Services de répartition de charge pour le Cloud : application au traitement de données multimédia

Sylvain Lefebvre

CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS | le cnam

École Doctorale Informatique, Télécommunications et Électronique (Paris)

CEDRIC

THÈSE DE DOCTORAT

présentée par : **Sylvain LEFEBVRE**

soutenue le : **10 Décembre 2013**

pour obtenir le grade de : **Docteur du Conservatoire National des Arts et Métiers**

Spécialité : **INFORMATIQUE**

Services de répartition de charge pour le Cloud : Application au traitement de données multimédia

THÈSE DIRIGÉE PAR

M. GRESSIER-SOUDAN Eric — *Professeur, CEDRIC-CNAM, Paris*

ENCADRÉE PAR

Mme. CHIKY Raja — *Enseignant-Chercheur, LISITE-ISEP, Paris*

RAPPORTEURS

Mme. MORIN Christine — *Chercheur Titulaire, INRIA-IRISA, Rennes*
M. PIERSON Jean-Marc — *Professeur, IRIT, Université Paul Sabatier, Toulouse*

EXAMINATEURS

M. ROOSE Philippe — *Directeur de recherche, LIUPPA, Pau*
M. SENS Pierre — *Directeur de recherche, LIP6, UPMC, Paris*
M. PAWLAK Renaud — *Directeur technique, IDCapture, Paris*
Mme. SAILHAN Françoise — *Maitre de conférence, CEDRIC-CNAM, Paris*

"When they first built the University of California at Irvine they just put the buildings in. They did not put any sidewalks, they just planted grass. The next year, they came back and put the sidewalks where the trails were in the grass."

Larry Wall

Remerciements

Les travaux menés durant ces trois ans ont été possible grâce à la confiance et aux encouragements que mes encadrants Eric Gressier-Soudan, Renaud Pawlak puis Raja Chiky m'ont accordé, à la fois lors du recrutement et tout au long de ces 36 derniers mois. Je tiens à remercier particulièrement Raja Chiky, pour son optimisme, ses conseils et ses encouragements en toutes circonstances.

Je tiens aussi à remercier tous les membres de l'équipe de Recherche et Développement Informatique du LISITE : Yousra pour les filtres de Bloom, Zakia, Bernard et Matthieu pour leurs conseils et commentaires avisés, et Olivier pour les probabilités. En restant à l'ISEP je remercie aussi mes collègues de bureaux : Giang, Sathya, Adam, Ahmad, Fatima, Maria et Ujjwal.

Mes sincères remerciements vont aux membres du jury qui ont accepté de juger mon travail : MM. Pierson et Roose, Mme Morin ainsi que le professeur P. Sens pour ses conseils lors de ma soutenance de mi-parcours et pour m'avoir donné accès à la plateforme GRID5000. Je dois aussi adresser des remerciements à ceux qui m'ont encouragé dans ma poursuite d'études : Professeurs B. Marchal, F. Pommereau, M. Bernichi de l'ESIAG et A. Ung d'Essilor sans lesquels je ne me serais pas lancé dans cette aventure.

Enfin, ce travail n'aurait pu être mené sans les encouragements de ma famille, de mes amis, et de Clara.

Résumé

Les progrès conjugués de l'algorithmique et des infrastructures informatiques permettent aujourd'hui d'extraire de plus en plus d'informations pertinentes et utiles des données issues de réseaux de capteurs ou de services web. Ces plateformes de traitement de données "massives" sont confrontées à plusieurs problèmes. Un problème de répartition des données : le stockage de ces grandes quantités de données impose l'agrégation de la capacité de stockage de plusieurs machines. De là découle une seconde problématique : les traitements à effectuer sur ces données doivent être à leur tour répartis efficacement de manière à ne surcharger ni les réseaux, ni les machines.

Le travail de recherche mené dans cette thèse consiste à développer de nouveaux algorithmes de répartition de charge pour les systèmes logiciels de traitement de données massives. Le premier algorithme mis au point, nommé "WACA" (Workload and Cache Aware Algorithm) améliore le temps d'exécution des traitements en se basant sur des résumés des contenus stockés sur les machines. Le second algorithme appelé "CAWA" (Cost Aware Algorithm) tire partie de l'information de coût disponible dans les plateformes de type "Cloud Computing" en étudiant l'historique d'exécution des services.

L'évaluation de ces algorithmes a nécessité le développement d'un simulateur d'infrastructures de "Cloud" nommé Simizer, afin de permettre leur test avant le déploiement en conditions réelles. Ce déploiement peut se faire de manière transparente grâce au système de distribution et de surveillance de service web nommé "Cloudizer", développé aussi dans le cadre de cette thèse, qui a servi à l'évaluation des algorithmes sur l'Elastic Compute Cloud d'Amazon.

Ces travaux s'inscrivent dans le cadre du projet de plateforme de traitement de données Multimédia for Machine to Machine (MCube). Ce projet requiert une infrastructure logicielle adaptée au stockage et au traitement d'une grande quantité de photos et de sons, issus de divers réseaux de capteurs. La spécificité des traitements utilisés dans ce projet a nécessité le développement d'un service d'adaptation automatisé des programmes vers leur environnement d'exécution.

Mots clés : Répartition de charge, Cloud, Données Massives

Abstract

Nowadays, progresses in algorithmics and computing infrastructures allow to extract more and more adequate and useful information from sensor networks or web services data. These big data computing platforms have to face several challenges. A data partitioning issue ; storage of large volumes imposes aggregating several machines capacity. From this problem arises a second issue : to compute these data, tasks must be distributed efficiently in order to avoid overloading networks and machines capacities. The research work carried in this thesis consists in developping new load balancing algorithms for big data computing software. The first designed algorithm, named WACA (Workload And Cache Aware algorithm) enhances computing latency by using summaries for locating data in the system. The second algorithm, named CAWA for "Cost AWare Algorithm", takes advantage of cost information available on so-called "Cloud Computing" platforms by studying services execution history. Performance evaluation of these algorithms required designing a Cloud Infrastructure simulator named "Simizer", to allow testing prior to real setting deployment. This deployement can be carried out transparently thanks to a web service monitoring and distribution framework called "Cloudizer", also developped during this thesis work and was used to assess these algorithms on the Amazon Elastic Compute Cloud platform.

These works are set in the context of data computing platform project called "Multimedia for Machine to Machine" (MCube). This project requires a software infrastructure fit to store and compute a large volume of pictures and sounds, collected from sensors. The specifics of the data computing programs used in this project required the development of an automated execution environement adaptation service.

Keywords : Load balancing, Cloud Computing, Big Data

Avant propos

Cette thèse se déroule dans l'équipe Recherche et Développement en Informatique (RDI) du laboratoire Laboratoire d'Informatique, Signal et Image, Électronique et Télécommunication (LISITE) de l'Institut Supérieur d'Électronique de Paris. L'équipe RDI se focalise principalement sur la thématique des systèmes complexes avec deux axes forts. Le premier axe vise la fouille de données et l'optimisation dans ces systèmes. L'optimisation est en effet pour la plupart du temps liée à la traçabilité et à l'analyse des données, en particulier les données liées aux profils d'utilisation. Le deuxième axe concerne l'étude de langages, de modèles, et de méthodes formelles pour la simulation et la validation des systèmes complexes, en particulier les systèmes biologiques et les systèmes embarqués autonomes sous contraintes fortes.

Les travaux de cette thèse s'inscrivent dans le cadre d'un projet européen de mise au point de plateforme de services Machine à Machine permettant d'acquérir et d'assurer l'extraction et l'analyse de données multimédia issues de réseaux de capteurs. Ce projet, baptisé Mcube pour *Multimedia for machine to machine* (Multimédia pour le machine à machine) est développé en partenariat avec le fond Européen de Développement Régional et deux entreprises de la région Parisienne : CAP 2020 et Webdyn. Il permet de réunir les différentes problématiques de recherches de l'équipe RDI au sein d'un projet commun, en partenariat avec les membres de l'équipe de recherche en traitement du signal du LISITE.

La particularité de ce projet est qu'il se concentre sur la collecte de données multimédias. Les difficultés liées au traitement de ces données nécessitent de faire appel à des technologies spécifiques pour leur stockage. Cette plateforme ne se limite pas au stockage des données, et doit fournir aux utilisateurs la possibilité d'analyser les données récupérées en ligne. Or, ces traitements peuvent s'exécuter dans un environnement dynamique soumis à plusieurs contraintes comme le coût financier, ou la consommation énergétique. La répartition des traitements joue donc un rôle prépondérant dans la garantie de ces contraintes. La répartition des données sur les machines joue aussi un rôle important dans la performance des traitements, en permettant de limiter la quantité de données à transférer.

Ces travaux ont été encadrés par Dr. Renaud Pawlak puis suite à son départ de l'ISEP en septembre 2011, par Dr. Raja Chiky, sous la responsabilité du professeur Eric Gressier-Soudan de l'équipe Systèmes Embarqués et Mobiles pour l'Intelligence Ambiante du CEDRIC-CNAM.

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

Ces dernières années, le besoin de traiter des quantités de données de plus en plus grandes s'est fortement développé. Les applications possibles de ces analyses à grande échelle vont de l'indexation de pages web à l'analyse de données physiques, biologiques ou environnementales. Le développement de nouvelles plateformes de services informatiques comme les réseaux de capteurs et les "nuages", ces plateformes de services de calcul à la demande, accentuent ce besoin tout en facilitant la mise en œuvre de ces outils. Les réseaux de capteurs ne sont pas une technologie récente, mais la capacité de traiter des grandes quantités de données issues de ces réseaux est aujourd'hui facilement accessible à un grand nombre d'organisations, en raison du développement de l'informatique en "nuage" : le *Cloud Computing*. Ce nouveau modèle de service informatique est, d'après la définition du National Institute of Standards and Technologies (NIST), un service mutualisé, accessible par le réseau, qui permet aux organisations de déployer rapidement et à la demande des ressources informatiques [MG11]. Dans ce contexte, les applications combinant plateformes d'acquisition de données autonomes et les "nuages" pour traiter les données acquises, se développent de plus en plus et permettent de fournir de nouveaux services en matière d'analyse de données, dans le domaine commercial [Xiv13], ou académique [BMHS13, LMHJ10].

