS*, ne touchant que le composant bas-niveau **nova.db.api**, la compatibilité, avec les mécanismes haut-niveau tels que les *hosts-aggregates* ou les autres services tels que *Rally* [3], est préservée. En mesurant le trafic réseau échangé au cours d'expériences isolant ou non chaque site au sein d'un même *host-aggregate*, nous avons remarqué que la proportion de communications intersites, dans le total des communications échangées, ne diminuait pas. Cette observation signifie donc qu'OpenStack ne tire pas parti des mécanismes de localité réseau pour éviter des communications intersites.

Enfin, ces travaux ont donné lieu à un rapport de recherche [6] et ont fait l'objet de présentations lors de l'OpenStack Summit (Austin 2016) et lors du séminaire *Distributed Cloud Computing* à Dagstuhl en 2015 [5]. De plus, une collaboration est en cours avec Orange pour étendre cette stratégie au service Neutron et en parallèle le code source du prototype est stabilisé par un ingénieur de recherche afin qu'il puisse être utilisé en mai 2017. Enfin, des discussions sont en cours pour intégrer une version stabilisée *Rome* en tant que pilote logiciel pour le projet *oslo.db* d'OpenStack.

[3] https://wiki.openstack.org/wiki/Rally

Prise en compte de la localité réseau : le cas de DVMS

Dans les chapitres précédents, nous avons étudié les solutions existantes en ce qui concerne les infrastructures de Cloud Computing, puis nous nous sommes intéressés à la manière d'utiliser ces solutions dans le cadre massivement distribué des micros centres de données déployés au-dessus d'un réseau d'opérateur de télécommunication. Nous avons proposé d'adapter OpenStack à un tel fonctionnement en lui ajoutant le support des bases de données non relationnelles, qui peuvent être un atout pour améliorer sa distribution. Nos modifications visent un composant bas niveau d'OpenStack, ce qui permet d'améliorer le passage à l'échelle de l'infrastructure sans avoir à modifier ses services. Bien que les performances soulignent le potentiel de telles modifications, nous pensons qu'il faut aller plus loin. La conception actuelle des services ne permettant pas de tirer avantage des propriétés de localité réseau, leur fonctionnement ne peut être optimal. Dans ce chapitre, nous présentons les travaux autour de l'ajout de la prise en charge des propriétés de localité à DVMS, un algorithme distribué de placement de machines virtuelles précédemment proposé par l'équipe ASCOLA, afin d'améliorer son fonctionnement dans un cadre multisite en minimisant les collaborations entre nœuds ayant de fortes latences. Ce travail montre le potentiel de l'exploitation des propriétés de localité dans un cadre massivement distribué.

7.1 Contexte

7.1.1 Ordonnancement de machines virtuelles pair-à-pair

La distribution des infrastructures de Cloud Computing offre des avantages indéniables pour s'attaquer à des problèmes tels que la tolérance aux pannes, la forte latence ré-

seau, mais aussi des problèmes d'incompatibilités juridictionnelles. Cependant, les mécanismes qui étaient jusqu'ici prévus pour fonctionner sur des infrastructures centralisées doivent être revus. C'est particulièrement le cas des mécanismes d'ordonnancement de machines virtuelles qui ont la charge d'assigner celles-ci à des hôtes physiques. Bien que les algorithmes de placement aient connu de grandes avancées, les approches centralisées [70] ne passent pas assez à l'échelle et ne sont pas assez robustes. Des solutions hiérarchiques ont été proposées [71] afin d'améliorer le passage à l'échelle, au prix d'une plus grande complexité. La correspondance (mapping) d'une structure hiérarchique au-dessus d'une infrastructure distribuée sur un réseau d'opérateur de télécommunication, n'est pas une opération naturelle. Un réseau de télécommunication propose généralement plusieurs chemins entre ses nœuds de manière non structurée, y imposer une structure aboutit à une perte d'efficacité ou d'information, ainsi qu'un surcoût en termes d'opérations pour maintenir la structure, si on considère les aspects liés à la tolérance aux pannes. Les algorithmes d'ordonnancement pair-à-pair (P2P) sont de meilleures alternatives quand il s'agit de travailler sur des infrastructures de Cloud Computing distribuées, car ils répondent aux problèmes de passage à l'échelle et de tolérance aux pannes, en se basant notamment sur les techniques de réseaux d'overlay network [7, 72].

7.1.2 DVMS : ordonnancement dynamique de machines virtuelles

DVMS (*Distributed Virtual Machine Scheduler*) est un algorithme permettant l'ordonnancement dynamique de machines virtuelles qui s'appuie sur un fonctionnement pair-à-pair au-dessus d'infrastructures massivement distribuées. Un agent DVMS est installé sur chacun des nœuds de l'infrastructure, et l'ensemble des agents maintient une topologie en forme d'anneau ordonné qui repose historiquement sur Chord [73], ce qui permet de jouir de propriétés de routage et de tolérance aux pannes de ce dernier. Quand un des nœuds de l'infrastructure ne peut garantir les demandes en ressources informatiques de toutes ses machines virtuelles et donc ne peut leur garantir une bonne qualité de service, DVMS démarre une *procédure d'ordonnancement itérative* (*Iterative Scheduling Procedure* - ISP) qui a pour objectif de trouver un meilleur placement de machines virtuelles, c'est-à-dire déplacer les machines virtuelles d'hôtes surchargés vers des hôtes moins chargés. À sa création, une procédure *ISP* ne contient que le nœud surchargé, qui en est le *leader*, et celle-ci va s'agrandir en suivant l'ordre de l'anneau. À chaque fois qu'un nouveau nœud rejoint une procédure *ISP*, un calcul de plan de reconfiguration est effectué pour voir si les membres de l'*ISP* peuvent mieux se répartir la charge de travail en s'échangeant des machines virtuelles au moyen de migrations. Si un plan est trouvé, alors il est appliqué, dans le cas contraire la procédure *ISP* va continuer à croître jusqu'à ce qu'un plan soit trouvé ou jusqu'à ce qu'il n'y ait plus de nœuds. Une des règles immuables permettant de garantir le fonctionnement en concurrence de plusieurs procédures *ISP* est qu'un nœud ne peut appartenir qu'à une seule procédure *ISP* à la fois. Quand deux procédures *ISP* se trouvant dans l'incapacité de grandir se rencontrent, celles-ci fusionnent et un de leurs deux leaders devient le leader de la nouvelle *ISP*. Enfin, quand un des nœuds d'une *ISP* ne répond plus, au bout d'un certain délai celle-ci est dissoute ; dans le cas où le nœud à l'origine de l'*ISP* se retrouve encore surchargé, celui-ci démarrera une nouvelle procédure

ISP.

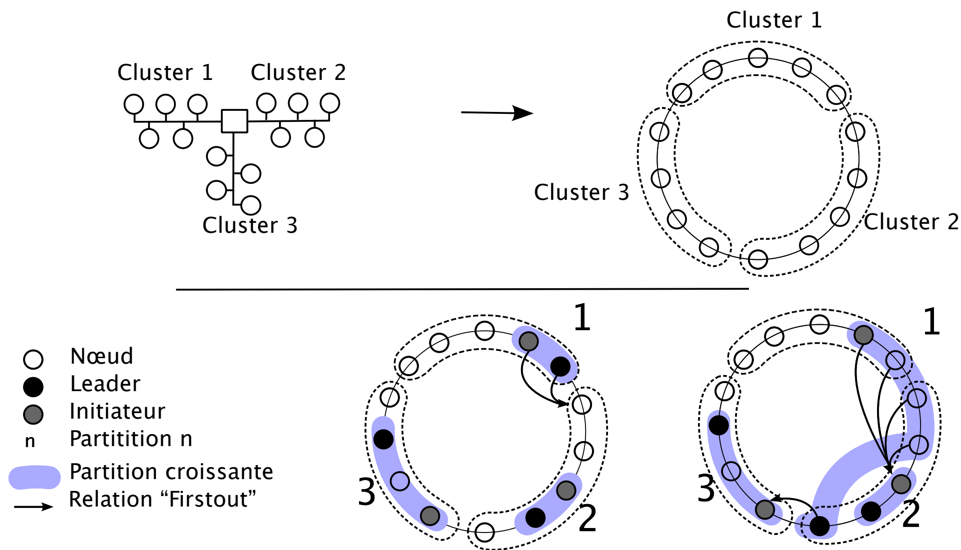

FIGURE 7.1 : Illustration du fonctionnement de DVMS avec un cas comprenant 3 clusters (extrait de [9]).

Une partition *ISP* croissant suivant l'ordre de l'anneau, ses nœuds membres seront choisis sans considération pour les paramètres réseau. Cela peut entrainer des situations inefficaces telles que des migrations de machines virtuelles entre des hôtes éloignés en matière de réseau, qui sont beaucoup plus longues les migrations qu'entre des hôtes proches. Il est possible de construire l'anneau de manière à ce que les nœuds qui sont proches se suivent, comme illustré dans la figure 7.1, mais les nœuds en fin de cluster restent connectés avec des nœuds lointains. Il est aussi possible de mettre en place une approche hiérarchique où un chaque cluster sera rassemblé dans un anneau local avec un super-anneau rassemblant les anneaux locaux [74]. En plus des limitations liées aux structures hiérarchiques déployées au-dessus de réseaux non structurés, cette dernière approche ne prend pas en compte les aspects liés à la localité : quand une procédure *ISP* sortira d'un anneau local, rien ne peut garantir que le prochain nœud soit proche.

7.1.3 Overlay network et prise en compte de la localité

Comme indiqué dans le précédent paragraphe, un des principaux inconvénients des réseaux d'overlay est que ce ceux-ci cassent la topologie physique en connectant directement des nœuds qui ne sont pas proches physiquement. Si l'on met de coté les tentatives de constructions hiérarchiques de réseaux d'overlay prenant en compte la proximité entre les nœuds d'une structure [74, 75, 76] (c.-à-d. la localité), nous pouvons tout d'abord mentionner la prise en compte de celle-ci dans le réseau d'overlay Pastry [77]. Afin de réduire la latence liée au processus de routage, chaque nœud se voit attribuer l'opportunité de remplir ses tables de routage avec des nœuds proches.

Des mécanismes similaires ont été adoptés au sein des réseaux d'overlay non structurés afin de faire en sorte que leurs connexions logiques reflètent la proximité physique qui

peut exister entre les nœuds. Ainsi un nœud cherchant son voisin le plus proche, le fera via des mécanismes de rumeur. Il faut noter que la proximité entre deux nœuds peut être estimée indirectement via certaines métriques telles que la latence réseau que l'on peut mesurer entre deux nœuds [78].

Ces approches nécessitent constamment de maintenir une connaissance des nœuds proches afin de proposer le voisin plus proche possible, au prix d'un surcoût lié à la nécessité de communications périodiques pour échanger de l'information (indépendamment des messages échangés par le système pour fonctionner).

Le réseau d'overlay que nous proposons diffère de ces approches dans le sens où il adopte une approche que l'on peut qualifier de "paresseuse". Elle consiste en la recherche de nœuds proches qui se base seulement sur les informations contenues dans les messages que le réseau d'overlay envoie pour fonctionner. De cette manière, quand un nœud recherche un voisin proche, la qualité de la réponse sera proportionnelle à la fréquence d'échange des messages au sein du réseau.

Notre protocole repose sur le protocole Vivaldi [79] qui permet de détecter des nœuds proches. Vivaldi place les nœuds dans un espace multidimensionnel où chacun d'entre eux se voit attribuer une coordonnée spatiale qui reflète sa position physique. Vivaldi s'appuie sur un protocole reposant exclusivement sur l'échange de messages. Initialement, chaque nœud se voit attribuer une position aléatoire ainsi qu'une liste de connaissances qui vont composer sa vue. Ensuite, chaque nœud commence par estimer la latence réseau existant entre lui et une des connaissances appartenant à sa vue, qui a été choisie aléatoirement. En fonction de la latence mesurée, la distance sera modifiée. Ainsi, si la latence est faible (resp. forte), les nœuds se rapprochent (resp. s'éloignent). Les nœuds répètent ce processus en parallèle de manière indépendante (chacun avec un des nœuds de sa vue), ce qui permet d'améliorer la précision du positionnement des nœuds. Les positions peuvent devenir précises rapidement (en un faible nombre d'étapes) si les nœuds gardent quelques connaissances physiquement éloignées dans leurs vues, et si le réseau sous-jacent n'est pas trop dynamique. Ces connaissances physiquement éloignées peuvent facilement être maintenues.

Enfin, il convient de rappeler que Vivaldi ne permet pas de directement savoir quels nœuds sont proches à l'intérieur du réseau, mais plutôt de reconnaître ceux qui ont des coordonnées proches. Notre réseau d'overlay se base sur l'examen des coordonnées Vivaldi des nœuds qui font partie des connaissances d'un nœud.