Cadre applicatif

Les travaux de cette thèse s'inscrivent dans le cadre d'un projet de recherche financé par le Fond Européen de DEveloppement Régional (FEDER) d'Ile de France [1], nommé Multimedia pour le Machine à Machine (MCube) [2]. Ce projet consiste à mettre au point un ensemble de technologies permettant de collecter, transférer, stocker et traiter des données multimédias collectées à partir de passerelles autonomes déployées géographiquement. L'objectif est d'en extraire des informations utiles pour les utilisateurs du service. Par exemple, il peut s'agir de détecter la présence d'une certaine espèce d'insecte nuisible dans un enregistrement sonore, afin de permettre à l'agriculteur d'optimiser le traitement insecticide de ses cultures, en épandant que lorsque cela est nécessaire.

Ce projet pose plusieurs difficultés de réalisation : en premier lieu, l'intégration et le stockage de volumes de données importants, pouvant venir de différentes sources comme un réseau de capteurs ou de multiples services WEB, implique de sélectionner les technologies appropriées pour traiter et stocker ces données[Lan01]. À titre d'exemple, les cas d'études utilisés pour le projet MCube consistent en un suivi photographique du développement de plans de tomates et des écoutes sonores pour la détection de nuisibles. Ces deux cas d'études effectués sur un seul champ durant les trois campagnes d'acquisition du projet (2011,2012 et 2013) ont nécessité la collecte d'un téraoctet et demi de données, soit environ 466,6 gigaoctets par utilisateur et par an. Le service définitif devant s'adresser à un nombre plus important de clients, l'espace requis pour ces deux applications représentera donc un volume non négligeable, en utilisation réelle. Le deuxième aspect de ces travaux consiste à mettre au point un système pouvant adapter ces applications à un environnement dynamique tel que les plateformes de *cloud computing*, de manière à pouvoir bénéficier de ressources importantes à la demande lorsqu'un traitement le requiert.

La troisième difficulté concerne les traitements d'images eux mêmes, qui sont réalisés par des scientifique ayant peu ou pas d'expériences de la programmation exceptés dans des langages spécifiques comme MATLAB [3], dont l'utilisation sur les plateforme de type

1. `http://www.europeidf.fr/index.php`, consulté le 5 octobre 2013
2. `http://mcube.isep.fr/mcube`, consulté le 5 octobre 2013
3. `http://www.mathworks.fr/products/matlab/`, consulté le 5 octobre 2013

"nuage" n'est pas facile. En effet, ces programmes ne sont pas écrits par des experts de la programmation parallèle ou des calculs massifs, mais sont écrits pour traiter directement un ou deux fichiers à la fois. Or, le contexte du projet MCube impose de traiter de grandes quantités de fichiers avec ces mêmes programmes, ce qui requiert un système permettant de les intégrer le plus simplement possible dans un environnement parallèle.

Ces trois verrous s'inscrivent dans des problématiques scientifiques plus larges liées aux "nuages" et aux services distribués en général.

Problématiques scientifiques

Les problématiques scientifiques abordées dans ce document sont au nombre de trois. La première d'entre elle consiste à assurer la répartition de requêtes dans un système distribué déployé sur un "Cloud", en tenant compte des aspects particuliers de ce type de plateformes. La deuxième problématique, plus spécifique au projet MCube, concerne l'adaptation automatique de traitements à leur environnement d'exécution. Troisièmement, Le développement de nouvelles stratégies de répartition a nécessité de nombreux tests en amont du déploiement sur une infrastructure réelle, or il n'existe pas, à notre connaissance, de simulateur d'applications distribuées sur le "cloud" approprié pour tester ce type de développements.

Distribution de requêtes dans le "Cloud" Les stratégies de répartition fournies par les services de "Cloud Computing" sont limitées dans leurs fonctionnalités ou leur capacité. Par exemple, le répartiteur élastique de la plateforme Amazon Web Services (ELB : Elastic Load Balancer), distribue les requêtes selon une stratégie en tourniquet (Round Robin) sur les machines les moins chargées du système [4]. Seule cette stratégie de répartition est disponible pour les utilisateurs. Or, les applications déployées sur les plateformes d'informatique en nuage sont diverses, par conséquent une stratégie générique n'est pas toujours adaptée pour obtenir la meilleure performance du système. De plus, Liu et Wee [LW09] ont démontré que déployer son propre répartiteur d'applications web sur une machine vir-

4. `https://forums.aws.amazon.com/message.jspa?messageID=129199#129199`, consulté le 5 octobre 2013

tuelle dédiée était plus économique et permettait de répondre à un plus grand nombre de requêtes concurrentes que le service de répartition de charge élastique fournit par Amazon. Les fournisseurs de plateformes Cloud imposent souvent à leurs utilisateurs des stratégies de répartition génériques pour pouvoir répondre à un grand nombre de cas d'utilisation, mais ces stratégies ne sont pas toujours les plus efficaces : Un exemple notable est celui de la plateforme Heroku [5], qui utilisait une stratégie de répartition aléatoire pour attribuer les requêtes à différents processeurs. Or, cette répartition aléatoire se faisait sans que les multiples répartiteurs du système ne communiquent entre eux pour surveiller la charge des machines. Par conséquent, certains répartiteurs envoyaient des requêtes vers des machines déjà occupées, ce qui créait des files d'attentes inattendues [6]. Utiliser des stratégies de répartition de charge appropriées, tenant compte des spécificités des plateformes de type "Cloud Computing" est donc indispensable. Les spécificités de ces plateformes sont notamment :

Des performances difficilement prévisibles : L'infrastructure physique de ces plateformes est partagée par un grand nombre d'utilisateurs (cf. annexe A). La conséquence de ce partage est qu'en fonction de l'utilisation des ressources faite par les multiples utilisateurs, il peut arriver que deux machines virtuelles identiques sur le papier ne fournissent pas obligatoirement le même niveau de performance [BS10, IOY+11].

Des latences réseaux plus élevées : La taille des centres de données utilisés dans les plateformes de Cloud Computing, ainsi que leur répartition géographique a pour conséquence une latence plus importante dans les communications réseaux, qu'il convient de masquer le plus possible à l'utilisateur final.

Une plateforme d'informatique à la demande doit donc permettre aux utilisateurs de choisir la stratégie de distribution la plus adaptée à leur cas d'utilisation. De plus, il est nécessaire de prendre en compte les spécificités de ces environnements pour développer de nouvelles stratégies de répartition adaptées.

5. `https://www.heroku.com/`, consulté le 5 octobre 2013
6. `https://blog.heroku.com/archives/2013/2/16/routing_performance_update`, consulté le 5 octobre 2013

Adaptation d'applications de traitements de données multimédia Ces nouvelles plateformes sont par définition simples d'utilisation, flexibles et économiques. Cependant, pour tirer parti des avantages de ces services, il est nécessaire de mettre en place des processus automatisés de déploiement, de distribution et de configuration des applications [AFG+09, ZCB10, MG11]. Les plateformes de *cloud computing* promettent à leurs utilisateurs l'accès à une infinité de ressources à la demande, mais toutes les applications ne sont pas nécessairement conçues pour être distribuées dans un environnement dynamique. Quelles sont donc les caractéristiques des applications pouvant être déployées dans les environnements de type "Cloud Computing" ? Quels mécanismes peuvent être mis en œuvre pour assurer le déploiement de ces applications de manière plus ou moins transparente ? Comment amener ces traitements à s'exécuter de manière distribuée ou parallèle pour pouvoir tirer parti des avantages fournis par les plateformes de "Cloud Computing" ?

Test et performance A ces questions s'ajoute la problématique du test de l'application. Il n'existe à ce jour pas de simulateur fiable permettant d'évaluer le comportement d'applications déployées dans le Cloud, car les efforts actuels, tels que les simulateurs CloudSim[RNCB11] et SimGrid[CLQ08], se concentrent sur la simulation des infrastructures physiques supportant ces services plutôt que sur les applications utilisant ces services [SL13]. Or, le développement de protocoles et d'algorithmes distribués nécessite des tests sur un nombre important de machines. L'ajustement de certains paramètres peut requérir de multiples tests sur diverses configurations (nombre de machines, puissance, type de charge de travail), mais il n'est pas toujours aisé de devoir déployer des tests grandeur nature pour ce type d'ajustements. Il faut donc développer ou adapter les outils existants pour faciliter ces tests, en fournissant une interface de programmation de haut niveau pour développer les protocoles à tester, sans que l'utilisateur n'ait à programmer à la fois la simulation et le protocole qu'il souhaite étudier.

Approche développée

L'approche développée pour répondre à ces problématiques a consisté à mettre au point une plateforme de distribution de services web destinée aux environnements de type

Service d'Infrastructure à la Demande (Infrastructure As A Service, IaaS, cf. annexe A), permettant de déployer et de distribuer des services web ou des applications en ligne de commande en les transformant en services Web. Cette application, nommée Cloudizer, a permis l'étude de différentes stratégies de distribution et de leur impact sur les différentes applications déployées à travers plusieurs publications [PLCKA11, SL12, LKC13].

Cette plateforme a été utilisée dans le projet MCUBE, pour distribuer les programmes de traitements des images collectées. Les volumes de données concernés par ce projet sont tels que l'utilisation d'une stratégie de distribution des requêtes fondée sur la localité des données [DG04, PLCKA11] permet de réduire efficacement les temps de traitement. Dans ce domaine, les algorithmes de distribution les plus utilisés sont ceux fondés sur la technique du hachage cohérent [KLL$^+$97, KR04], qui est mise en œuvre dans plusieurs systèmes de stockage de données massives. Le but de ces travaux est de montrer à travers différentes simulations que les filtres de Bloom [Blo70, FCAB98, DSSASLP08] peuvent être facilement combinés à d'autres algorithmes de répartition de charge, pour fournir une alternative performante au hachage cohérent, dans certaines conditions. Ce travail a permis la mise au point d'une stratégie de répartition de charge fondée sur des filtres de Bloom indexés appelée WACA (Workload And Cache Aware Algorithm, Algorithme Renseigné sur la Charge et le Cache).

Les Clouds de type Infrastructure à la Demande constituent l'environnement de déploiement cible de ces services. La propriété de ces plateformes est de facturer leurs services à l'utilisation. Afin de prendre ce facteur en compte, nous avons développé une stratégie fondée sur le coût comme indicateur de performance, et nommée CAWA (Cost-AWare Algorithm, Algorithme Renseigné sur le Coût) dont le fonctionnement a été testé à travers des simulations. Tel que montré en section 1 de cette introduction, il n'existe pas encore de simulateur adéquat pour les utilisateurs de plateformes de *Cloud Computing*. Le logiciel Simizer a donc été développé pour permettre la simulation d'applications orientées services déployées sur des infrastructures Cloud. Ce simulateur a permis de simuler et tester les stratégies de distribution développées pour la plateforme Cloudizer, et d'étudier leur comportement à large échelle.

Organisation du présent document

Le reste de ce document est organisé en deux parties principales. La première partie, composée des chapitres 2 à 4, décrit les développements des différentes plateformes logicielles évoquées dans la section précédente. Le chapitre 2 présente les particularités du projet MCube et ce qui le différencie des plateformes de traitement de données multimédia existantes. Le chapitre 3 décrit le fonctionnement de la plateforme Cloudizer et son utilisation dans le cadre du projet MCUBE, puis le chapitre 4 décrit les particularités et le fonctionnement du simulateur Simizer.

La seconde partie de ce document se concentre sur les stratégies de distribution de requêtes mises au point au cours de cette thèse. L'état de l'art en matière de répartition de charge est présenté dans le chapitre 5. les chapitres suivants décrivent respectivement la stratégie WACA, (chapitre 6), qui utilise des résumés compacts pour assurer une distribution des tâches en fonction de la localisation des données dans le système, et la stratégie CAWA (chapitre 7) qui se fonde sur une estimation du coût d'exécution des tâches pour effectuer la répartition. Le chapitre 8 présentera les conclusions et perspectives de l'ensemble de ces travaux.

Chapitre 2

MCube : Une plateforme de stockage et de traitement de données multimédia

2.1 Introduction

L'objectif technique du projet MCube (Multimedia 4 Machine 2 Machine) est de développer une technologie Machine à Machine pour la capture et l'analyse de données multimédia par des réseaux de capteurs avec des problématiques de faible consommation et de transmission GPRS/EDGE. Le projet couvre la chaîne complète : l'acquisition des données, la transmission, l'analyse, le stockage, et le service d'accès par le WEB. Il s'appuie sur les infrastructures M2M actuellement offertes par les opérateurs du secteur comme les réseaux 3G/GPRS pour la communication des données. Le but de cette plateforme est de fournir des services d'analyse de données multimédia à faible coût permettant diverses applications telles que la surveillance de cultures et de sites industriels (détections d'intrusions ou d'insectes nuisibles).

2.1.1 Architecture du projet MCube

L'architecture globale de la technologie MCube correspond à l'état de l'art en matière d'architecture M2M et est résumée en figure 2.1. Les "passerelles" sont des systèmes embarqués permettant de piloter des périphériques d'acquisition de données multimédia comme des appareils photos ou des microphones USB. Les passerelles assurent la transmission des

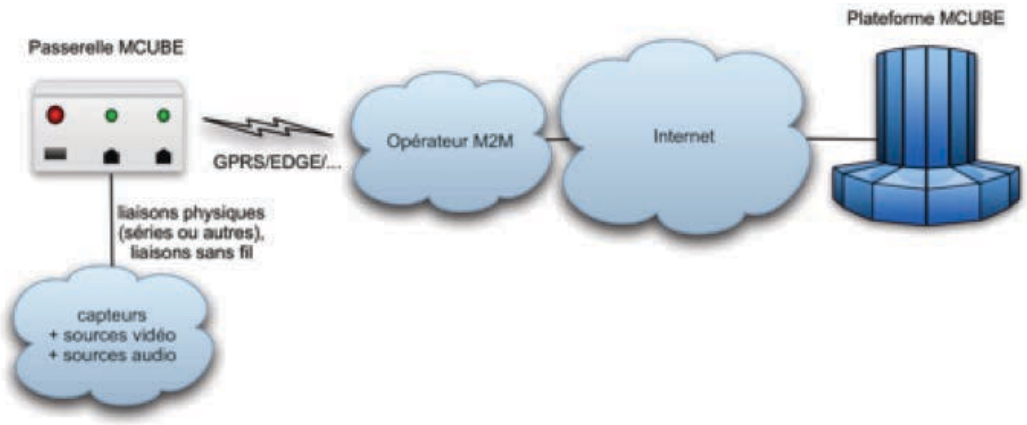


FIGURE 2.1 – Architecture du projet MCube

données collectées vers la plateforme MCube, via une connexion 3G si un accès Ethernet n'est pas disponible. La communication s'effectue via le protocole HTTP. La plateforme est un serveur accessible via le web, dont le rôle consiste à :

Stocker les données collectées La plateforme assure le stockage des données multimédia capturées et remontées par les passerelles. Ceci nécessite un large espace de stockage redondant et distribué afin de limiter les risques de pertes de données, ainsi que des méthodes efficaces pour retrouver les données stockées.

Configurer les passerelles Le système offre la possibilité de communiquer avec les passerelles afin de les mettre à jour dynamiquement et de changer différents paramètres tels que le type de capture à effectuer (sons, images ou vidéos) ou la fréquence de ces captures. Cette communication se fait via une interface de programmation prenant la forme d'un service web REST[Fie00], déployée sur les serveurs de la plateforme.

Traiter et analyser les données multimédia Les utilisateurs du système MCube peuvent développer leurs propres programmes d'analyse de données, et les stocker sur la plateforme pour former leur propre bibliothèque de traitements. Ces traitements doivent pouvoir être exécutés par lots, en parallèle sur un grand nombre de fichiers, mais aussi en temps réel au fur et à mesure de la collecte des données.

Par conséquent, mis à part le rôle de communication avec les passerelles, la plateforme MCube devrait reposer sur une base de données multimédia. Ces systèmes stockent et

traitent des données en volumes importants (plusieurs giga/téraoctets). Les données concernées peuvent être de types différents (sons, images, vidéos,...) et sont séparées en un nombre important de petits fichiers pouvant provenir de différents périphériques d'acquisition. De plus, ces données sont généralement peu ou pas structurées. Afin d'assurer l'efficacité des traitements sur les données multimédia, il est nécessaire de les exécuter de façon parallèle. À nouveau, le résultat de ces traitements doit être stocké dans le système, ce qui permet la réutilisation des données pour des analyses supplémentaires. La somme de toutes ces données exige une capacité de stockage importante.

2.1.2 Problématiques

Une contrainte primordiale du projet MCube est d'aboutir à un service suffisamment flexible pour pouvoir être personnalisé à chaque scénario d'utilisation, dans le but d'optimiser le modèle économique du client en fonction de nombreux critères. Ainsi, les traitements des données multimédia pourront être faits aussi bien côté passerelle que côté plateforme suivant les critères à optimiser. La plateforme fournira des services d'aide à la décision permettant d'optimiser au mieux la solution. Les choix d'architecture logicielle décrits en section 2.4 reflètent ces besoins.

La plateforme MCube permet d'exécuter des analyses poussées sur les données reçues des passerelles. Les algorithmes utilisés pour ces analyses sont souvent soit expérimentaux, soit développés spécifiquement pour certains clients, et ne sont pas disponibles dans les librairies de traitements de données multimédias existantes telles que Image Terrier[HSD11] ou OpenCV[Bra00]. La conséquence est que ces algorithmes ne sont pas toujours directement parallélisables. Or, la collecte d'un nombre important de fichiers dans les passerelles MCube impose de pouvoir être en mesure d'exécuter les traitements sur une grande quantité de données en parallèle.

Cet état de fait a des conséquences sur les choix d'architecture à effectuer car il est nécessaire de fournir un accès aux données à la fois au niveau fichier pour que les données soient accessibles aux différents programmes de traitements et un accès automatisé, via une interface de programmation, pour permettre aux utilisateurs d'interroger leurs données en temps réel.

Ce chapitre expose donc dans un premier temps les composants des systèmes de stockage et de traitement de données multimédias en section 2.2, et analyse les différentes plateformes existantes de ce domaine par rapport aux besoins de la plateforme MCube en section 2.3. La section 2.4 décrit la solution adoptée pour la mise au point de l'architecture de la plateforme MCube et les problématiques qui découlent de ces choix sont discutées en section 2.6.

2.2 Composants des plateformes de stockage de données multimédia

Il est possible de définir une plateforme de traitement de données multimédias comme un système logiciel distribué permettant de stocker des fichiers contenant des données photographiques, vidéos ou sonores, et de fournir les services nécessaires à l'extraction et au stockage d'informations structurées issues de ces fichiers.