7.2 Contributions

L'objectif de cette contribution est de revisiter l'algorithme d'ordonnancement DVMS, afin de lui faire prendre en compte les propriétés de localité. Pour arriver à cette fin, nous nous sommes tout d'abord concentrés sur le réseau d'overlay, puis dans un second temps nous proposons une abstraction qui permet de combiner ce réseau d'overlay avec DVMS tout limitant les modifications du code source de ce dernier.

7.2.1 Réseau d'overlay prenant en compte la localité

Nous présentons ici notre réseau d'overlay qui prend en compte les propriétés de localité. Il se compose de deux couches. La couche basse correspond à l'implémentation du protocole Vivaldi qui permet de rendre les nœuds conscients de leurs positions respectives au sein d'une infrastructure. En utilisant ces coordonnées, la couche supérieure est responsable de la construction dynamique de notre réseau d'overlay. Cette couche s'inspire de l'algorithme de recherche du chemin le plus court de Djikstra, afin de pouvoir collecter un ensemble de nœuds les plus proches d'une position donnée.

Recherche de voisins proches via la vue en spirale

Une fois que la carte Vivaldi est faîte et que chacun des nœuds possède une coordonnée, nous sommes en mesure d'estimer le degré de proximité existant entre deux nœuds en calculant leurs distances sur la carte. Cependant, nous rappelons que la vue de chacun des nœuds ne contient à priori pas la liste de ses voisins les plus proches [1]. Par conséquent, nous avons besoin de mécanismes additionnels afin de localiser un ensemble de nœuds qui sont proches d'un nœud, mais non présents dans un voisinage donné.

Nous avons utilisé une version modifiée et distribuée de l'algorithme de recherche du chemin le plus court de Djikstra, qui s'appuie sur la carte Vivaldi pour construire un tel voisinage.

Considérons que le nœud initial s'appelle n_R. La première étape consiste à trouver un nœud afin de construire une spirale comprenant deux nœuds. Cela peut être fait en sélectionnant le nœud n_i de la vue réseau de n_R, qui devient le deuxième nœud de la spirale. À cet instant, n_R considère n_i comme son successeur, tandis que n_i considère n_R comme son prédécesseur. n_R envoie sa vue réseau à n_i, qui à sa réception commence la construction de la vue en spirale contenant les N nœuds les plus proches de n_R, issus des vues réseaux de n_R et n_i. Cela va permettre à n_i de trouver le prochain nœud qui participera à la construction de la spirale. En faisant l'hypothèse qu'au sein de la vue en spirale de n_i, n_j est le nœud le plus proche de n_R, n_j va être ajouté dans la spirale et deviendra le successeur de n_i. n_j reçoit ensuite la vue en spirale de n_i qui sera complétée par les nœuds proches de n_R contenus à la fois dans la vue en spirale de n_i et la vue réseau de n_j. Cet algorithme est répété jusqu'à ce que le nombre de nœuds demandés par l'application corresponde au nombre de nœuds interconnectés dans la spirale.

Il faut noter qu'il y a un risque qu'à un moment donné la spirale ne puisse plus évoluer, car contenant tous les nœuds susceptibles d'être ajoutés à celle-ci. Ce phénomène peut être facilement résolu en ajoutant quelques nœuds situés à plus grande distance quand la vue en spirale est créée ou mise à jour.

Apprentissage

Appliquant le protocole décrit précédemment, la question de la qualité de la spirale peut se poser dans le sens où les nœuds qui sont actuellement proches du nœud racine n_R peuvent

[1] Nous appelons "vues réseau" ces vues, afin de les distinguer des "vues en spirale" qui seront introduites plus tard

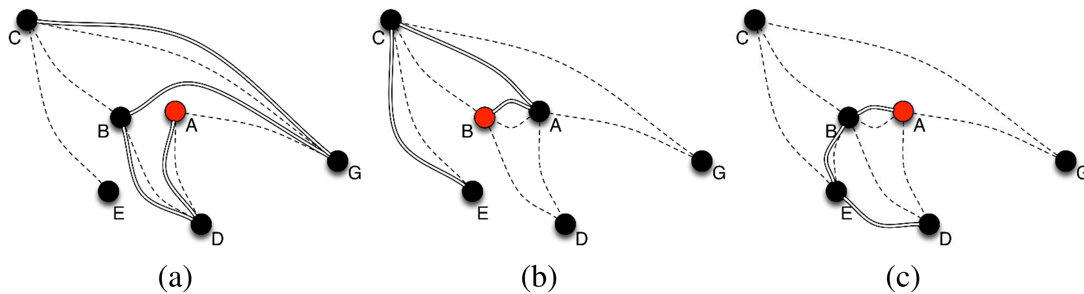

FIGURE 7.2 : Mécanisme d'apprentissage : (a) La vue initiale de chaque nœud est représentée par les lignes pointillées. En considérant ces vues, la spirale obtenue à partir du nœud node A est représentée par la ligne épaisse. Dans cet exemple, la spirale a permis à A et B de se découvrir. (b) Si B commence à construire une spirale, il va commencer par contacter A. La construction de la spirale permet aussi à E et B de se découvrir. (c) Si A se voit demandé de construire une autre spirale, celui-ci va montrer un degré plus important de conscience de la localité.

ne pas y être inclus. Pour améliorer la qualité de la spirale, c.-à-d. réduire la distance moyenne existant en le nœud initial et ses voisins, nous nous appuyons sur un mécanisme d'apprentissage qui se fait sans coût supplémentaire en matière de communication. Ainsi, quand un nœud est contacté afin de devenir le prochain membre d'une spirale, et quand il reçoit la vue en spirale associée, il peut inclure dans cette vue les nœuds qui lui sont proches. Ainsi la qualité de la construction de la future spirale en sera améliorée. Une telle amélioration est illustrée dans la figure 7.2. Il est à noter que l'apprentissage peut aussi être utilisé pour améliorer continuellement les spirales déjà construites. Bien que cela apporte des avantages indéniables, cela se ferait au prix d'imposer de changer dynamiquement les liens contenus dans la spirale, ce qui peut être incompatible avec les applications impliquant de limiter les communications réseau.

7.2.2 PeerActor : Une brique de base pour abstraire les réseaux d'overlay

En tant qu'algorithme d'ordonnancement pair-à-pair, DVMS peut être divisé en deux composants majeurs qui sont (i) Le réseau d'overlay en forme d'anneau et (ii) le protocole en charge de la détection et de la résolution des problèmes d'ordonnancement. Comme notre objectif consiste en la prise en compte du critère de localité sans changer le protocole de DVMS, nous avons conçu l'architecture *Peer Actor*, une brique de base qui nous permet de reconsidérer DVMS en abstrayant le réseau d'overlay sur lequel il s'appuie. De manière générale, l'architecture *Peer Actor* peut être vue comme une couche générique destinée aux services distribués avec un fort niveau d'abstraction, leur fournissant des abstractions réseau ainsi que des communications robustes entre les agents déployés sur chacun des nœuds. En s'appuyant sur l'*API* du *Peer Actor*, les concepteurs de logiciels peuvent se concentrer la construction du service plutôt que d'avoir à gérer les cas d'apparitions/disparitions de nœuds ainsi que les pertes de connectivité réseau.

Du point de vue logiciel, l'architecture *Peer Actor* repose sur des cadriciels logiciels

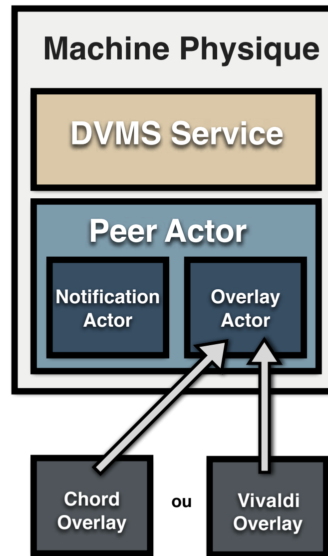

FIGURE 7.3 : DVMS s'appuyant sur l'architecture en Peer actor.

modernes (Scala et Akka) suivant les règles du modèle d'acteurs [80]. Dans ce modèle, chaque instance collabore avec les autres uniquement pas échanges de messages, et la priorité est donnée aux collaborations entre instances qui sont proches en utilisant notre réseau d'overlay prenant en compte la localité réseau (*Locality Based Overlay* - LBO).

Comme illustré par la figure 7.3, l'architecture *Peer Actor* contient deux sous acteurs : le ***Notification Actor*** et le ***Overlay network Actor***. Le *Notification actor* permet aux services de souscrire à des évènements qui vont être déclenchés par d'autres services, comme lors de la détection d'une surcharge de travail d'un nœud ou pour s'occuper de la panne d'un voisin. L'*Overlay network Actor* est en charge d'envoyer et de recevoir des messages grâce au réseau. Afin de comparer de manière équitable l'approche utilisant un anneau et celle utilisant le LBO, nous avons conçu deux mises en œuvre du *Overlay network Actor*. La première mise en œuvre offre un réseau d'overlay s'inspirant fortement de Chord [73], tandis que la seconde propose de s'appuyer sur le LBO décrit dans la section 7.2.1.

7.3 Validation expérimentale sur Grid'5000

L'objectif des expériences était d'estimer l'impact que la prise en compte des propriétés de localité pouvait avoir sur les algorithmes d'ordonnancement. Avec DMVS, une partie significative du temps de reconfiguration est passée à migrer des machines virtuelles, et le temps de migration dépend de paramètres réseau tels que la latence et la bande passante, qui doivent être minimisés. Une façon d'améliorer les performances de DVMS consiste à favoriser les migrations de machine virtuelles entre des hôtes proches, ce qui revient à maximiser le ratio *nombre_migrations_intrasite / nombre_migrations_inter_site*.

Protocole expérimental

Pour permettre la comparaison entre les expériences menées sur DVMS, nous avons développé un injecteur dédié à la simulation d'une charge de travail répartie sur les machines virtuelles. Celles-ci sont réparties équitablement (*round-robin*) sur les hôtes physiques, et les charges de travail de ces premières vont varier dans le temps. L'expérience consiste à changer continuellement la charge des machines virtuelles. La charge d'une machine virtuelle est changée en moyenne toutes les t secondes par l'injecteur en suivant une loi de probabilité gaussienne. Pour respecter cette fréquence, l'injecteur travaille à partir d'une queue d'événements, chacun représentant un temps, un identifiant de machine virtuelle et une nouvelle valeur de charge. Les temps associés aux événements suivent une distribution exponentielle définie par son intensité (λ), et les charges de travail suivent une distribution normale définie par sa moyenne (μ) et son écart-type (σ). Les valeurs des paramètres mis en place pour cette expérience étaient les suivantes :

- λ = nombre de machines virtuelles / 300
- $\mu = 70$
- $\sigma = 30$

Ce qui revenait à ce que la charge de chaque machine virtuelle varie toutes les 5 minutes (300 secondes), et une proportion très significative des machines virtuelles avaient une charge comprise entre 40% et 100%. Afin d'éviter que l'injecteur ne soit perturbé par la charge de travail des hôtes physiques, celui-ci est déployé sur un nœud dédié.

La figure 7.4 montre l'infrastructure utilisée pour réaliser cette expérience. Pour chaque expérience, nous avons utilisé la plate-forme Grid'5000. Nous avons réservé 40 nœuds de calculs éparpillés sur 4 sites géographiques (10 serveurs par sites) et un nœud de service. Les nœuds de calcul étaient utilisés pour héberger les machines virtuelles et DVMS, tandis que le nœud de service faisait fonctionner l'injecteur de charge. Chaque nœud de calcul avait un total de 8 cœurs CPU et hébergeait un nombre de machines virtuelles proportionnel à son nombre de cœurs (nombre de machines virtuelles = 1.3 x nombre de cœurs), pour un total de 416 machines virtuelles pour toute l'infrastructure déployée. Bien qu'un tel nombre semble petit en comparaison des expériences déjà menées sur DVMS [8], l'objectif n'était pas de revalider son passage à l'échelle, mais de se concentrer sur l'étude de l'ajout de la localité réseau dans cet algorithme.

Résultats

1. Maximisation de la proportion de migrations intrasite

   La Table 7.1 compare les taux de migrations intrasite parmi les migrations faites par DVMS au cours de deux expériences : une fois en utilisant l'overlay historique de DVMS et l'autre en utilisant notre réseau d'overlay non structuré prenant en compte la localité (LBO). Cette table montre que l'impact lié à l'ajout de la localité dans DVMS est significatif. En utilisant le LBO, le taux de migration intrasite est de 86.3%, tandis qu'en utilisant un overlay qui ne permet pas de tirer avantage de la localité, le taux de migrations intrasites est de 49.6%.

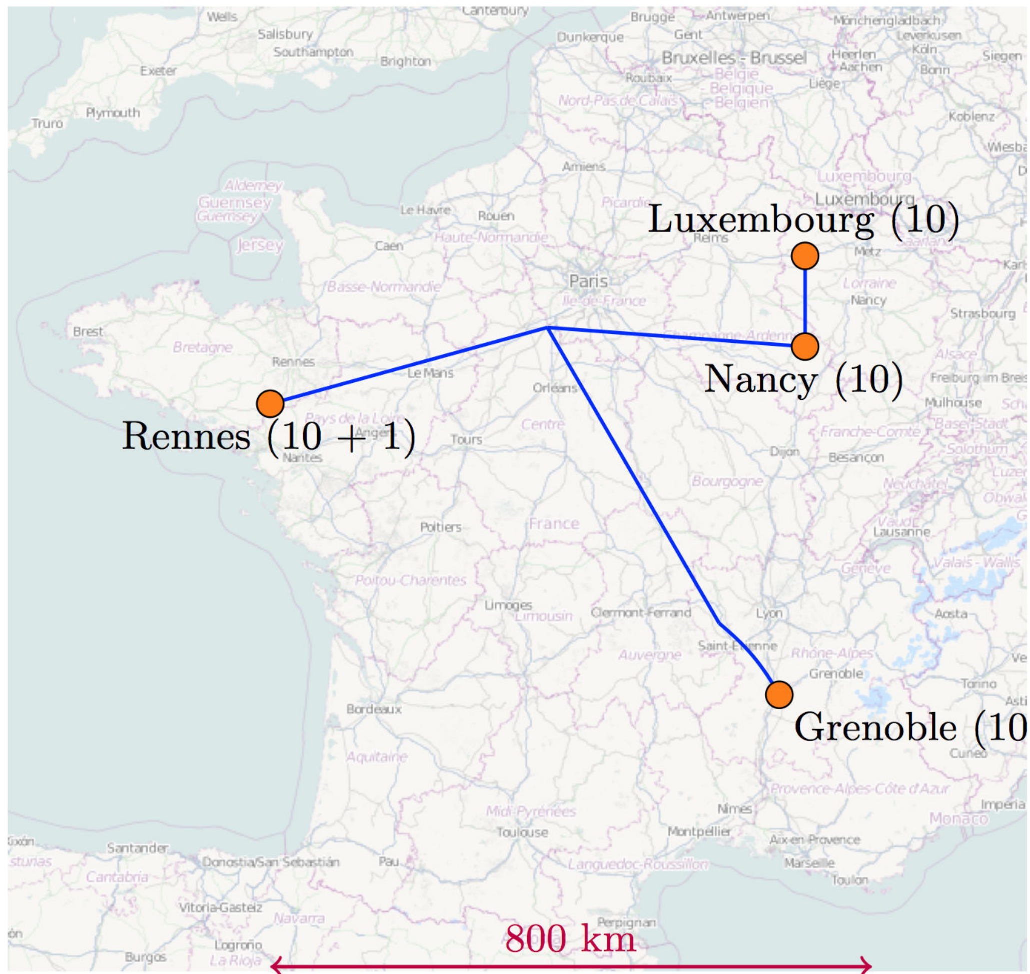

FIGURE 7.4 : Déploiement Grid'5000 utilisé pour les expériences.

2. Regroupement dynamique des nœuds

   Au cours de l'étude des résultats, nous avons remarqué qu'une part importante des migrations intersites étaient faites entre le site *Luxembourg* et le site *Nancy*. La table 7.2 indique clairement que ces deux sites ont une latence qui est significativement inférieure aux latences intersites habituellement mesurées, ce qui est expliqué par le fait Nancy et Luxembourg sont deux villes séparées par une distance de 100 kilomètres. Alors que les sites de *Rennes* et *Grenoble* ont de fortes latences avec tous les autres sites, et donc ne peuvent privilégier de site pour les collaborations externes, ce n'est pas le cas de *Luxembourg* et *Nancy* qui devraient se privilégier mutuellement quand ils ont besoin de collaborer avec des ressources externes, en particulier quand il s'agit de migrer une machine virtuelle sur un autre site. Cette observation montre que le LBO a permis à DVMS d'apprendre quel site privilégier, en promouvant les collaborations entre nœuds partageant de faibles latences réseau, ce qui a rendu beaucoup de migrations intersites acceptables en comparaison de celles faites avec la version utilisant Chord.

TABLE 7.1 : Comparaison des taux de migrations intrasite (DVMS/Chord vs DVMS/LBO)

	Chord	LBO
Moyenne	0.496	0.863
Minimum	0.378	0.798
Maximum	0.629	0.935

TABLE 7.2 : Latences réseau mesurées entre les sites Grid'5000 utilisés pour les expériences

	Grenoble	Luxembourg	Nancy	Rennes
Grenoble	0.09 ms	16.55 ms	14.24 ms	15.92 ms
Luxembourg		0.17 ms	2.70 ms	13.82 ms
Nancy			0.27 ms	11.42 ms
Rennes				0.23 ms

3. Réactivité

   Ici nous nous intéressons au temps de réaction de DVMS lors des différentes expériences. La table 7.3 contient des données qui permettent de faire une comparaison plus avancée du fonctionnement de l'usage des deux réseaux d'overlay par DVMS. En plus de réduire le nombre de migrations intersites, l'utilisation du LBO a permis de réduire la latence entre les nœuds d'une même partition, et donc d'accroitre la vitesse des communications, ce qui pour effet secondaire de réduire le temps moyen passé pour trouver un plan de reconfiguration, qui est 46% plus rapide que le temps moyen mesuré en utilisant Chord. Ce résultat est conforme avec le LBO permet de faire baisser le nombre moyen de sites impliqués dans une partition de 36%, à une valeur très proche d'un site unique. Ainsi la promotion des collaborations entre nœuds proches permet d'accélérer l'échange d'information et donc d'améliorer la réactivité de DVMS.

TABLE 7.3 : Comparaison de données sur les partitions ISP (DVMS/Chord vs DVMS/LBO)

	Chord	LBO
nombre moyen de sites impliqués	1.645	1.082
durée moyenne pour trouver un plan valide	154.63	98.50

7.4 Résumé

Dans ce chapitre nous avons présenté les travaux autour de la mise en place d'une solution logicielle permettant de résoudre les problèmes de localité réseau dans l'ordonnancement de machines virtuelles. En développant un réseau d'overlay non structuré reposant sur le système de coordonnées décentralisé Vivaldi [81], nous avons pu permettre à DVMS de

pouvoir prendre en compte la proximité en matière de réseau pour trouver des nœuds avec lesquels collaborer.

Nous avons décidé d'appliquer ce réseau d'overlay sensible à la localité réseau (*Locality Based Overlay network* - LBO) sur un algorithme d'ordonnancement de machines virtuelles, afin de le rendre sensible aux propriétés de localité réseau. Nous avons choisi DVMS [7], un algorithme d'ordonnancement fonctionnant de manière pair-à-pair au-dessus d'un réseau d'overlay Chord [73]. Nous avons donc remplacé ce dernier par notre LBO.

Enfin, nous avons conduit des expériences sur Grid'5000 pour estimer le gain apporté par la prise en compte des propriétés de localité réseau. Les résultats d'expériences ont montré que le LBO aboutissait à une baisse sensible des migrations intersites de machines virtuelles, très longues, au profit des migrations intrasites, plus courtes, sans nécessiter de changement dans la logique de l'algorithme DVMS. Cela a permis de mettre en avant l'intérêt de la prise en compte des propriétés de localité réseau dans l'organisation des collaborations au sein d'algorithmes distribués qui sont déployés dans un cadre multsite. Les appliquer aux services d'un gestionnaire *IaaS*, tel qu'OpenStack, permettrait de viser un fonctionnement dans un contexte multsite.

Contributions à VMPlaceS : un simulateur pour algorithmes de placement de machines virtuelles

Dans le chapitre 7, nous avions étudié l'ajout de la prise en compte des propriétés de localité réseau au sein d'un algorithme de placement de machines virtuelles. Les politiques de placement avancées sont évaluées soit via des expériences *in vivo* à échelle réduite, comme pour celles présentées dans la section 7.3, soit au moyen de techniques de simulation. Ces méthodes de simulations ne sont pas satisfaisantes, car elles ne permettent pas de modéliser précisément les plates-formes de Cloud Computing existantes (taille, représentation de la charge de travail, . . .), car elles ne permettent pas de comparer équitablement et facilement les politiques entre elles. Pour résoudre ces problèmes, VMPlaceS a été développé afin de proposer un logiciel dédié à la simulation d'algorithmes de placement de machines virtuelles, mettant particulièrement l'accent sur la collecte de métriques précises et sur l'assurance que la comparaison entre les algorithmes de placement est équitable. VMPlaces a été construit au-dessus de Simgrid [82], et est proposé comme un cadriciel (framework) qui offre des interfaces de programmation pour faciliter la conception d'algorithmes de placement, la conception d'injecteurs de charges de travail, et l'analyse des traces d'exécution.

Dans ce chapitre nous commencerons par donner un aperçu de l'architecture de VMPlaces, puis nous étudierons trois exemples d'algorithmes de placement différents implémentés dans VMPlaceS, et enfin nous verrons comment VMPlaces permet de les comparer de manière équitable.

8.1 VMPlaceS, un simulateur s'appuyant sur SimGrid

VMPlaceS est un outil de simulation pensé pour permettre d'étudier de manière poussée le fonctionnement des algorithmes de placement de machines virtuelles et pour permettre de faire des comparaisons équitables entre les algorithmes de placement. Pour permettre de faire face aussi bien à l'accroissement de la taille des centres de données qu'à la dynamicité des charges de travail et à l'élasticité caractérisant le paradigme de Cloud Computing, VMPlaceS permet aux concepteurs d'algorithmes de placement d'étudier le comportement de leurs travaux sur des scénarios simulant des infrastructures à large échelle contenant des centaines de milliers de machines virtuelles, soumises à une charge de travail variable au cours de la simulation.

VMPlaceS a été construit au-dessus de SimGrid [82], une boîte à outils pour simuler des algorithmes destinés à être exécutés sur des systèmes distribués fonctionnant à large échelle. SimGrid a été choisi car (i) sa pertinence en termes de performance et de validité est établie [83], et (ii) son support de la simulation de machines virtuelles ainsi que l'ajout d'un modèle pour les migrations à chaud (live migrations) [84] le rendent très utile pour la simulation d'infrastructures de Cloud Computing. Ses utilisateurs peuvent contrôler l'exécution d'une machine virtuelle (création, exécution, pause, destruction, migration, ...) de la même manière que si elles existaient, et le modèle de migrations à chaud permet de déterminer avec précision la durée ainsi que le trafic réseau des migrations de machine virtuelles, ce qui est indispensable à VMPlaceS pour assurer la précision des simulations.

8.2 Fonctionnement de VMPlaceS

VMPlaceS a été écrit en langage Java en s'appuyant sur l'interface logicielle MSG fournie par SimGrid. Comme le montre la figure 8.1, une simulation exécutée par VMPlaceS comporte trois phases :

1. **Initialisation** : VMPlaceS crée des machines virtuelles qui sont abstraites sous forme d'objets Java. Ces machines virtuelles sont assignées à des hôtes physiques, eux aussi représentés sous la forme d'objets Java. Chacune des machines virtuelles est créée à partir d'une classe de machines virtuelles qui va spécifier ses attributs matériels ainsi que l'intensité de son usage en matière de mémoire vive. Les machines virtuelles démarrent avec une consommation CPU nulle, car l'injection de charge n'a pas encore commencé. Par défaut les évènements indiquant les modifications en termes charge CPU des VMs sont créés lors de la phase d'initialisation. Le tirage des valeurs de charge de travail obéit à des lois de probabilité qui sont identiques à celles décrites dans la section 7.3. Il est possible de surcharger le générateur d'évènements de charge afin de définir son propre comportement, ou encore de définir ses propres évènements tels que l'apparition et la disparition de nœuds (simulation de pannes).

2. **Injection de charge / Ordonnancement** : Une fois l'infrastructure définie et les évènements générés, la simulation peut commencer. En premier lieu, l'injecteur de

charge consomme de manière itérative les évènements aboutissant à une fluctuation de la charge des machines virtuelles. Sur la base des décisions de l'algorithme de placement testé lors de la simulation, les machines virtuelles sont mises en pause, redémarrée ou migrée sur d'autres hôtes physiques. Les décisions prises par les algorithmes de placement reposent sur des calculs réels exécutés par la machine responsable de la simulation. Cela permet d'avoir des décisions conformes à celles qui auraient été prises dans des conditions réelles.