Les systèmes d'analyse de données multimédia possèdent un ensemble hiérarchisé de composants permettant de transformer les données brutes en informations précises [CSC09, HSD11]. Le schéma 2.2 montre l'articulation de ces différents composants, que l'on retrouvera dans la plupart des systèmes décrits dans ce chapitre.

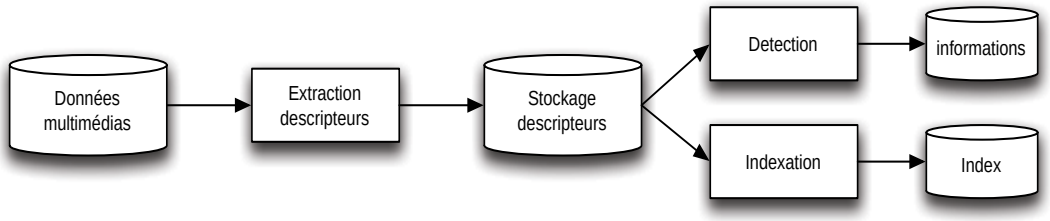


FIGURE 2.2 – Architecture pour la fouille de données multimédia

Le premier composant de ces systèmes correspond à la couche de stockage des données. Ce composant doit assurer le stockage et la disponibilité des données brutes, c'est à dire les documents audiovisuels dans leur format d'origine (RAW, MPEG, AVI, WAV), mais aussi des données intermédiaires nécessaires pour les traitements (les descripteurs) et les résultats des algorithmes d'indexation et d'analyse.

Les descripteurs permettent de décrire le contenu des documents audiovisuels stockés

dans le système. Il s'agit de représentations intermédiaires des données multimédias, comme par exemple les valeurs obtenues après l'application d'une transformée de Fourrier à un enregistrement sonore. Il existe un nombre important de représentations différentes, chacune dépendant du type d'information que l'utilisateur souhaite extraire des données collectées. Les systèmes de stockage multimédia fournissent une bibliothèque d'algorithmes divers permettant de procéder à l'extraction de plusieurs types de représentations.

À partir de ces descripteurs, l'information obtenue peut être traitée de plusieurs manières : elle peut être utilisée pour construire un index des données afin de faciliter la recherche de contenu dans la banque de données multimédia (*indexation*), ou bien les données peuvent être traitées pour en extraire des informations précises, à travers l'utilisation d'algorithmes de détection d'objets, d'artefacts ou de mouvements (*détection*).

Les composants génériques décrits dans cette section sont présents dans la plupart des systèmes évoqués dans la suite de ce chapitre. Chaque système doit cependant relever un certain nombre de défis pour parvenir à indexer ou extraire des informations des contenus.

2.2.1 Stockage de données multimédia

Les données multimédia sont dans la plupart des cas des données extrêmement riches en informations et donc très volumineuses. Les données ainsi stockées doivent aussi faire l'objet de traitements et pré-traitements, dont il faut stocker les résultats pour pouvoir les analyser à l'aide d'outils appropriés. Les volumes de stockage atteints deviennent très vite handicapant pour les systèmes de gestion de base de données traditionnels. Différentes stratégies de stockage distribué peuvent être utilisées pour résoudre ce problème. Les données multimédias recouvrent généralement des espaces disque importants et sont difficilement compressibles sans perte d'information. Par conséquent, la distribution des données sur plusieurs machines à la fois apparaît comme la solution simple et économique. Plusieurs stratégies coexistent dans le domaine pour parvenir à ce but : les données peuvent être stockées dans un système de fichiers distribué [SKRC10], dans une base de données distribuée [Kos05] ou encore dans une base de données distribuée non relationnelle [Cor07, CDG+06].

Système de fichiers distribués Un système de fichiers distribué est un système dans lequel l'espace disque de plusieurs machines est mis en commun et vu par toutes les machines du système comme une seule entité de manière transparente. Cela permet d'augmenter l'espace disponible en partageant la capacité des disques de chaque machine du système, ou d'augmenter la fiabilité du système en autorisant la réplication des données. Des exemples de systèmes de fichiers distribués sont le Network File System (NFS) [SCR+00], XTreemFS[HCK+07] ou encore le Hadoop Distributed File System [SKRC10].

Cette approche est utilisée dans de nombreux projets de stockage et de traitement de données multimédia. Par exemple, les projets RanKloud [Can11] et ImageTerrier [HSD11] reposent sur le système de fichiers distribué issu du projet Hadoop, le Hadoop Distributed File System [SKRC10]. Ce système de fichiers a la particularité de ne permettre que l'ajout et la suppression de fichiers. La modification des données n'est pas possible. Cette limitation ne pose pas de problème dans les systèmes de stockage et traitement de données multimédia, car les fichiers bruts ne font pas l'objet de modifications internes, mais sont copiés puis lus plusieurs fois de suite.

Ce type de système est particulièrement adapté lorsque les données doivent être traitées en parallèle sur plusieurs machines par divers programmes, comme dans le système de traitement de données parallèle Hadoop [Whi09].

Stockage en base de données La solution la plus directe pour fournir l'accès à des données dans une plateforme de services est la mise au point d'une base de données relationnelle. D'après [Kos05] l'arrivée dans les années 90 des systèmes de bases de données orientés objets tels qu'Informix [1] ou Oracle [2] a permis l'émergence de systèmes plus efficaces pour représenter les données multimédia. Les standards MPEG-7 [MPE12b] et MPEG-21 [Mpe12a], par exemple, ont fourni un cadre de représentation des données multimédia pouvant servir de référence pour la mise au point de schémas de base de données spécifiques. C'est cette approche qui a été retenue dans le projet de "base de donnée MPEG-7", développée par M. Döller et décrite dans [Döl04]. En utilisant un système de gestion de bases de données objet, M. Döller a créé un nouveau schéma respectant le format de re-

1. `http://www.ibm.com/software/data/informix/`, consulté le 5 octobre 2013
2. `http://www.oracle.com/us/products/database/overview/index.html`, consulté le 5 octobre 2013

présentation fourni par le standard MPEG-7. Afin de parvenir à utiliser le langage SQL pour rechercher les données, les méthodes d'indexation ont aussi été étendues pour pouvoir indexer des données multi-dimensionnelles. Ceci permet aux utilisateurs de bénéficier de tous les mécanismes des bases de données traditionnelles et notamment d'utiliser le langage SQL pour la manipulation et la recherche de données multimédia. Les fichiers sont stockés soit en tant qu'objets binaires (BLOBs), comme simples champs d'une table, soit dans le système de fichiers, et seul le chemin vers les données figure dans la base.

La principale limite de cette approche réside dans le fait que le système soit implémenté comme une extension du système commercial de bases de données Oracle. Il est donc difficilement envisageable d'ajouter de nouvelles fonctionnalités au système, en raison de l'interface de programmation de bas niveau nécessaire à ces changements. L'autre limite de cette approche est que le système n'est pas distribué, or, le volume de données généralement associé avec les bases de données multimédia peut nécessiter un fonctionnement réparti afin de multiplier la capacité de stockage et de traitement, ce qui n'est pas le cas dans ces travaux.

Bases de données distribuées : Les bases de données distribuées permettent d'agréger la capacité de stockage de plusieurs machines au sein d'un seul système de base de données. Retenue par K. Chatterjee et al. [CSC09], cette approche consiste à utiliser les capacités existantes de systèmes de bases de données distribuées, comme par exemple MySql Cluster [3], et à adapter la base de données en ajoutant un index réparti permettant de retrouver rapidement les données recherchées sur l'ensemble des machines.

IrisNet [GBKYKS03] est un réseau de capteurs pair à pair sur lequel les images et données des différents capteurs sont stockées dans une base de données XML. Les nœuds sont divisés en deux catégories : les "Sensing Agents" (SA) qui sont des machines de puissance moyenne reliées aux capteurs (webcam, météo, ...) et les "Organizing Agents" (OA) qui sont des serveurs sur lesquels sont déployés les services du réseau de capteurs. Les capteurs eux-mêmes sont des machines de puissance moyenne, avec peu ou pas de contraintes environnementales, et un espace de stockage important.

3. `http://www.mysql.com`, consulté le 5 octobre 2013

Ces nœuds appelés “Sensing Agents” sont utilisés pour déployer des “Senselets”, qui filtrent les données issues des capteurs. La base de données est distribuée sur les OA. Les utilisateurs peuvent requêter la base de données en utilisant le langage XPATH (langage de requêtage de données XML).

Ces deux approches reposent sur les systèmes de base de données traditionnels fournissant de fortes garanties de cohérence des données, suivant le principe ACID (Atomicité, Cohérence, Isolation, Durabilité). Or, il a été démontré que lorsque la taille d'un système distribué augmente, fournir une garantie sur la cohérence des données du système ne permet pas d'assurer à la fois disponibilité et la performance de celui-ci. C'est le théorème de CAP, Consistency, Availability et Partition Tolerance (Cohérence, Disponibilité et Tolérance aux partitions) démontré par Gilbert et Lynch dans [GL02]. Les systèmes multimédias distribués ne nécessitent pas de supporter de fortes garanties de cohérence sur les données : les fichiers à traiter ne sont écrits qu'une seule fois puis lus à plusieurs reprises pour mener à bien les analyses voulues. Par conséquent, certains acteurs ont construit leur propre système de gestion de données multimédia en s'abstrayant de ces garanties, comme Facebook [4] l'a fait avec le système Haystack [Vaj09]. Les composants de ce système ne reposent pas sur une base de données relationnelle mais sur un index distribué permettant de stocker les images dans des fichiers de grandes tailles (100Go), ce qui permet de grouper les requêtes et de réduire les accès disques nécessaires à l'affichage de ces images. La particularité de ce système est que les images y sont peu recherchées, mais sont annotées par les utilisateurs pour y ajouter des informations sur les personnes présentes dans l'image et la localisation de la prise de vue.