3. **Analyse des traces d'exécution** : Pendant la simulation, de nombreuses informations sont collectées sur l'état de l'infrastructure simulée ainsi que sur la prise de décisions des algorithmes de placement. L'analyse qui en est faite se décompose en deux phases. Tout d'abord, les données qui ont été collectées sont croisées afin de produire des données à plus forte valeur ajoutée, telles que la pertinence des décisions prises par l'algorithme de placement ou la comparaison entre algorithmes de placement. Enfin, à partir de l'ensemble des données à la fois collectées et construites, des graphiques ainsi que des tableaux sont générés sans nécessiter d'intervention humaine.

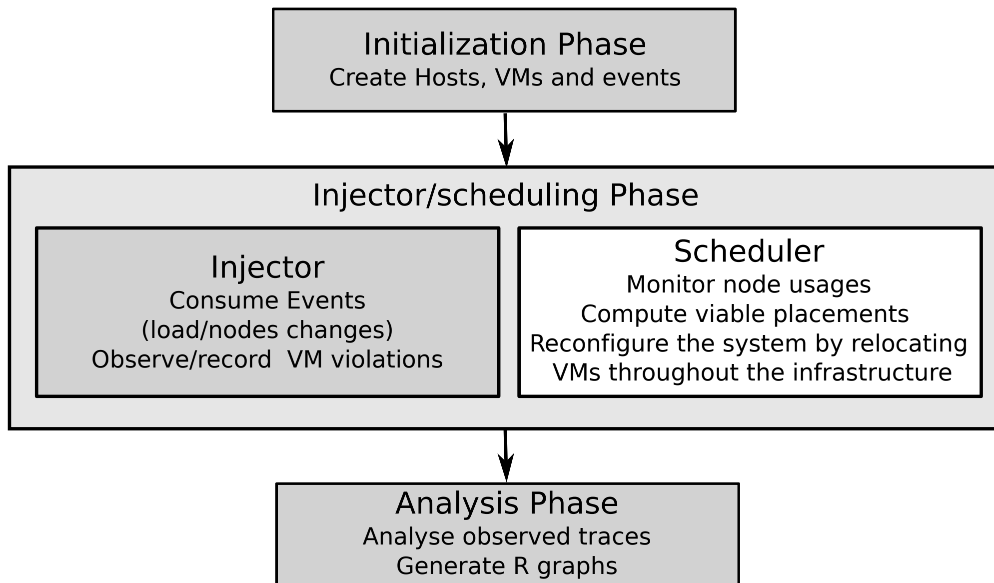

FIGURE 8.1 : Illustration du fonctionnement de VMPlaceS.

8.3 Validation expérimentale de VMPlaceS

La validation de VMPlaceS s'est faite selon deux axes : en premier lieu la précision des simulations, puis la facilitation du processus d'évaluation et de comparaison des algorithmes de placement, en prenant le cas de trois algorithmes.

8.3.1 Précision des simulations

Pour valider la précision de VMPlaceS, nous avons implémenté un outil qui pilote une infrastructure déployée sur Grid'5000, tout en exposant les mêmes interfaces logicielles que VMPlaceS. Cet outil intègre un algorithme de placement centralisé s'appuyant sur Entropy [22]. L'algorithme d'Entropy comprend deux phases, la première correspondant au calcul d'un plan de reconfiguration et la seconde consiste en l'application du plan au moyen de migrations à chaud. Afin de pouvoir comparer le comportement que cet algorithme a lors d'une simulation avec celui qu'il adopte lors d'une expérience réelle (*in vivo*), nous avons déployé une infrastructure de 32 machines physiques sur Grid'5000, chacune hébergeant initialement 6 machines virtuelles. Une infrastructure identique a été recréée dans une simulation exécutée par VMPlaceS, notamment en ce qui concerne les capacités réseau des serveurs utilisés pour l'expérience. La durée de la simulation a été fixée à 3600 secondes. Initialement les machines virtuelles ont toutes une charge de travail représentant 0% des capacités de calcul d'un cœur CPU. De la charge de travail est ensuite envoyée aux machines virtuelles. Dans le cas des simulations, la charge est simulée en modifiant des variables des objets représentant l'infrastructure, tandis que pour les exécutions *in vivo* la charge de travail est simulée grâce à un programme **memtouch** [84] qui charge le CPU et la mémoire RAM. Au cours de l'expérience, l'injection de charge de travail dans les machines virtuelles se faisait en suivant un processus aléatoire reproductible (car initialisée avec la même graine aléatoire). Pour chaque évènement de modification de la charge d'une machine virtuelle, la machine virtuelle concernée était choisie selon une loi exponentielle de paramètre λ=nombre_de_VM/300 et la nouvelle valeur était tirée d'après une loi normale de paramètres μ=60 et σ=20.

La figure 8.2 montre les temps passés à réaliser les deux phases de l'algorithme Entropy à chaque fois qu'il a été invoqué, lors d'une simulation et d'une exécution sur Grid'5000. À chaque invocation, Entropy avait pour charge d'analyser le placement des machines virtuelles, de détecter les hôtes physiques surchargés et de proposer des migrations de machines virtuelles pour équilibrer la charge de travail. Durant les 500 premières secondes, la charge de travail des machines virtuelles augmente progressivement. On n'observe pas d'invocations à Entropy. À partir de 540 secondes, Entropy commence à détecter des hôtes surchargés et se met en quête de reconfigurations permettant d'équilibrer la charge de travail. Une comparaison des deux graphiques présentés dans la figure 8.2 montre que les temps de calcul effectués par la simulation et l'expérience *in vivo* sont très proches. En ce qui concerne le temps d'application des plans de reconfiguration, on a mesuré que ceux de la simulation et de l'expérience *in vivo* diffèrent d'un niveau variant entre 6% et 18%. Ces différences s'expliquent par notre observation que le fait de faire des migrations parallèles vers le même hôte physique aboutissait à une fluctuation du temps de migration. Bien que ce phénomène ne soit pas intégré dans le modèle de migration à chaud de SimGrid, ces résultats montrent que VMPlaceS est suffisamment précis pour permettre d'étudier les performances d'un algorithme de placement ainsi que de son comportement général.

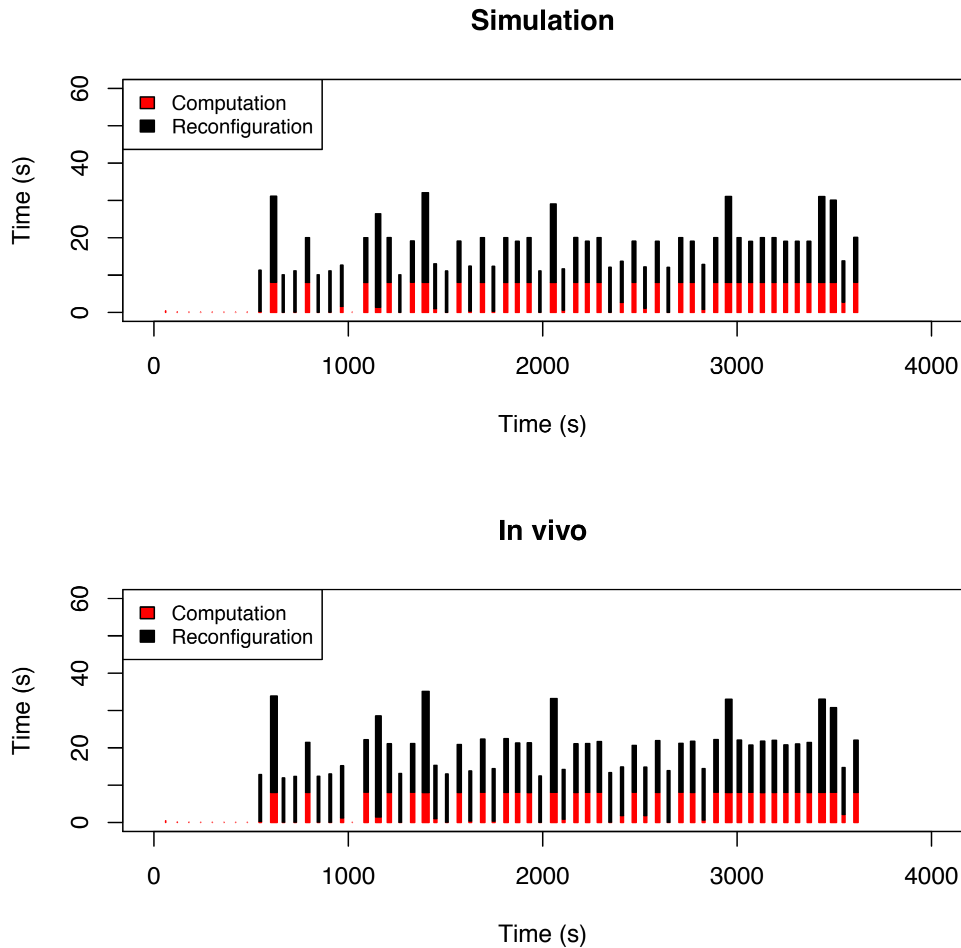

FIGURE 8.2 : Comparaison des exécutions simulées (haut) et *in vivo* (bas).

8.3.2 Comparaison de trois algorithmes de placement

Pour démontrer la pertinence de VMPlaceS à permettre la comparaison équitable d'algorithmes de placement de machines virtuelles, nous avons décidé de l'utiliser pour évaluer trois algorithmes, au moyen de simulations réalisées avec VMPlaceS :

- **Algoritme centralisé utilisant Entropy** : Cette approche est identique à celle utilisée dans la section 8.3. Un processus centralisé Entropy vérifie en continu pour détecter les hôtes surchargés. La surveillance de la consommation des ressources se fait en accédant aux états des machines virtuelles et hôtes physiques, exposés par l'API de VMPlaceS.

- **Algorithme hiérarchique à la Snooze** : Snooze [71] exploite une approche hiérarchique où un *group leader* (GL) centralise les informations propres à un cluster entier en se servant de données synthétiques fournies par les *group managers* (GMs)

qui représentent une étape hiérarchique intermédiaire en charge d'exploiter un certain nombre de *local controllers* (LCs) hébergeant des machines virtuelles. Pendant l'exécution, les composants haut-niveau envoient des signaux aux composants plus bas niveau, afin de vérifier que ceux fonctionnent, et pour récupérer leurs charges de travail. En cas de surcharge d'un des LCs, c'est le GM qui en est responsable qui s'occupe de trouver un nouveau placement de machines virtuelles. Dans notre implémentation de Snooze, le placement est calculé au moyen d'Entropy.

- **Algorithme pair-à-pair reposant sur DVMS** : Cette approche utilise l'algorithme DVMS historique qui a été présenté dans la section 7.1.2. Lors de la préparation de la simulation, un processus SimGrid hébergeant un agent DVMS est associé à chaque hôte virtuel. Ces agents DMVS surveillent en continu l'usage des ressources des machines virtuelles hébergées sur leurs hôtes respectifs. Quand un nœud est surchargé, son agent DVMS collabore avec les autres agents pour équilibrer la charge de travail. DMVS utilise lui aussi Entropy pour trouver un plan de reconfiguration.

Les simulations ont été faites avec des infrastructures définies comme ayant un nombre variable d'hôtes physiques variant entre 128 et 1024 nœuds, chacun ayant 8 cœurs CPU, 32 GB de RAM et une liaison réseau de 1Gbps. Dix machines virtuelles ont été lancées sur chacun des nœuds avec une charge de travail initiale nulle. Au cours de la simulation, les machines virtuelles sont progressivement soumises à une charge de travail injectée grâce au mécanisme décrit dans la section 8.3.1. La durée de simulation a été fixée à 1800 secondes et la charge a atteint un état stationnaire de 85% des capacités CPU comme le montre la figure 8.3.

La figure 8.4 présente, pour chacun des algorithmes et chacune des tailles d'infrastructures, le cumul des temps durant lesquels les hôtes physiques ne pouvaient satisfaire complètement les demandes en ressources des machines physiques qu'ils hébergeaient. Il est remarquable qu'à partir d'une taille de 512 nœuds l'efficacité d'Entropy baisse considérablement, ses performances devenant proches de ce qui aurait été obtenu sans faire de reconfiguration. Ce résultat souligne l'intérêt de faire des simulations avec VMPlaceS, car vérifier ce résultat expérimentalement en utilisant 1024 serveurs physiques sur une plateforme telle que Grid'5000 serait très coûteux, VMPlaceS permet de mettre en évidence le manque de passage à l'échelle de la solution centralisée sans avoir à mettre en place d'infrastructures surdimensionnées. Enfin, cette expérience permet de comparer Snooze et DVMS. On constate que DVMS fait sensiblement mieux que Snooze, ce qui s'explique par le fait que dans Snooze seuls les GMs calculent des plans de reconfiguration, tandis que DVMS répartit le travail sur l'ensemble des nœuds, tout en travaillant sur des ensembles de nœuds plus petits en moyenne, ce qui aboutit à une meilleure réactivité.

Enfin, la figure 8.5 illustre la capacité de VMPlaceS à collecter des détails précis sur les évènements qui surviennent au cours de la simulation. La figure 8.5 illustre chacune des surcharges d'hôtes physiques qui ont eu lieu pour l'expérience reposant sur Snooze avec 1024 nœuds. En plus du temps d'apparition et de la durée de chacune des surcharges, on peut aussi voir qu'elles peuvent être de 4 types différents :

- **Surcharge détectée** : surcharge d'un hôte physique causée par les fluctuations des

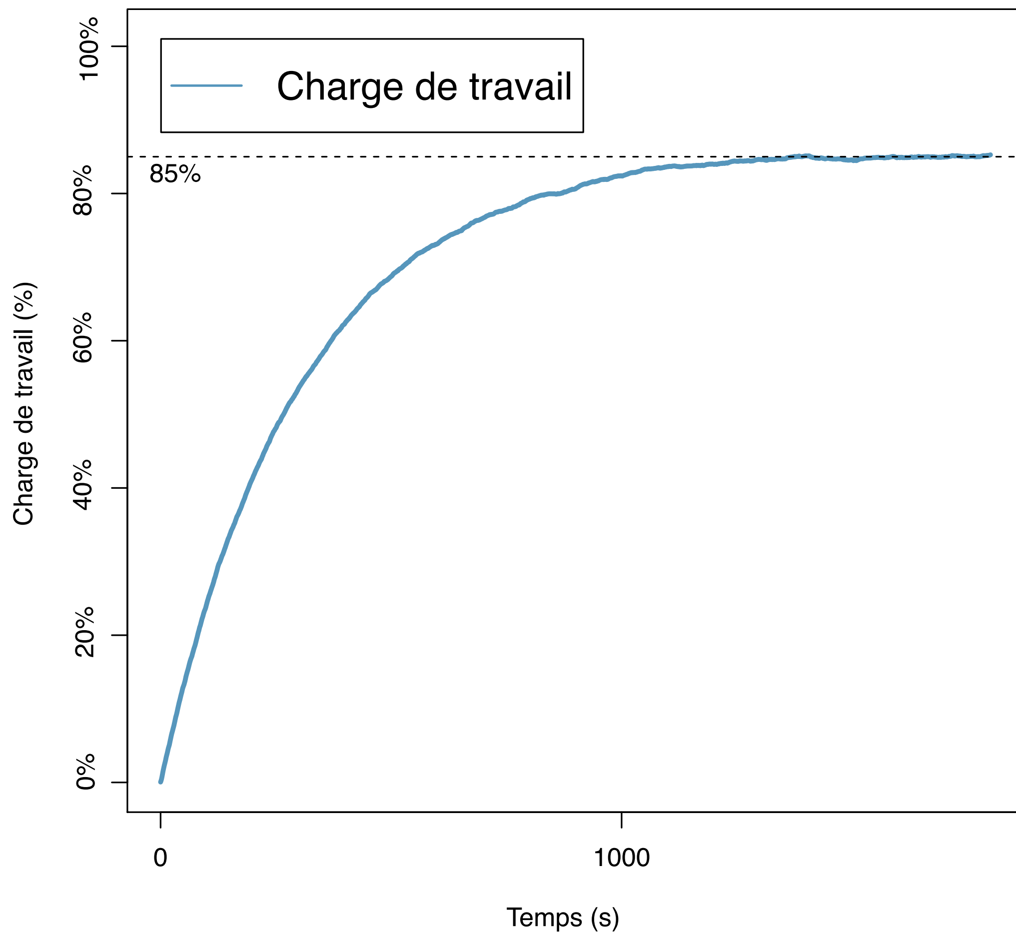

FIGURE 8.3 : Évolution de la charge de travail imposée à l'infrastructure.

charges de travail de ses machines virtuelles. L'algorithme de placement parvient à les détecter.

- **Surcharge cachée** : surcharge d'un hôte physique causée par les fluctuations des charges de travail de ses machines virtuelles. L'algorithme de placement ne parvient pas à les détecter. Dans la plupart des cas, c'est dû à un manque de réactivité.

- **Surcharge induite détectée** : surcharge d'un hôte physique causée par une migration de machine virtuelle dont la charge de travail s'ajoute à celle des autres machines de l'hôte. L'algorithme de placement parvient à les détecter.

- **Surcharge induite cachée** : surcharge d'un hôte physique causée par les fluctuations des charges de travail de ses machines virtuelles. L'algorithme de placement ne parvient pas à les détecter à cause, là aussi, d'un manque de réactivité.

Ces informations ont un intérêt pour comprendre le fonctionnement des algorithmes de placement de machines virtuelles, en particulier lors de la détection des comportements erratiques. Ce graphe a été généré à partir des données concernant les évènements de

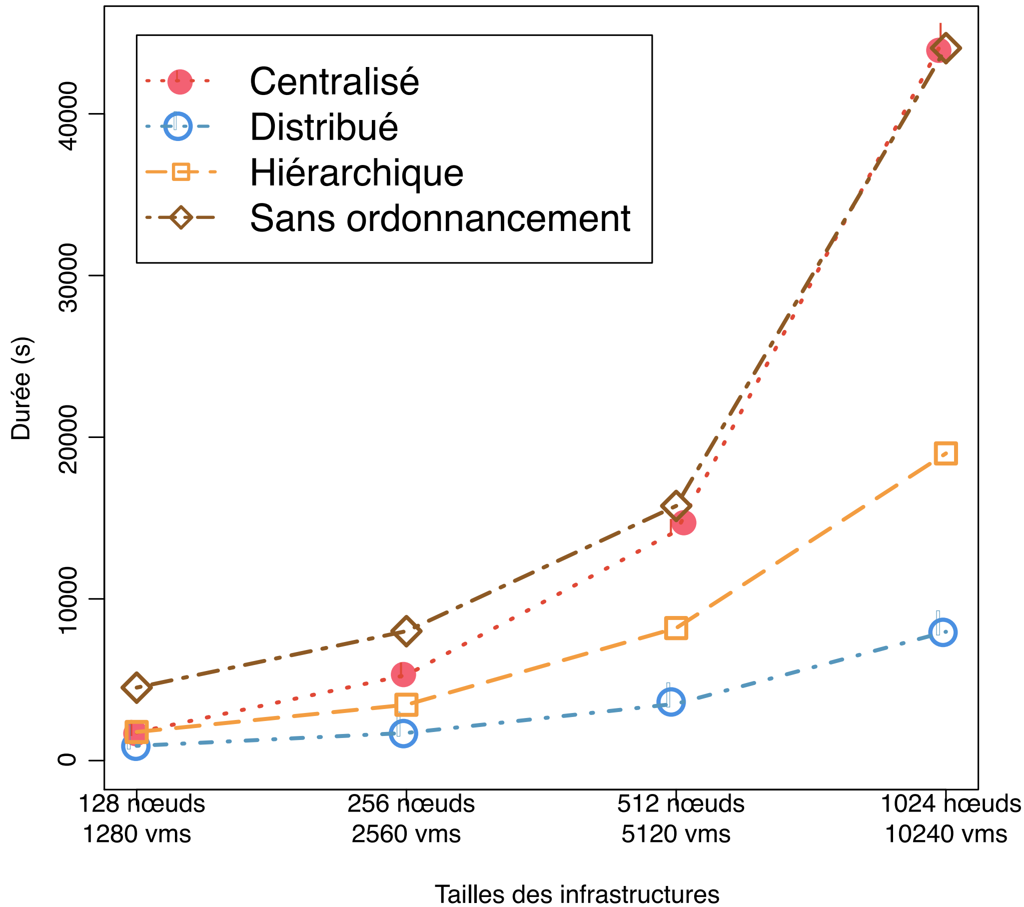

FIGURE 8.4 : Cumul des temps de surcharge des hôtes physiques, en fonction des algorithmes.

surcharge des hôtes, il existe de nombreux autres évènements (changement de charge des VMs, migrations de VMs, pannes d'hôtes, ...) pouvant être utilisés pour comprendre des aspects autres que les surcharges d'hôtes pendant les simulations.

8.4 Travaux connexes

L'ordonnancement de travaux/tâches dans les systèmes distribués est un vieux problème et donc plusieurs simulateurs ont été proposés au cours des dernières années, afin d'évaluer les avantages et les inconvénients des nouvelles propositions en termes de stratégies d'ordonnancement. Dans un exemple récent, Google a rendu accessible au public le simulateur [1] utilisé par le cadriciel Omega [85]. Cependant, les simulateurs testant l'ordonnancement de travaux et de tâches ne prennent pas en compte la notion de machine virtuelle et de ses capacités (pause/reprise, migration) et donc ne sont pas appropriés pour évaluer les stratégies de mise en place d'infrastructures de production de *Cloud Computing*.

[1] https://github.com/google/cluster-scheduler-simulator

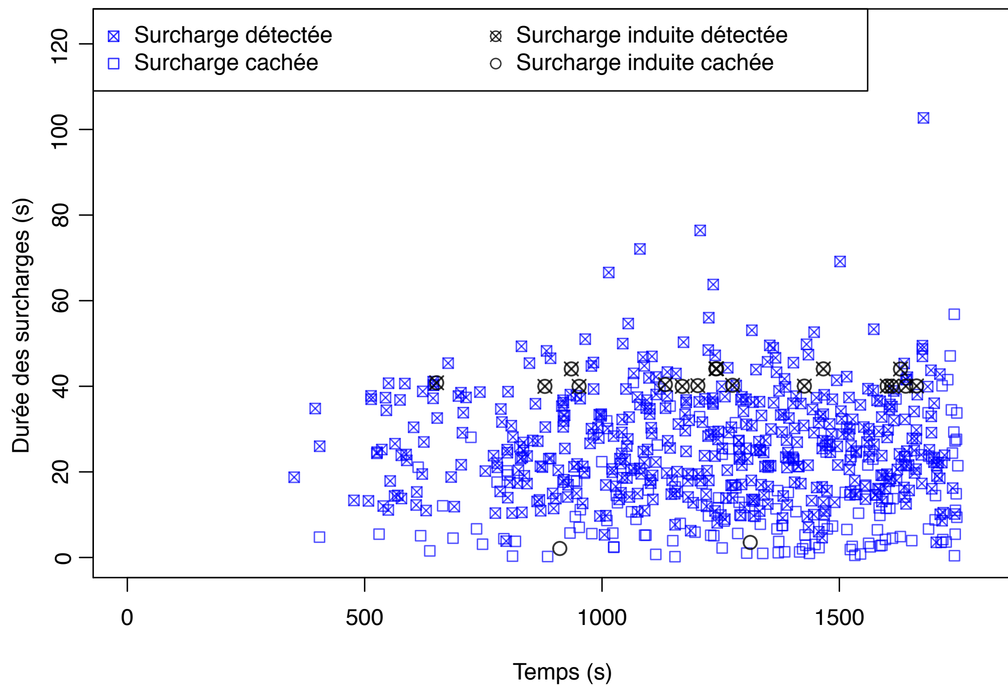

FIGURE 8.5 : Illustration du détail des informations récoltées sur les surcharges des hôtes physiques (Hiérarchique déployé sur 1024 nœuds).

Les outils de simulations qui ont été proposés pour la simulation d'infrastructures de Cloud Computing [86, 87, 88, 89, 90] peuvent être classifiés en deux catégories, la première correspondant aux simulateurs "ad-hoc" qui ont été développés pour s'attaquer à un problème donné. Par exemple CReST [87] est un outil de simulation permettant d'évaluer les algorithmes de provisionnement de ressources de Cloud Computing. Si les simulateurs "ad-hoc" permettent de dégager les tendances comportementales des systèmes de Cloud Computing, ils ne prennent pas en compte l'incidence des couches entre elles, qui peut conduire à des résultats peu représentatifs. De plus, de telles solutions ad-hoc sont développées pour des cas d'utilisations très spécialisés et ne sont donc pas accessibles à la communauté scientifique dans son ensemble. La seconde catégorie [86, 89, 90] correspond à des outils plus génériques pour la simulation d'infrastructures de Cloud Computing, permettant de s'attaquer de manière plus large aux problèmes toujours ouverts du modèle de Cloud Computing. Cependant, ces projets se sont principalement concentrés sur la définition d'*API* et non sur la modélisation des différents mécanismes que l'on retrouve dans les infrastructures de Cloud Computing.