Un autre exemple de base de données distribuée utilisée pour stocker les données multimédia est celui de Youtube [5], qui utilise la base de données orientée colonnes BigTable [CDG+06, Cor07], pour stocker et indexer les aperçus de ses vidéos.

Agrégation de bases de données : Le canevas AIR créé par F. Stiegmaier et al. [SDK+11] se concentre sur l'extensibilité en utilisant un intergiciel pour connecter plusieurs bases de données entre elles, et utilise des requêtes au format MPEG-7 [MPE12b].

4. `http://facebook.com`, consulté le 5 octobre 2013
5. `http://youtube.com`, consulté le 5 octobre 2013

Cependant ces travaux restent dirigés sur la recherche de contenus plutôt que sur la détection d'éléments précis dans les images. Les données et leurs métadonnées (descripteurs) sont réparties sur différents systèmes de gestion de bases de données (SGBD) pouvant fonctionner sur différentes machines. Lorsque le système reçoit une requête, deux cas se présentent : soit les nœuds sont homogènes et la requête est envoyée à chaque machine puis les résultats sont fusionnés et / ou concaténés, soit les nœuds sont des systèmes différents (bases de données différentes) et seule la partie la plus adaptée de la requête est transmise au noeud le plus approprié. Les intergiciels LINDO[BCMS11] et WebLab[GBB+08] fournissent des modèles de représentations de données communs à plusieurs systèmes et permettent les transferts de données entre des services de traitements adoptant ce modèle. Ces systèmes ne reposent sur aucun système de stockage particulier mais permettent d'agréger différentes sources de données et différents services de traitements hétérogènes. La problématique de cette approche est la nécessité de devoir adapter les services et les traitements au modèle exposé par les interfaces d'un tel système.

Une grande variété de solutions existent au problème du stockage des données multimédia. Un critère de choix possible est le type d'utilisation qui sera fait des données : par exemple un système d'extraction des connaissances fonctionnera mieux avec un système de fichiers distribué dans lequel les données sont accessibles de manière transparente, au contraire d'une base de donnée SQL qui requiert une interface particulière pour accéder aux données stockées. Dans le cadre du projet MCube l'accès aux données doit être possible de plusieurs manières : il faut pouvoir fournir un accès permettant de traiter les données en parallèle ainsi qu'une interface permettant de requêter les données générées par les résultats des analyses effectuées. Le système se rapprochant le plus de ce cas d'utilisation est donc l'utilitaire HADOOP [Whi09], qui fournit à la fois un système de haut niveau pour requêter les données et un accès de bas niveau via le système de fichiers distribué.

2.2.2 Distribution des traitements

Un volume important de données à analyser se traduit par la nécessité de paralléliser le traitement des données, de manière à réduire les temps d'exécution. Cependant, tous

les systèmes de stockage de données multimédia ne permettent pas de traitement parallèle des données. Dans le domaine des traitements de données multimédia il est possible de distinguer deux manières de traiter les données[Can11] : les traitements à la chaîne (Multiple Instruction Multiple Data, MIMD) et les traitements parallèles (Single Instruction Multiple Data, SIMD).

Traitements à la chaîne : Multiple Instruction Multiple Data Le système est constitué de plusieurs unités, chacune exécutant une étape particulière d'une chaine de traitements. Les données passent d'une étape à l'autre en étant transférées d'une entité à l'autre, de sorte que plusieurs données sont à différents stades de traitement au même instant.

Le projet WebLab [GBB+08] est une architecture de traitement de données multimédia orientée service qui permet de faciliter le développement d'applications distribuées de traitement multimédia. Cette plateforme repose sur un modèle de données commun à tous ses composants représentant une ressource multimédia. Ce modèle ne repose pas sur un standard mais permet de créer des ressources conformes au standard MPEG-7. La raison de ce choix est que le standard MPEG-7 a été créé en 2002 et ne permet pas de bénéficier des dernières avancées en matière de descripteurs de données multimédias. Cette représentation générique permet d'associer à chaque ressource des "annotations" représentant le contenu extrait de la ressource (appelés "morceaux de connaissance"). Les services disponibles se divisent en services d'acquisition, services de traitements, services de diffusion. Le modèle de données commun permet à ces différents services de pouvoir être composés pour créer des flux de traitements, de l'acquisition à la diffusion du contenu multimédia. Cette solution se veut très flexible et générique mais ne propose pas de solution en ce qui concerne la répartition des traitements.

Le système SAPIR (Search in Audio-visual content using Peer-to-peer Information Retrieval) [KMGS09] est un moteur de recherche multimédia implémentant la recherche par l'exemple : il permet de retrouver à partir d'un document multimédia donné les documents similaires ou correspondant au même objet. SAPIR utilise le Framework apache UIMA [UIM12] pour extraire les informations nécessaires à l'indexation des données au

format MPEG-7. Les flux ou fichiers reçus sont divisés en plusieurs modalités : par exemple dans le cas d'une vidéo, un "splitter" va séparer le son des images, puis la suite d'images va aussi être divisée en plusieurs images fixes, chaque élément ainsi extrait va ensuite être traité par un composant spécifique pour le son, la vidéo et les images le tout de façon parallèle. Une fois la phase d'extraction terminée le fichier d'origine est recomposé en intégrant les métadonnées qui en ont été extraites. UIMA rend la main au système SAPIR, qui va insérer le fichier dans son index distribué en se basant sur les métadonnées que le fichier contient désormais. Le système utilise une architecture pair à pair pour stocker et répartir les données.

Cependant, la dépendance du système SAPIR avec Apache UIMA, destine ce système uniquement à la recherche de contenus multimédia plutôt qu'à servir de plateforme de traitement de données multimédia générique.

Traitement en parallèle : Single Instruction Multiple Data (SIMD) Ce type de système sépare les étapes du traitement dans le temps plutôt que dans l'espace : à chaque étape du traitement, toutes les machines du système exécutent le même traitement en parallèle sur l'ensemble des données. C'est le modèle suivi par le framework Hadoop [Whi09] pour traiter les données. Ce framework générique de traitement de données massives en parallèle est utilisé dans plusieurs systèmes de traitements de données multimédia. Par exemple, les travaux menés dans le cadre du projet RanKloud [Can11] utilisent et étendent le framework HADOOP [Whi09] pour répondre à des requêtes de classement, ainsi que pour permettre des jointures dans les requêtes portant sur plusieurs ensembles de données. Pour cela le système échantillonne les données et évalue statistiquement leur répartition pour optimiser certaines procédures (notamment les opérations de jointures). Un index des données locales est construit sur chaque machine du système, permettant à chaque nœud de résoudre les requêtes localement et d'envoyer les résultats à une seconde catégorie de nœuds qui assureront la finalisation des requêtes. Cependant ce système est conçu pour analyser une grande quantité de données et non pas pour répondre à des requêtes en temps réel, et se destine plutôt à des requêtes de recherche de contenus.

Le projet Image Terrier [HSD11] est un système d'indexation et de recherche d'images

et de vidéos fondé sur le système de recherche textuelle Terrier [6]. Ce système utilise lui aussi la technique de distribution des traitements fournie par le canevas HADOOP afin de construire incrémentalement l'index des images. Ce système présente l'avantage d'être très ouvert, mais il est uniquement destiné à la recherche de contenus.

Limites des outils existants Ces deux types de répartitions ne sont pas exclusives et les deux approches peuvent être combinées de manière à augmenter l'efficacité d'un système. Il est ainsi possible de créer des flux de travaux enchaînant différentes étapes de traitement ou les exécutant en parallèle. La principale limite des outils existants réside dans le fait que les traitements répartis font partie d'une bibliothèque logicielle utilisant un canevas donné, comme ImageTerrier [HSD11] et Hadoop. Or, dans le projet MCube, les traitements à répartir sont codés et compilés sans suivre d'API ou de canevas logiciel particulier. Il est aussi parfois nécessaire d'exécuter les traitements à la demande au fur et à mesure de l'arrivée des données. Dans ce contexte il apparait donc nécessaire de faire appel à des plateformes de plus haut niveau permettant d'adapter les traitements à leur contexte d'exécution.

2.2.3 Plateformes génériques de traitements de données multimédias

La recherche en matière de traitement des données multimédia se concentre aussi pour une large part sur la conception de nouveau algorithmes d'extraction d'informations. Le système doit fournir un catalogue de services permettant de procéder aux extractions de données et aux analyses voulues par l'utilisateur. Cette librairie doit être extensible afin de permettre aux utilisateurs d'y implanter leurs propres analyses. L'ouverture d'un système se mesure à sa disponibilité, et à sa capacité à s'interfacer avec d'autres systèmes de manière à pouvoir être utilisé dans différents contextes. Plusieurs systèmes de stockage évoqués dans ce chapitre sont décrits dans la littérature mais ne sont pas disponibles pour évaluation, du fait du peu de cas d'utilisations qu'ils permettent de résoudre. Par exemple le système Haystack développé par Facebook ne fait pas l'objet d'un développement à l'extérieur de l'entreprise. Le projet RanKloud bien que décrit dans diverses publications, n'est pas

6. `http://terrier.org/`, consulté le 5 octobre 2013

publiquement disponible. De même, peu de projets sortent du simple cadre du prototype de recherche, comme par exemple le projet LINDO [BCMS11].