Par exemple, CloudSim [86], qui a été largement utilisé dans des publications scientifiques pour pour valider des algorithmes ainsi que des applications, repose plutôt sur une approche descendante (*top-down*) des environnements de Cloud Computing. C'est ainsi qu'il n'y a aucun pas de travaux scientifiques qui reproduisent fidèlement les différents modèles de ces environnements. Ainsi le temps de migration d'une machine virtuelle est calculé en divisant sa quantité de mémoire vive par la bande passante réseau. En plus des imprécisions liées à ces couches basses, les outils de simulations utilisent souvent des

modèles de virtualisation qui sont trop simplifiés, qui conduisent au final à des résultats de simulation qui sont non représentatifse. Comme le souligne ce chapitre, VMPlaceS a été construit au-dessus de Simgrid afin de produire un outil générique qui puisse bénéficier de sa précision en ce qui concerne la modélisation des techniques de virtualisation [91].

8.5 Résumé

Ce chapitre a présenté les travaux autour de VMPlaceS, une boîte à outils construite sur SimGrid et proposant des mécanismes génériques pour concevoir des algorithmes de placement de machines virtuelles, des mécanismes pour pouvoir les tester dans des simulations à large échelle, ainsi que des moyens d'analyses des traces d'exécution permettant la compréhension de leur fonctionnement. La précision des simulations a été validée en comparant expérimentalement les résultats d'une simulation d'infrastructure de Cloud Computing avec les résultats mesurés avec un déploiement Grid'5000 équivalent. Nous avons aussi illustré la pertinence de VMPlaceS en l'utilisant pour implémenter et comparer des algorithmes de placement représentatif de trois différentes classes de fonctionnement (centralisé, hiérarchique et complètement distribué). Les résultats présentés dans la section 8.3.2 de ce chapitre représentent la première comparaison équitable de ces trois algorithmes de placement, sur des simulations d'environnements comptant jusqu'à 1000 hôtes physiques et 10000 machines virtuelles. Enfin ces travaux sont toujours actifs et une soumission journal est prévue où nous montrons notamment comment le modèle énergétique présent dans SimGrid peut être exploité pour étudier les performances énergétiques des différentes stratégies de placement.

Conclusions et perspectives

9

Rappel des contributions

La principale contribution de cette thèse a été d'étudier comment le gestionnaire OpenStack multisite pouvait exploiter une infrastructure multisite tout en garantissant un passage à l'échelle. Nous nous sommes attaqués à une des limitations actuelles en remplaçant la base de données relationnelle centralisée par une base de données non relationnelle de type clé/valeur. Afin d'éviter de modifier de manière importante le code source existant, et donc éviter de casser la compatibilité avec les autres mécanismes d'OpenStack, nous avons développé *Rome*, un outil de mapping objet-relationnel (ORM) qui reprend les interfaces logicielles de l'ORM utilisé par OpenStack (*SQLAlchemy*). De cette manière, nous avons réussi à faire fonctionner le service Nova avec une base de données non relationnelle de type clé/valeur. Cette approche a été validée expérimentalement sur la plateforme Grid'5000 où des expériences monosites ont montré que notre approche améliorait la réactivité moyenne des appels à l'API de Nova pour 80% des requêtes. Ensuite, des expériences multisites nous ont permis de démontrer que l'utilisation d'une base de données de type clé/valeur permettait un fonctionnement dans une configuration multisite. Ces résultats ont notamment été présentés lors de l'*OpenStack Summit 2016* à Austin, et ont permis d'engager un dialogue avec la communauté des contributeurs du service Nova. De plus, ces travaux ont facilité la mise en place d'une ADT *Mercury* dans le cadre du projet *Discovery*, et une collaboration est en cours avec Orange afin d'appliquer cette stratégie au service Neutron.

Une analyse poussée des résultats précédents a mis en avant que malgré l'utilisation de mécanismes de ségrégation tels que les *host-aggregates* et les *availability zones*, OpenStack ne cherchait pas à favoriser les collaborations entre les nœuds situés sur un même site, et donc limiter le trafic réseau intersite. La deuxième contribution de cette thèse a consisté à étudier le potentiel de la prise en compte de la localité réseau pour améliorer le fonctionnement de mécanismes d'infrastructures de Cloud Computing existants. Nous nous sommes intéressés à DVMS, un algorithme de placement de machines virtuelles fonctionnant de manière distribuée. Historiquement, ce mécanisme utilisait une topologie

en anneau qui ne permettait pas de favoriser les collaborations entre des nœuds proches. Nous avons remplacé la topologie en anneau par une topologie reposant sur l'algorithme Vivaldi pour identifier les nœuds proches géographiquement en mesurant leurs latences réseau respectives. Des expériences sur Grid'5000 ont confirmé le potentiel de cette approche pour améliorer la qualité des décisions des algorithmes de placement de machines virtuelles. Ainsi la proportion de migrations intrasites est passée de 49.6% à 86.3%, les migrations intersites se faisant entre des sites géographiquement proches ayant une latence réseau acceptable.

Enfin, les travaux sur DVMS nous ont conduits à participer au développement de VMPlaceS, un simulateur permettant la comparaison équitable d'algorithmes de placement de machines virtuelles, ainsi que l'analyse de leurs traces d'exécution afin de faciliter la détection de tout défaut de conception. Nous avons développé trois différents algorithmes de placement de machines virtuelles avec ce simulateur : un algorithme centralisé utilisant Entropy, un algorithme hiérarchique reprenant Snooze et un algorithme distribué s'appuyant sur DVMS. Nous avons démontré la précision des simulations en comparant les résultats d'une simulation du fonctionnement de l'algorithme centralisé avec son équivalent déployé sur Grid'5000. Enfin, grâce à VMPlaceS nous avons pu réaliser la première étude comparative de ces trois algorithmes, en les mettant dans des conditions d'exploitation d'infrastructures dont la taille est montée jusqu'à 1024 nœuds et 10240 machines virtuelles. Le code source de VMPlaceS est disponible librement et le projet est actif car un algorithme supplémentaire est mis en oeuvre dans le cadre d'un postdoctorat en cours au sein de l'équipe ASCOLA, et le simulateur est utilisé par des membres de l'IRT B-COM dans le cadre de l'outil Watcher de la communauté OpenStack.

10

Perspectives

Après avoir rappelé les contributions faites pendant ces travaux de doctorat, nous donnons dans ce dernier chapitre plusieurs perspectives possibles. Nous commençons par exposer des perspectives à court terme qui mêlent des actions d'ingénierie qui visent certains points des contributions et d'autres actions plus ouvertes qui pourraient accroître la valeur scientifique de ces travaux. Enfin nous terminons sur des perspectives à long terme qui pourraient s'inscrire dans la continuité des travaux réalisés lors de ce doctorat.

10.1 Perspectives à court terme

10.1.1 Rome

Amélioration de l'implémentation de Rome

Dans le chapitre 6, nous avons montré qu'il était possible de modifier le service Nova pour qu'il supporte les bases de données non relationnelles. Cette modification a permis de le faire fonctionner dans un cadre multisite (plus de 8 sites) de manière complètement décentralisée avec des performances meilleures que l'approche existante utilisant l'extension Galera. Dans une utilisation monosite, notre solution était globalement plus rapide pour 80% des requêtes (figure 6.12) traitées lors de nos expériences. La principale piste est d'accroître la qualité de l'implémentation de la bibliothèque *Rome* afin qu'elle se rapproche des standards industriels. En plus d'améliorer l'implémentation, ce qui permettrait d'envisager une meilleure réactivité en s'attaquant aux 20% de requêtes API qui sont plus lentes en utilisant *Rome*. Enfin, cela permettrait de rendre plus facile la reprise de l'implémentation par des personnes désireuses de poursuivre les travaux autour de ce projet. Dans le cadre de l'ADT *Mercury*, un ingénieur de recherche sera en charge de stabiliser le code de *Rome* d'ici mai 2017.

Vers une validation plus poussée

La section 6.4.2 a présenté des expériences qui se sont limitées à 8 sites géographiques, seuil à partir duquel seul *Rome* fonctionnait correctement alors que les solutions reposant sur SQLAlchemy (serveur MySQL unique ainsi que réplication avec Galera) subissaient de grandes difficultés. Ces expériences montrent que notre approche va dans la bonne direction pour mettre en place une infrastructure OpenStack massivement distribuée qui pourrait être déployée en périphérie du réseau Internet. Multiplier le nombre de sites géographiques permettrait de confirmer la validité de notre approche.

De plus, ces expériences se sont limitées à la création de 500 machines virtuelles en parallèle, ce qui permet d'étudier le comportement en cas de charge de travail très intensive, mais qui n'est pas représentatif de la charge de travail d'une infrastructure de Cloud Computing de taille moyenne. Nous pourrions ainsi tester notre approche avec une charge de travail plus petite, mais continue sur une durée plus longue, ou tester la tolérance aux pannes de notre approche en simulant des pertes de réseau entre des sites.

Extension aux autres services d'OpenStack

Bien que *Rome* ait été validée avec le service *Nova*, l'étendre aux autres services d'OpenStack permettrait d'envisager la mise en place d'une infrastructure où chacun des mécanismes pourrait être massivement distribué. Un premier prototype du service Glance fonctionnant sur *Rome+REDIS* a été développé, mais est dans une phase précoce de validation sur Grid'5000. En parallèle, nous avons commencé à étudier l'application de cette même stratégie au service Neutron. Le fait que celui-ci utilise des pilotes réseau (*networking drivers*) difficilement distribuables et le fait que l'organisation de ses accès en base de données diffère de Nova, pourrait rendre plus difficile l'intégration de Neutron avec *Rome*. *Orange labs* a débuté des travaux qui vont dans cette direction.

Prise en compte de la localité réseau dans OpenStack

Dans la section 6.4.4 nous avions vu que malgré l'utilisation de mécanismes de ségrégation tels que les *host-aggregates*, la quantité de communications intersites demeurait très importante (87.7%). Les trois quarts de ces communications étaient dus à *Nova* car ses agents communiquaient par échange de message sur un bus partagé et partageaient leurs états grâce à une base de données partagée. Une piste pour améliorer cette situation serait d'utiliser des mécanismes de ségrégation plus forts, tels que les cells-v2. La partie supérieure des cells-v2 pourrait être distribuée en utilisant *Rome* pour stocker ses états dans un système clé/valeur, tandis que les parties basses des cells-v2 auraient leurs propres base de données et système de bus de messagerie, ce qui isolerait une grande partie du trafic réseau à l'intérieur des cells-v2.

Vers une implémentation utilisant les bases de données NewSQL ?

Rome a permis de traduire les appels vers l'API de SQLAlchemy (opérations relationnelles exposées avec une API objet) en actions sur une base de données non relationnelle

de type clé/valeur. Le paradigme des bases de données *NewSQL* [92, 93, 94] a commencé à émerger au cours de la dernière décennie, permettant de combiner les possibilités des bases de données relationnelles avec le passage à l'échelle des bases de données non relationnelles. Des implémentations libres de bases de données *NewSQL* utilisant l'API de MySQL [95, 96] sont récemment apparues, permettant d'envisager leur utilisation dans les services OpenStack. Ainsi, en choisissant une telle base, nous pourrions déléguer une grande partie du travail de *Rome* (traduction des appels à l'API de SQLAlchemy vers une base de données NoSQL) à la base *NewSQL*, en implémentant un pilote logiciel *SQLAlchemy* dédié aux bases *NewSQL*. Avant de se lancer dans une telle approche, il conviendrait naturellement de valider expérimentalement que les bases de données *NewSQL* sont adaptées à la charge de travail d'une infrastructure de Cloud Computing massivement distribuée.

10.1.2 VMPlaceS et DVMS

Augmentation de l'échelle des simulations

Les simulations présentées dans la section 8.3.2 ont porté sur des infrastructures comprenant jusqu'à 1024 nœuds et 10240 machines virtuelles, ce qui représente une infrastructure de Cloud Computing une taille moyenne. Pour atteindre des simulations d'infrastructures proches de celles déployées dans les mégas centres de données, nous nous donnons comme objectif à long terme d'améliorer le code existant pour atteindre des simulations impliquant 10000 hôtes physiques et 100000 machines virtuelles. Ce travail est dépendant des modifications réalisées au cœur de l'outil Simgrid. Bien qu'une collaboration soit en cours avec les principaux contributeurs de Simgrid, la refonte majeure de Simgrid version 4 pourrait imposer de remettre ces travaux à plus tard.

Complétion du modèle de ressources de VMPlaceS

Il serait intéressant de compléter le modèle de ressources de VMPlaceS pour que les ressources de calcul subissent les effets des fluctuations des entrées/sorties issues du réseau et du stockage. De plus, alors qu'actuellement les machines virtuelles sont créées statiquement, avoir la possibilité de provisionner ou détruire des machines virtuelles en cours de simulation permettrait de mettre en place des scénarios plus fidèles à ce qui se passe au sein des infrastructures de Cloud Computing. Enfin, l'introduction d'une API proche de celles des gestionnaires *IaaS* permettrait d'étendre le champ des simulations conduites par *VMPlaceS*. De tels travaux sont abordés en partie par un postdoctorant au sein de l'equipe ASCOLA et également par l'Université de Strasbourg qui travaille sur l'ajout d'une couche Cloud Computing au sein de Simgrid.

Amélioration du modèle de prise de décision de DVMS

Le système initial de prise décision de DVMS reposait uniquement sur la prise en compte des consommations en ressources CPU et RAM. Les travaux présentés dans le chapitre 7 ont permis la favorisation indirecte des migrations intrasites. Cependant la prise de

décision de DVMS n'intègre pas les métriques réseau telles que la latence ou la bande passante, ce qui pourrait être un problème dans les quelques cas où une migration inter-site devrait être privilégiée. Le moteur de prise de décision pourrait ainsi être amélioré en intégrant une plus grande variété de métriques. Ainsi, le remplacement d'Entropy par BtrPlace pourrait permettre à DVMS de supporter des extensions faites par ses utilisateurs, réglant ainsi le problème du manque de variété des métriques intégrées aux prises de décision. Ainsi, on pourrait par exemple prendre en compte le trafic réseau (via supervision) et l'intégrer dans le moteur de décision de DVMS.

10.2 Perspectives à long terme

Au cours de ces travaux de doctorat, nous avons étudié la question de la conception d'une infrastructure de *Cloud Computing* permettant l'exploitation d'un réseau de nanos centres de données et s'appuyant sur le projet OpenStack. Nous nous intéressons maintenant aux perspectives à long terme qui pourraient suivre ces travaux.

La première perspective à long terme serait que les efforts entrepris au cours de cette thèse soient poursuivis et complétés au sein de l'initiative *Discovery*, ce qui permettrait idéalement d'aboutir à un système complet permettant de fédérer la puissance de plusieurs petits centres de données déployés en périphérie du réseau Internet, permettant ainsi de proposer des ressources informatiques proches de leurs consommateurs. La majorité des centres universitaires qui raccordés au réseau *RENATER* hébergent déjà des ressources de calcul, il est donc raisonnable d'imaginer qu'ajouter quelques serveurs dans chacun de leurs points de présence *RENATER* et y installer une version aboutie du système défendu dans cette thèse permettrait de mettre en place une infrastructure de Cloud Computing universitaire distribuée à l'échelle nationale. Une telle infrastructure pourrait avoir des avantages indéniables, et pourrait être utilisée pour les activités d'enseignement et de recherche autour du Cloud Computing. En admettant que l'initiative *Discovery* réussisse à produire un système permettant d'associer des mécanismes tirant parti de la localité réseau à une version d'OpenStack épurée des éléments limitant son passage à l'échelle, nous pourrions aller plus loin et imaginer tirer parti de la fédération qui existe entre les réseaux universitaires européens pour étendre la vision précédente à une fédération native d'infrastructures de Cloud Computing européennes.

La seconde perspective à long terme serait celle qui accompagnera le phénomène d'Internet des objets (*Internet of Things* - IoT). Certains estiment que 50 milliards d'appareils (capteurs, téléphones, tablettes, ...) seront connectés à Internet à l'horizon 2020. Ces appareils consommeront des ressources informatiques, produiront de manière combinée de très grandes quantités de données et seront présents partout sur la planète (voire au-delà). Cette vision correspond au *Fog Computing* et appelle à la mise en place d'infrastructures déployées au plus proche de ces appareils. En faisant l'hypothèse qu'une telle vision se réalise, la mise en place d'une telle infrastructure pose naturellement certains défis qui ne sont pas encore complètement résolus, tels que le stockage massif des données produites par des milliards d'appareils, la mise en place de mécanismes de sécurité évitant qu'une intrusion dans un des éléments de l'infrastructure ne compromette le reste de l'infrastructure, ainsi que la mise en place de mécanismes de facturation et de partage de ressources

entre différents opérateurs locaux s'associant pour mettre en place une infrastructure à l'échelle mondiale.

References

Bibliographie

[1] Albert GREENBERG et al. "The cost of a cloud : research problems in data center networks". In : *ACM SIGCOMM computer communication review* 39.1 (2008), p. 68–73.

[2] AMAZON. *Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region*. 2012. URL : https://aws.amazon.com/message/680587/.

[3] IEEE 802.3 Ethernet Working GROUP. *IEEE 802.3TM Industry Connections Ethernet Bandwidth Assessment*. 2012.

[4] P GODFREY, Scott SHENKER et Ion STOICA. *Minimizing churn in distributed systems*. T. 36. 4. ACM, 2006.

[5] Yvonne COADY et al. "Distributed Cloud Computing (Dagstuhl Seminar 15072)". In : *Dagstuhl Reports* 5.2 (2015). Sous la dir. d'Yvonne COADY et al., p. 64–79. ISSN : 2192-5283. DOI : http://dx.doi.org/10.4230/DagRep.5.2.64. URL : http://drops.dagstuhl.de/opus/volltexte/2015/5045.

[6] Adrien LÈBRE, Jonathan PASTOR et Frédéric DESPREZ. *A Ring to Rule Them All - Revising OpenStack Internals to Operate Massively Distributed Clouds*. Technical Report RT-0480. INRIA, fév. 2016, p. 1–24. URL : https://hal.inria.fr/hal-01320235.

[7] Flavien QUESNEL, Adrien LÈBRE et Mario SÜDHOLT. "Cooperative and reactive scheduling in large-scale virtualized platforms with DVMS". In : *Concurrency and Computation : Practice and Experience* 25.12 (2013), p. 1643–1655.

[8] Flavien QUESNEL et al. "Advanced Validation of the DVMS Approach to Fully Distributed VM Scheduling". In : *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*. IEEE. 2013, p. 1249–1256.

[9] Jonathan PASTOR et al. "Locality-Aware Cooperation for VM Scheduling in Distributed Clouds". In : *Euro-Par 2014 Parallel Processing*. Springer, 2014, p. 330–341.

[10] Adrien LEBRE, Jonathan PASTOR et Mario SÜDHOLT. "VMPlaceS : A Generic Tool to Investigate and Compare VM Placement Algorithms". In : *Euro-Par 2015 : Parallel Processing*. Springer, 2015, p. 317–329.

[11] Alan Mathison TURING. “On computable numbers, with an application to the Entscheidungsproblem”. In : *J. of Math* 58.345-363 (1936), p. 5.

[12] Arthur Walter BURKS. “From ENIAC to the stored program computer : two revolutions in computers”. In : (1978).

[13] Chris MACK et al. “Fifty years of Moore’s law”. In : *Semiconductor Manufacturing, IEEE Transactions on* 24.2 (2011), p. 202–207.

[14] Michael HAUBEN. “History of ARPANET”. In : *Retrieved* 23 (2010), p. 13.

[15] Ian FOSTER et Carl KESSELMAN. “The history of the grid”. In : *computing* 20.21 (2010), p. 22.

[16] Douglas THAIN, Todd TANNENBAUM et Miron LIVNY. “Condor and the Grid”. In : *Grid computing : Making the global infrastructure a reality* (2003), p. 299–335.

[17] Ian FOSTER et al. “Cloud computing and grid computing 360-degree compared”. In : *Grid Computing Environments Workshop, 2008. GCE’08*. Ieee. 2008, p. 1–10.

[18] Luis M VAQUERO et al. “A break in the clouds : towards a cloud definition”. In : *ACM SIGCOMM Computer Communication Review* 39.1 (2008), p. 50–55.

[19] K. L. JACKSON. *An ontology for tactical cloud computing*. 2009. URL : http://kevinljackson.blogspot.fr/2009/03/ontology-for-tactical-cloud-computing.html.

[20] Lamia YOUSEFF et al. “Understanding the cloud computing landscape”. In : *Cloud Computing and Software Services* (2010), p. 1.

[21] James E SMITH et Ravi NAIR. “The architecture of virtual machines”. In : *Computer* 38.5 (2005), p. 32–38.

[22] Fabien HERMENIER et al. “Entropy : a consolidation manager for clusters”. In : *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM. 2009, p. 41–50.

[23] Dirk MERKEL. “Docker : lightweight linux containers for consistent development and deployment”. In : *Linux Journal* 2014.239 (2014), p. 2.

[24] Rajkumar BUYYA et al. “Cloud computing and emerging IT platforms : Vision, hype, and reality for delivering computing as the 5th utility”. In : *Future Generation computer systems* 25.6 (2009), p. 599–616.

[25] Rafael MORENO-VOZMEDIANO, Rubén S MONTERO et Ignacio M LLORENTE. “IaaS cloud architecture : From virtualized datacenters to federated cloud infrastructures”. In : *Computer* 12 (2012), p. 65–72.

[26] Bob LANTZ, Brandon HELLER et Nick MCKEOWN. “A network in a laptop : rapid prototyping for software-defined networks”. In : *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM. 2010, p. 19.

[27] OpenStack FOUNDATION. *OpenStack user stories*. 2015. URL : https://www.openstack.org/user-stories/.

[28] Ian FOSTER. “Globus toolkit version 4 : Software for service-oriented systems”. In : *Network and parallel computing*. Springer, 2005, p. 2–13.

[29] GLOBUS. *About the Globus Toolkit*. 2015. URL : http://toolkit.globus.org/toolkit/about.html.

[30] Daniel NURMI et al. “The eucalyptus open-source cloud-computing system”. In : *Cluster Computing and the Grid, 2009. CCGRID’09. 9th IEEE/ACM International Symposium on*. IEEE. 2009, p. 124–131.

[31] Borja SOTOMAYOR et al. “Virtual infrastructure management in private and hybrid clouds”. In : *Internet computing, IEEE* 13.5 (2009), p. 14–22.

[32] OPENNEBULA.ORG. *OpenNebula architecture*. 2015. URL : http://archives.opennebula.org/_detail/documentation:rel1.2:one-architecture.png?id=documentation%3Aarchives%3Arel2.0%3Aarchitecture.

[33] Kenneth CHURCH, Albert G GREENBERG et James R HAMILTON. “On Delivering Embarrassingly Distributed Cloud Services.” In : *HotNets*. Citeseer. 2008, p. 55–60.

[34] Mahadev SATYANARAYANAN et al. “The case for vm-based cloudlets in mobile computing”. In : *Pervasive Computing, IEEE* 8.4 (2009), p. 14–23.

[35] Eduardo PINHEIRO, Wolf-Dietrich WEBER et Luiz André BARROSO. “Failure Trends in a Large Disk Drive Population.” In : *FAST*. T. 7. 2007, p. 17–23.

[36] Amazon AWS. *Amazon Outage 29th June 2012*. 2015. URL : https://aws.amazon.com/message/67457/.

[37] Sharon CHOY et al. “The brewing storm in cloud gaming : A measurement study on cloud to end-user latency”. In : *Proceedings of the 11th annual workshop on network and systems support for games*. IEEE Press. 2012, p. 2.

[38] D MARGULIUS. “Apps on the Edge”. In : *InfoWorld* 24.21 (2002), p. 898–909.

[39] Benny ROCHWERGER et al. “Reservoir-when one cloud is not enough”. In : *Computer* 3 (2011), p. 44–51.

[40] Katarzyna KEAHEY et al. “Sky computing”. In : *Internet Computing, IEEE* 13.5 (2009), p. 43–51.

[41] Michael MATTESS, Christian VECCHIOLA et Rajkumar BUYYA. “Managing peak loads by leasing cloud infrastructure services from a spot market”. In : *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*. IEEE. 2010, p. 180–188.

[42] Flavio BONOMI et al. “Fog computing and its role in the internet of things”. In : *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM. 2012, p. 13–16.

[43] Luis M VAQUERO et Luis RODERO-MERINO. “Finding your way in the fog : Towards a comprehensive definition of fog computing”. In : *ACM SIGCOMM Computer Communication Review* 44.5 (2014), p. 27–32.

[44] Pedro Garcia Lopez et al. "Edge-centric Computing : Vision and Challenges". In : *SIGCOMM Comput. Commun. Rev.* 45.5 (sept. 2015), p. 37–42. ISSN : 0146-4833. DOI : 10.1145/2831347.2831354. URL : http://doi.acm.org/10.1145/2831347.2831354.