Le projet LINDO [BCMS11] a pour objectif de développer une infrastructure générique de traitement de données multimédia permettant de distribuer les traitements et les données, en assurant l'interopérabilité de différents systèmes de traitement de données multimédias. Il a permis la mise au point d'une infrastructure d'indexation générique et distribuée. Pour cela, un modèle de données représentatif a été créé pour les données multimédia et pour les algorithmes d'indexation de ces données. Les données sont donc réparties sur un système de fichiers distribué et le système sélectionne l'algorithme d'indexation le plus approprié en fonction des requêtes les plus féquemment envoyées par l'utilisateur. Ce système a été appliqué à la vidéosurveillance, pour détecter des événements dans les flux vidéo. C'est une plateforme multi-utilisateurs, qui peut être utilisée dans plusieurs domaines à la fois, et permet de représenter les données au format MPEG-7 ou MPEG-21. Cependant les informations ne sont extraites qu'à la demande explicite des utilisateurs (à travers une requête) et traversent toutes un serveur central qui contrôle les machines indexant les contenus. Ce système risque donc de limiter l'élasticité de la solution. L'élasticité d'un système est sa capacité à s'adapter à l'apparition ou la disparition dynamique de nouvelles ressources. Cette notion est particulièrement importante dans le contexte des plateformes de Cloud Computing, où les ressources peuvent être allouées et disparaître rapidement.

Le projet WebLab est un projet libre [7], il est construit sur le même principe d'interface et de modèle d'échange de données que le projet LINDO. Le projet WebLab ne modélise pas les traitements, mais seulement le format des données à échanger entre différentes plateformes de traitements hétérogènes. Ce système repose sur le bus de service Petals [8] pour ses communications, or ce composant semble posséder des problèmes de performance comme le montre les résultats de différents bancs d'essais successifs [9].

Ces deux systèmes servent donc d'intergiciels pour la communication entre différentes plateformes de traitements et permettent l'agrégation de différents sites ou grilles spécialisés

7. `http://weblab-project.org/`, consulté le 5 octobre 2013
8. `http://petals.ow2.org/`, consulté le 5 octobre 2013
9. `http://esbperformance.org/display/comparison/ESB+Performance+Testing+-+Round+6`, consulté le 5 octobre 2013

dans certaines étapes de traitement.

Le propos du projet MCube est au contraire de concentrer les données des différents utilisateurs afin de mutualiser la plateforme de traitement et permettre des économies d'échelles en matière de stockage et de capacité de calcul. Ce projet se situe à un niveau intermédiaire entre les systèmes de traitements de données de bas niveau (ImageTerrier, Hadoop, ...) et ces plateformes de fédération que sont les projets LINDO et WebLab. Il s'agit donc d'un système multi-tenant, car il sert plusieurs clients en mutualisant ses ressources.

2.3 Analyse

Les systèmes de stockage et de traitements multimédias décrits ici possèdent diverses architectures. Les architectures fondées sur les bases de données traditionnelles laissent peu à peu place à des systèmes basés sur les technologies issues des plateformes Web à large échelle telles que le canevas Hadoop [Whi09]. Bien que l'adoption de ces solutions se démocratise, il n'existe pas pour le moment d'alternative permettant de tirer parti des différents services de ces plateformes sans nécessiter de développement supplémentaire pour être adapté à l'environnement cible. De plus, hormis les initiatives telles que LINDO, Weblab ou ImageTerrier, peu de projets dépassent le stade du prototype de recherche. Pour une large part, ces projets servent de plateformes d'évaluation des algorithmes d'extraction de connaissances et d'indexation, ou sont orientés vers la recherche de contenus. Le tableau 2.1 résume les propriétés disponibles pour chacun des systèmes listés dans les sections précédentes, par rapport aux défis décrits dans la section 2.1.2 de ce chapitre.

L'examen des différentes plateformes de stockage et de traitement de données multimédias montre qu'il n'existe pas, actuellement, de projets remplissant le cahier des charges du projet MCube. Il existe des plateformes de traitements de données génériques, qui ne sont pas spécialisées dans les données multimédia, comme le canevas Hadoop, et qui peuvent donc servir de base à la plateforme MCube.

TABLE 2.1 – Propriétés des systèmes de traitements de données multimédias

	Stockage			Traitements			Mutualisation	ouverture	Elasticité
Système	Données	Descripteurs	Informations	Descripteurs	Indexation	Informations			
Mpeg-7 DB	oui	oui	non	oui	oui	non	oui	non	non
AIR	distribué	distribué	non	oui	distribué	non	oui	non	limitée
Lindo	distribué	distribué	non	distribué	distribué	oui	oui	limitée	limitée
Weblab	distribué	distribué	non	non	non	non	oui	limitée	limitée
SAPIR	distribué	distribué	non	oui	oui	non	oui	limité	oui
IrisNet	distribué	distribué	distribués	oui	distribué	oui	oui	oui	oui
[CSC09]	distribué	distribué	non	oui	distribué	non	oui	non	oui
RanKloud	distribué	distribué	non	distribué	distribué	non	oui	non	oui
Haystack	distribué	distribué	distribués	distribué	distribué	oui	non	non	oui
ImageTerrier	distribué	distribué	non	distribué	distribué	non	non	oui	oui

2.4 Architecture de la plateforme MCube

2.4.1 Architecture matérielle

Passerelles MCube. Les passerelles MCube sont des appareils mis au point par la société Webdyn [10]. Il s'agit d'un système embarqué reposant sur le système Linux et un processeur ARM. La présence du système d'exploitation complet permet de connecter divers périphériques d'acquisition de données multimédias comme des appareils photos ou des micros. Le système de la passerelle exécute un planificateur de tâches pouvant être configuré à distance en recevant des commandes depuis les serveurs MCube. Le rôle des passerelles est d'assurer l'exécution à intervalles réguliers (configurés par l'utilisateur) d'un programme appelé "librairie passerelle" actuellement déployé sur le système. Les passerelles sont équipées d'un port Ethernet et d'un modem 3G permettant de se connecter à internet pour dialoguer avec le serveur MCube.

Serveurs MCube. Les serveurs MCube sont trois machines Dell équipées chacune de deux processeurs intel Xeon embarquant 8 cœurs chacun. L'espace de stockage important sur les machines (8 téraoctets par serveur) permet d'anticiper sur la quantité de données à stocker dans le cadre du projet MCube, dont les collectes annuelles peuvent représenter dans certains cas plusieurs centaines de gigaoctets par mois.

10. `http://www.webdyn.com`, consulté le 5 octobre 2013

2.4.2 Architecture logicielle

L'architecture logicielle du projet MCube a été conçue de manière à permettre aux utilisateurs de pouvoir eux-mêmes développer les applications déployées sur les passerelles et la plateforme. Pour cela, une interface de programmation a été mise au point par le constructeur de la passerelle, la société WebDyn [11], permettant de séparer les tâches d'acquisition des données de leur planification. Les composants développés sur cette interface sont appelés "librairies passerelles". Les librairies de passerelles sont des librairies dynamiques implantant les fonctions de l'interface de programmation MCube, ce qui permet de les déployer à la demande depuis la plateforme Web du projet. Ces librairies sont développées pour remplir deux fonctions principales :

- *L'acquisition des données* : C'est la fonction essentielle de ces librairies, elle consiste à piloter les différents périphériques de capture de données multimédia connectés à la passerelle pour acquérir des données brutes : photos, sons, vidéos.
- *Filtrage des données* : Dans certains cas, les données capturées peuvent être volumineuses. Par exemple, un cas d'utilisation possible du service MCube consiste à détecter la présence d'insectes nuisibles à partir d'enregistrements sonores. Dans ce cas précis les données enregistrées ne sont envoyées au serveur que lorsqu'un signal suspect est détecté dans l'enregistrement. Un traitement plus précis exécuté à la demande sur le serveur permet ensuite de confirmer ou d'infirmer la détection.

La communication entre les passerelles et les serveurs repose sur un protocole HTTP/ REST qui permet aux passerelles :

- d'envoyer des données collectées à la plateforme MCube via le serveur Web,
- de recevoir une nouvelle librairie,
- de recevoir des commandes à transmettre à la librairie déployée sur la passerelle.

2.4.3 Architecture de la plateforme MCube

La plateforme MCube a un rôle de gestion des équipements et d'agrégation d'information. Elle assure la communication avec les passerelles déployées, en permettant aux utilisateurs de les configurer à l'aide d'une interface web. La plateforme fournit aux utilisa-

11. `http://www.webdyn.com/`, consulté le 5 octobre 2013

teurs l'accès aux librairies et programmes d'analyse de données qu'ils ont développés. Les données transférées par les passerelles sont stockées sur les serveurs de la plateforme, et une interface web permet aux utilisateurs de lancer ou de programmer les analyses qu'ils souhaitent exécuter sur les données téléchargées depuis les passerelles. L'architecture finale de la plateforme MCube repose sur trois composants logiciels :