[45] Stefan Saroiu et al. "An analysis of internet content delivery systems". In : *ACM SIGOPS Operating Systems Review* 36.SI (2002), p. 315–327.

[46] Rashid Mijumbi et al. "Network function virtualization : State-of-the-art and research challenges". In : *IEEE Communications Surveys & Tutorials* 18.1 (2015), p. 236–262.

[47] Shanhe Yi, Cheng Li et Qun Li. "A Survey of Fog Computing : Concepts, Applications and Issues". In : (2015).

[48] Tom H Luan et al. "Fog computing : Focusing on mobile users at the edge". In : *arXiv preprint arXiv :1502.01815* (2015).

[49] Mahadev Satyanarayanan et al. "Cloudlets : at the leading edge of mobile-cloud convergence". In : *Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on*. IEEE. 2014, p. 1–9.

[50] Tim Verbelen et al. "Cloudlets : bringing the cloud to the mobile user". In : *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM. 2012, p. 29–36.

[51] Zhi Yang et al. "AmazingStore : available, low-cost online storage service using cloudlets." In : *IPTPS*. T. 10. 2010, p. 2–2.

[52] James Hamilton. "Cooperative expendable micro-slice servers (CEMS) : low cost, low power servers for internet-scale services". In : *Conference on Innovative Data Systems Research (CIDR'09)(January 2009)*. Citeseer. 2009.

[53] OpenStack Foundation. *Architectural Design : Massively scalable*. 2015. URL : http://docs.openstack.org/arch-design/massively-scalable.html.

[54] Rick Cattell. "Scalable SQL and NoSQL data stores". In : *ACM SIGMOD Record* 39.4 (2011), p. 12–27.

[55] CERN. *Initial Experience with Openstack Nova Cells*. URL : https://blog.zhaw.ch/icclab/initial-experience-with-openstack-nova-cells/.

[56] Tim Bell et al. "Scaling the CERN OpenStack cloud". In : *Journal of Physics : Conference Series*. T. 664. 2. IOP Publishing. 2015, p. 022003.

[57] Codership. *Galera cluster for MySQL*. URL : http://galeracluster.com/.

[58] Adrien Lèbre et al. "Beyond The Clouds, How Should Next Generation Utility Computing Infrastructures Be Designed ?" In : *Cloud Computing : Challenges, Limitations and R&D Solutions* (2014).

[59] Brett JONES et al. "RabbitMQ Performance and Scalability Analysis". In : *project on CS* 4284 ().

[60] Pieter HINTJENS. *ZeroMQ : Messaging for Many Applications.* " O'Reilly Media, Inc.", 2013.

[61] Giuseppe DECANDIA et al. "Dynamo : amazon's highly available key-value store". In : *ACM SIGOPS Operating Systems Review*. T. 41. 6. ACM. 2007, p. 205–220.

[62] Bryan CARPENTER et al. "Object serialization for marshalling data in a Java interface to MPI". In : *Proceedings of the ACM 1999 conference on Java Grande*. ACM. 1999, p. 66–71.

[63] Mike BURROWS. "The Chubby lock service for loosely-coupled distributed systems". In : *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, p. 335–350.

[64] Patrick HUNT et al. "ZooKeeper : Wait-free Coordination for Internet-scale Systems." In : *USENIX Annual Technical Conference*. T. 8. 2010, p. 9.

[65] REDISLAB. *Redis : REmote DIctionary Server*. 2015. URL : http://redis.io.

[66] Michael KIRCHER. "Lazy Acquisition." In : *EuroPLoP*. Citeseer. 2001, p. 151–164.

[67] Justin ZOBEL, Alistair MOFFAT et Ron SACKS-DAVIS. "An efficient indexing technique for full-text database systems". In : *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE). 1992, p. 352–352.

[68] Daniel BALOUEK et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In : *Cloud Computing and Services Science*. Sous la dir. d'IvanI. IVANOV et al. T. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, p. 3–20. ISBN : 978-3-319-04518-4. DOI : 10.1007/978-3-319-04519-1_1.

[69] Matthieu IMBERT et al. "Using the EXECO Toolbox to Perform Automatic and Reproducible Cloud Experiments". In : *1st Int. Workshop on UsiNg and building ClOud Testbeds (UNICO, collocated with IEEE CloudCom*. Déc. 2013. URL : https://hal.inria.fr/hal-00861886.

[70] Fabien HERMENIER, Julia LAWALL et Gunter MULLER. "Btrplace : A flexible consolidation manager for highly available applications". In : *Dependable and Secure Computing, IEEE Transactions on* 10.5 (2013), p. 273–286.

[71] Eugen FELLER, Louis RILLING et Christine MORIN. "Snooze : A scalable and autonomic virtual machine management framework for private clouds". In : *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society. 2012, p. 482–489.

[72] Eugen FELLER, Christine MORIN et Armel ESNAULT. “A case for fully decentralized dynamic VM consolidation in clouds”. In : *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*. IEEE. 2012, p. 26–33.

[73] Ion STOICA et al. “Chord : A scalable peer-to-peer lookup service for internet applications”. In : *ACM SIGCOMM Computer Communication Review* 31.4 (2001), p. 149–160.

[74] Luis GARCES-ERICE et al. “Hierarchical peer-to-peer systems”. In : *Parallel Processing Letters* 13.04 (2003), p. 643–657.

[75] Z. XU, M. MAHALINGAM et M. KARLSSON. “Turning Heterogeneity into an Advantage in Overlay Routing”. In : *INFOCOM*. 2003.

[76] Z. XU et Z. ZHANG. *Building Low-Maintenance Expressways for P2P Systems*. Rapp. tech. HPL-2002-41. Hewlett-Packard Labs, 2002.

[77] Antony I. T. ROWSTRON et Peter DRUSCHEL. “Pastry : Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems”. In : *IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware)*. Nov. 2001.

[78] Márk JELASITY et Ozalp BABAOGLU. “T-Man : Gossip-based Overlay Topology Management”. In : *In Third International Workshop on Engineering Self-Organising Applications (ESOA’05)*. Springer-Verlag, 2005, p. 1–15.

[79] Frank DABEK et al. “Vivaldi : A Decentralized Network Coordinate System”. In : *2004 conf. on Applications, technologies, architectures, and protocols for computer comm.* SIGCOMM ’04. 2004, p. 15–26.

[80] Carl HEWITT, Peter BISHOP et Richard STEIGER. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In : *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI’73. Stanford, USA : Morgan Kaufmann Publishers Inc., 1973, p. 235–245.

[81] Frank DABEK et al. “Vivaldi : A decentralized network coordinate system”. In : *ACM SIGCOMM Computer Communication Review*. T. 34. 4. ACM. 2004, p. 15–26.

[82] Henri CASANOVA et al. “Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms”. In : *Parallel and Distributed Computing* 74.10 (juin 2014).

[83] SIMGRID. *Publications du projet Simgrid*. URL : http://simgrid.gforge.inria.fr/Publications.html.

[84] Takahiro HIROFUCHI, Adrien LÈBRE et Laurent POUILLOUX. “Adding a live migration model into simgrid : One more step toward the simulation of infrastructure-as-a-service concerns”. In : *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*. T. 1. IEEE. 2013, p. 96–103.

[85] Malte SCHWARZKOPF et al. “Omega : Flexible, Scalable Schedulers for Large Compute Clusters”. In : *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys ’13. ACM, 2013, p. 351–364.

[86] Rodrigo N. CALHEIROS et al. "CloudSim : a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". In : *Software : Practice and Experience* 41.1 (2011).

[87] John CARTLIDGE et Dave CLIFF. "Comparison of Cloud Middleware Protocols and Subscription Network Topologies using CReST". In : *CLOSER'13*. Mai 2013.

[88] Alexandru IOSUP et al. "DGSim : Comparing Grid Resource Management Architectures through Trace-Based Simulation". In : *Euro-Par'08*. LNCS 5168. Springer, août 2008.

[89] D. KLIAZOVICH et al. "GreenCloud : A Packet-Level Simulator of Energy-Aware Cloud Computing Data Centers". In : *IEEE GlobeCom'10*. Déc. 2010.

[90] Alberto NUNEZ, Jose L. VAZQUEZ-POLETTI et al. "iCanCloud : A Flexible and Scalable Cloud Infrastructure Simulator". English. In : *Journal of Grid Computing* 10.1 (2012), p. 185–209. ISSN : 1570-7873. DOI : 10.1007/s10723-012-9208-5.

[91] Takahiro HIROFUCHI, Adrien LEBRE et Laurent POUILLOUX. "Adding a Live Migration Model into SimGrid : One More Step Toward the Simulation of Infrastructure-as-a-Service Concerns". In : *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01*. CLOUDCOM '13. Washington, DC, USA : IEEE Computer Society, 2013, p. 96–103. ISBN : 978-0-7695-5095-4. DOI : 10.1109/CloudCom.2013.20. URL : http://dx.doi.org/10.1109/CloudCom.2013.20.

[92] Robert KALLMAN et al. "H-store : a high-performance, distributed main memory transaction processing system". In : *Proceedings of the VLDB Endowment* 1.2 (2008), p. 1496–1499.

[93] James C CORBETT et al. "Spanner : Google's globally distributed database". In : *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[94] Jeff SHUTE et al. "F1-The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business". In : (2012).

[95] COCKROACHDB.ORG. *A Scalable, Survivable, Strongly-Consistent SQL Database*. Juin 2016. URL : https://github.com/cockroachdb/cockroach.

[96] PINGCAP. *TiDB is a distributed NewSQL database compatible with MySQL protocol*. Juin 2016. URL : https://github.com/pingcap/tidb.

UNIVERSITE BRETAGNE LOIRE

MINES Nantes

Institut Mines-Télécom

Thèse de Doctorat

Jonathan PASTOR

Contributions à la mise en place d'une infrastructure de Cloud Computing à large échelle.

Contributions to massively distributed Cloud Computing infrastructures.

Résumé
La croissance continue des besoins en puissance de calcul a conduit au triomphe du modèle de Cloud Computing. Des clients demandeurs en puissance de calcul vont s'approvisionner auprès de fournisseurs d'infrastructures de Cloud Computing, mises à disposition via Internet. Pour réaliser des économies d'échelles, ces infrastructures sont toujours plus grandes et concentrées en quelques endroits, conduisant à des problèmes tels que l'approvisionnement en énergie, la tolérance aux pannes et l'éloignement des utilisateurs.
Cette thèse s'est intéressée à la mise en place d'un système d'IaaS massivement distribué et décentralisé exploitant un réseau de micros centres de données déployés sur la dorsale Internet, utilisant une version d'OpenStack revisitée pendant cette thèse autour du support non intrusif de bases de données non relationnelles. Des expériences sur Grid'5000 ont montré des résultats intéressants sur le plan des performances, toutefois limités par le fait qu'OpenStack ne tirait pas avantage nativement d'un fonctionnement géographiquement réparti.
Nous avons étudié la prise en compte de la localité réseau pour améliorer les performances des services distribués en favorisant les collaborations proches. Un prototype de l'algorithme de placement de machines virtuelles DVMS, fonctionnant sur une topologie non structurée basée sur l'algorithme Vivaldi, a été validé sur Grid'5000. Ce prototype a fait l'objet d'un prix scientifique lors de l'école de printemps Grid'5000 2014. Enfin, ces travaux nous ont amenés à participer au développement du simulateur VMPlaceS.

Mots clés
Cloud Computing, Infrastructure IaaS, nanos centres de données, OpenStack, Bases de données non relationnelles

Abstract
The continuous increase of computing power needs has favored the triumph of the Cloud Computing model. Customers asking for computing power will receive supplies via Internet resources hosted by providers of Cloud Computing infrastructures. To make economies of scale, Cloud Computing that are increasingly large and concentrated in few attractive places, leading to problems such energy supply, fault tolerance and the fact that these infrastructures are far from most of their end users.
During this thesis we studied the implementation of an fully distributed and decentralized IaaS system operating a network of micros data-centers deployed in the Internet backbone, using a modified version of OpenStack that leverages non relational databases. A prototype has been experimentally validated on Grid'5000, showing interesting results, however limited by the fact that OpenStack doesn't take advantage of a geographically distributed functioning.
Thus, we focused on adding the support of network locality to improve performance of Cloud Computing services by favoring collaborations between close nodes. A prototype of the DVMS algorithm, working with an unstructured topology based on the Vivaldi algorithm, has been validated on Grid'5000. This prototype got the first prize at the large scale challenge of the Grid'5000 spring school in 2014. Finally, the work made with DVMS enabled us to participate at the development of the VMPlaceS simulator.

Key Words
Cloud Computing, Infrastructure IaaS, nanos datacenters, OpenStack, non relational databases.