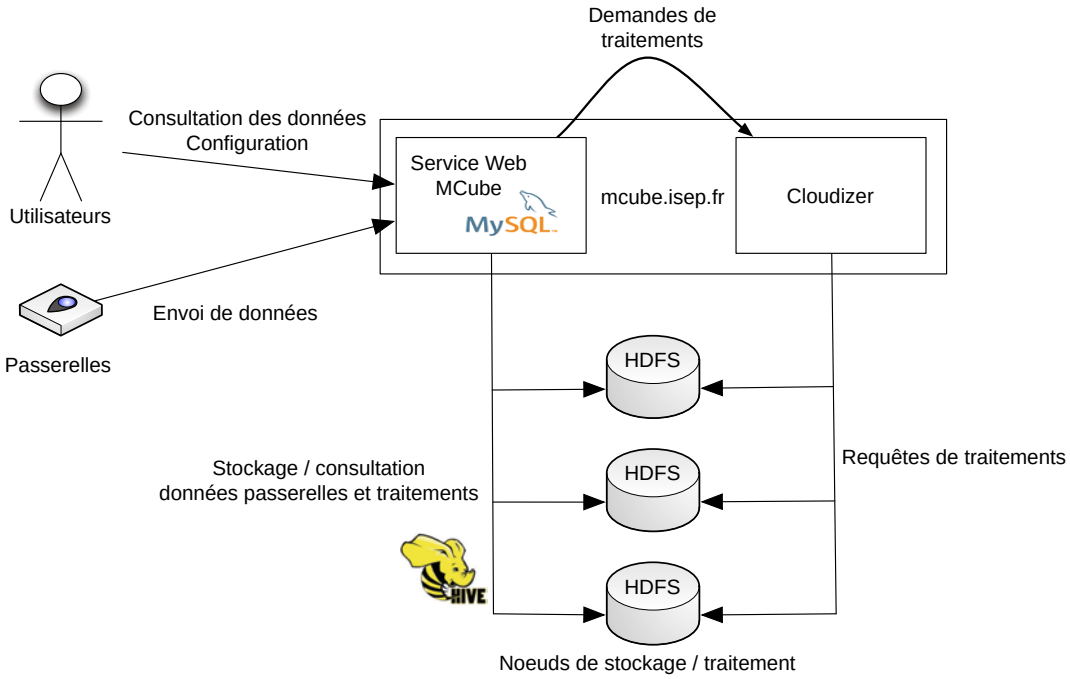


FIGURE 2.3 – Architecture de la plateforme MCube

L'interface Web C'est l'interface utilisateur du système. Elle permet aux agriculteurs de gérer leurs passerelles et d'accéder à leurs données afin de les récupérer dans le format qu'ils souhaitent ou de lancer / planifier les traitements à effectuer. De plus l'interface permet aux utilisateurs de charger de nouveaux programmes de traitements de données.

Le canevas Hadoop Le système Hadoop permet de stocker les données de manière répartie via le système de fichiers distribué HDFS [SKRC10]. Il permet d'avoir un système de fichiers redondant pouvant accueillir un grand nombre de machines. De plus, les données ainsi stockées peuvent être accédées à l'aide de langages de plus haut niveau tel que le langage de requêtage HQL (Hive Query Language) [TSJ+09]

Le canevas Cloudizer Le framework Cloudizer est un canevas logiciel développé durant cette thèse qui permet de distribuer des requêtes de services web à un ensemble de machines. Ce système est utilisé pour traiter les fichiers envoyés par les passerelles au fur et à mesure de leur arrivée dans le système.

La figure 2.3 montre comment s'articulent les différents composants de la plateforme. Les données de configuration, les informations utilisateurs, et les coordonnées des passerelles sont stockées dans une base de données SQL traditionnelle (MySql [12]). Les données remontées par les passerelles, ainsi que les librairies et les programmes d'analyses sont stockés dans le système HDFS [SKRC10].

Chaque utilisateur possède un répertoire dédié sur le système de fichiers distribué. Ce répertoire contient les fichiers reçus des passerelles et classés par nom de la librairie d'origine des données. Les résultats des traitements effectués sur ces données sont stockés dans un répertoire dédié et classés par noms de traitements et date d'exécution. Ceci permet aux utilisateurs de pouvoir retrouver leurs données de manière intuitive.

Les utilisateurs peuvent choisir deux modes de traitement des données : un mode événementiel et un mode planifié. Lorsque le mode événementiel est sélectionné, l'utilisateur définit le type d'événement déclenchant le traitement voulu : ce peut par exemple être l'arrivée d'un nouveau fichier ou le dépassement d'un certain volume de stockage. Le traitement est alors exécuté sur les données désignées par l'utilisateur. Le mode planifié correspond à une exécution du traitement régulière et à heure fixe. Le déclenchement des traitements est géré par la plateforme Cloudizer décrite dans le chapitre suivant.

2.4.4 Description des algorithmes de traitement de données multimédia

Les utilisateurs de la plateforme peuvent charger de nouveaux algorithmes de traitement de données multimédia. Pour que le système puisse utiliser cet algorithme, il est nécessaire d'en fournir une description afin de le stocker dans la base de données. Le terme "algorithme" désigne ici par abus de langage l'implantation concrète d'un algorithme donné, compilée sous la forme d'un programme exécutable. Un modèle type de ces algorithmes a donc été conçu, il est représenté en figure 2.4. Ce schéma est inspiré des travaux sur

12. `www.mysql.com`, consulté le 5 octobre 2013

l'adaptation de composants de [AGP+08, BCMS11] et [GBB+08].

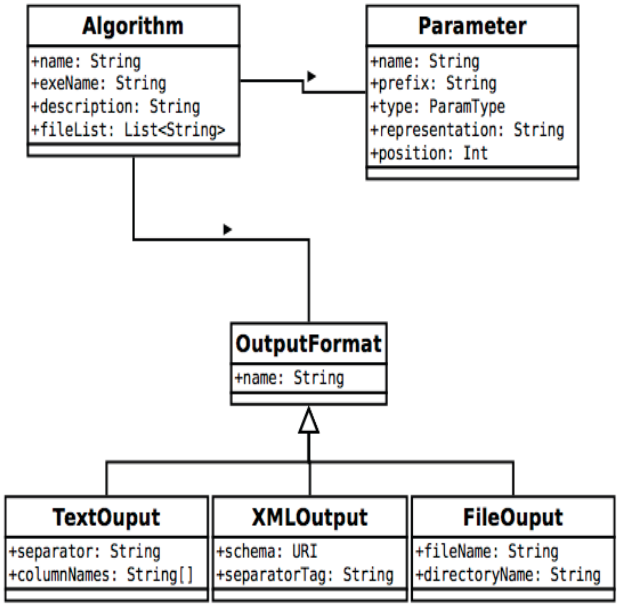


FIGURE 2.4 – Modèle de description d'algorithme

Un programme et l'ensemble de ses fichiers sont décrits par la classe "*Algorithm*". Cette classe contient une liste qui permet de spécifier les fichiers exécutables et de configurations nécessaires à l'exécution du programme. Un programme est généralement exécuté avec des paramètres fournis sur l'entrée standard, la liste des paramètres (la classe "*Parameter*") décrit les paramètres à fournir à l'exécutable ainsi que leurs positions et préfixes respectifs si besoin. La classe "*OutputFormat*" permet de décrire le type de donnée en sortie de l'exécution du programme. Par exemple, la sélection du format "TextOuput" signifie que la sortie du programme sur l'entrée standard correspond à des données textes tabulaires, séparées par un caractère spécifique. D'autres formats sont disponibles et peuvent être ajoutés à la librairie.

Exemple de modèle d'algorithme : Calcul de tailles de tomates Le listing 2.1 présente un exemple de modèle d'algorithme pour un programme développé dans le cadre du projet MCube. Dans ce cas d'utilisation, un dispositif expérimental a été mis au point par les membres de l'équipe de traitement du signal de l'ISEP, afin de prendre en photo un

plant de tomates selon deux angles différents. Grâce à ce décalage et un calibrage précis du dispositif, il est possible de donner une estimation de la taille des tomates détectées dans les photos, si leurs positions sont connues. Cet algorithme prend en entrée deux listes de points correspondants respectivement aux contours droit et gauche d'une tomate du plant photographié, et fournit une estimation de la taille du fruit.

Listing 2.1 – Exemple d'instance du modèle d'algorithme

```
<?xml version="1.0" encoding="UTF-8"?>
<algorithm>

        <name>tomate_measurement</name>
        <version>1.0</version>
        <author>Ujjwal Verma</author>

        <description>
                Detection de visages
                methode de Viola-Jones, librairie OPEN-CV
         </description>
         <mediatype>txt</mediatype>
         <files>
                 <exename>tomate_measurement.jar</exename>
         </files>

         <parameters>
                 <parameter pos="1" type="FileParameter">
                         leftImageContours
                 </parameter>
                <parameter pos="2" type="FileParameter">
                         rightImageContours
                </parameter>
         </parameters>
         <output type="TextOutputFormat"
                                 name="tomate_size"
                                 separator="">
            <column num="0" name="size"/>
        </output>
</algorithm>
```

Cet algorithme est implémenté en java sous la forme d'une archive JAR. Cependant le système le voit comme une boite noire et ne connait donc que les paramètres à lui fournir en entrée et le type de données en sortie. Cette information sur les paramètres à fournir permet de pouvoir commander l'exécution du programme à la demande, via un service

Web utilisant le canevas Cloudizer décrit au chapitre 3.

2.5 Développement du projet

La conception détaillée de la plateforme Web MCube et son modèle de données a été faite par l'auteur de ce document. Le développement de l'interface Web permettant la gestion des passerelles et le stockage des librairies ont été fait par trois stagiaires de niveau Master 1 et 2. Le mode de développement s'est fondé sur des itérations courtes (une à deux semaines), en donnant des objectifs de fonctionnalités précises à développer durant ce laps de temps, via une feuille de route établie à l'avance. Chaque nouvelle fonctionnalité était testée manuellement puis déployée sur le serveur de production du projet [13]. Cette approche de développement d'application rapide a été facilitée par l'utilisation du canevas Grails comme base au projet. L'interface web de la plateforme représente un volume de 4000 lignes de code en langage Groovy. Mon rôle au niveau du développement de ce système s'est donc limité à la conception de la base de données, puis à la supervision et aux tests fonctionnels des développements réalisés, ainsi qu'à la correction de quelques bogues éventuels trouvés lors des phases de tests. Le code doit encore faire l'objet d'une intégration avec le projet Cloudizer pour assumer l'ensemble des fonctionnalités attendues. Il sera à terme disponible sous forme de programme libre, sous réserve d'acceptation par les parties prenantes du projet.

2.6 Discussion

Le projet MCube est une architecture logicielle de collecte et de traitement de données multimédias destinée à fournir des services avancés de détection et de surveillance aux agriculteurs. La mise au point de ce système nécessite de faire interagir différents composants logiciels hétérogènes. La problématique de la distribution et de l'exécution en parallèle de traitements de nature spécifique, a des conséquences sur le choix de la plateforme utilisée pour cette répartition et sur son efficacité. L'architecture finale s'appuie donc sur trois services : un serveur d'applications destiné à la communication avec les utilisateurs du

13. `http://mcube.isep.fr`, consulté le 5 octobre 2013

système et les services extérieurs, un système de fichiers distribués permettant de stocker différents fichiers tout en permettant différents modes d'accès aux données stockées en son sein, et un système de gestion de services et de distribution de requêtes qui assurera la répartition de la charge lors du traitement des données.

Ce service, appelé Cloudizer et développé au cours de cette thèse, permet à l'utilisateur de choisir différentes stratégies de répartition de charge, et de comparer leur performances respectives pour un même service. La mise au point de stratégies de répartition adéquates est donc nécessaire. Le chapitre suivant décrit l'architecture technique du projet Cloudizer et son intégration au projet MCube.

Chapitre 3

Le framework Cloudizer : Distribution de services REST

3.1 Introduction

Cloudizer est une plateforme logicielle ouverte permettant de déployer, répartir et superviser des services web REST. Cette plate-forme est destinée à faciliter le déploiement et la répartition d'applications existantes sur les plateformes d'informatique dans les nuages de type "infrastructure à la demande". Cloudizer a été développé avec deux objectifs : premièrement faciliter le développement et le déploiement de stratégies de répartition de charge adaptées aux applications, en fournissant une interface de programmation suffisamment expressive et claire pour le programmeur. Deuxièmement la plateforme doit assurer la disponibilité de plusieurs services différents en permettant de choisir la stratégie de répartition de charge la plus appropriée pour chacun de ces services.

Pour assurer cette répartition, le problème de conservation de l'état du service se pose. Lorsqu'un service web maintient des informations sur chacun des clients qui lui sont connectés par exemple, il est nécessaire de toujours renvoyer les requêtes de ce client vers la machine qui possède les informations le concernant, ce qui réduit considérablement les possibilités de répartition de la charge. Ce constat a été établi par Roy T. Fielding dans sa thèse publiée en 2000 [Fie00], où il propose d'organiser les services client-serveur autour du principe "Representational State Transfer" (REST). Selon ce principe, les séquences de requêtes et de réponses des systèmes clients serveurs sont considérées comme une suite

d'échanges d'objets appelés "ressources". A chaque échange, une représentation particulière de la ressource considérée est transmise. Le client transmet au serveur les informations nécessaires pour retrouver la ressource associée à chaque requête, de sorte que le serveur n'ait pas à maintenir d'information sur la séquence d'échange en cours. Le serveur ne maintient donc pas d'état entre les requêtes, ce qui simplifie sa distribution car il n'y a pas d'état à synchroniser entre les machines.

Le système Cloudizer permet de gérer et répartir l'exécution de services web sur un ensemble de machines de deux manières : soit en exécutant les requêtes au fil de l'eau, soit en les exécutant en parallèle, par lots de requêtes. Ce comportement est approprié dans le contexte de la plateforme MCube, qui a besoin de ces deux modes de fonctionnement pour gérer les traitements des données multimédia.

La première partie de ce chapitre décrit donc l'architecture et le fonctionnement général de la plateforme Cloudizer 3.2. La seconde partie de ce chapitre est consacrée aux mécanismes permettant d'intégrer Cloudizer dans la plateforme MCube 3.3. La conclusion de ce chapitre (section 3.5) discutera des limites de cette approche et les développements nécessaires pour l'améliorer.

3.2 Architecture du canevas Cloudizer

Cloudizer tire partie de l'architecture multi-tiers de la plupart des applications Web pour permettre la répartition de charge. Comme le montre la figure 3.1, le code du service à distribuer est répliqué sur toutes les machines du système. La gestion des données est déléguée à une base de données répartie, et la gestion des fichiers d'application est déléguée à un système de fichiers distribué. Ce découplage permet à la plateforme de n'assurer que la répartition des traitements. Les composants de cette plateforme communiquent via des services REST. Ces composants sont au nombre de trois, il s'agit du *répartiteur*, du *contrôleur* et de la librairie déployée sur les machines du système, appelées "nœuds". La répartition est assurée par le *répartiteur* qui filtre les requêtes. Lorsqu'une requête d'exécution destinée à un service déployé arrive, elle est envoyée vers les *nœuds Cloudizer* selon la stratégie de répartition sélectionnée par l'utilisateur et appliquée par le *répartiteur*.

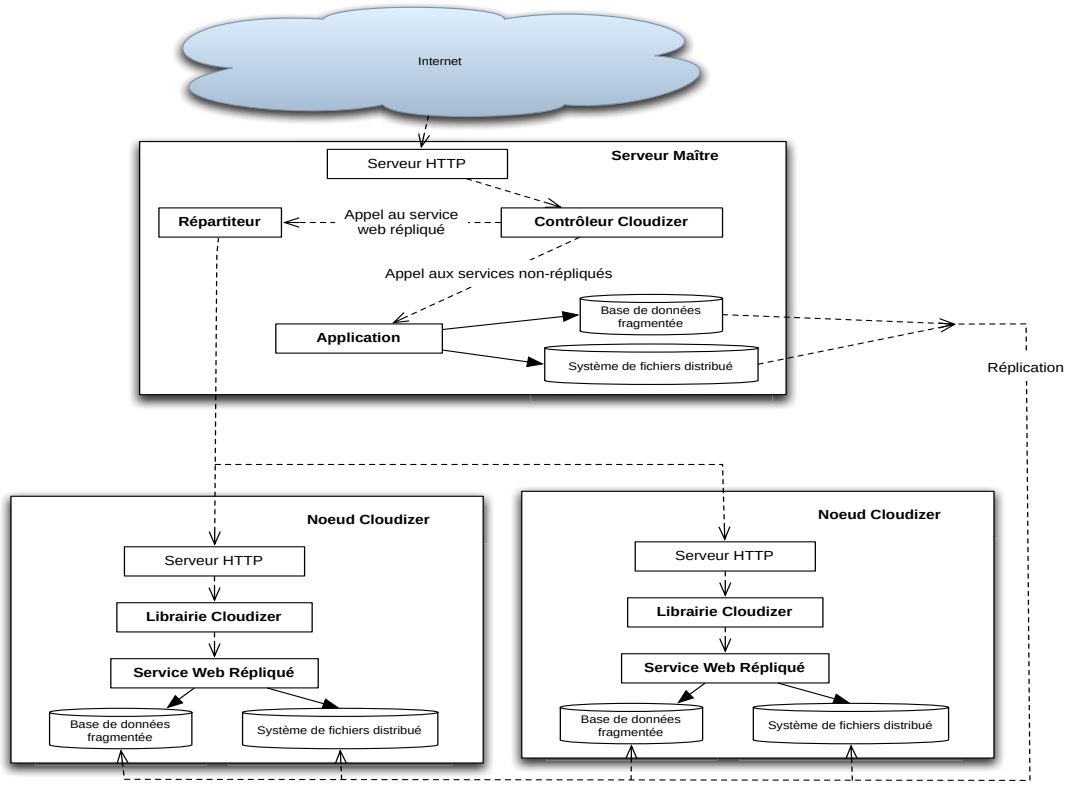


FIGURE 3.1 – Architecture de la plateforme Cloudizer

Dans tous les autres cas, la requête est transmise à l'application déployée sur le *serveur maître*. Le but de ce système est de parvenir à répartir le service de manière transparente. Pour cela le code de l'application est répliqué sur tous les nœuds, mais les fichiers ou les bases de données sont répartis via l'utilisation d'un système de fichiers distribué type "HDFS" ([SKRC10]) ou d'une base de données utilisant la fragmentation horizontale.

3.2.1 Les nœuds

Les nœuds de la plateforme Cloudizer sont des machines sur lesquelles la librairie Cloudizer est déployée. Cette librairie fournit les routines nécessaires à l'enregistrement et au déploiement de nouveaux services sur la machine où elle s'exécute. Le seul paramètre de configuration nécessaire au fonctionnement du programme est l'adresse du *Contrôleur*. Quand un nœud (*nœud Cloudizer*) s'enregistre dans le système, il envoie à une fréquence pré-configurée des "battements de coeur" au *contrôleur* afin que celui-ci puisse connaître

les nœuds actifs. Les nœuds enregistrent également la liste des services web actifs sur la machine. Chaque nœud est donc responsable de la supervision des services de la machine sur laquelle il est déployé et reporte son statut au *contrôleur*. Ce statut contient les informations décrivant la machine telles que le nombre de cœurs, la capacité de stockage, la taille de la mémoire et le taux d'utilisation du processeur.

Les nœuds dialoguent avec le *contrôleur* via une interface de programmation REST. Une seconde interface de programmation est utilisée entre les nœuds pour procéder à des appels de méthodes à distance. Les souches d'appels sont générées dynamiquement par le nœud Cloudizer à partir de l'inspection de l'interface des méthodes cibles par réflexion. Les paramètres effectifs sont automatiquement encodés au format U.R.L (Universal Resource Locator). Les données de résultats de ces invocations sont, quant à elles, converties au format XML et décodées lors de la réception. Les machines peuvent donc servir de mandataires pour appeler des services distribués.

3.2.2 Le répartiteur

Le rôle de ce composant est d'appliquer la politique de répartition de charge spécifiée par l'utilisateur pour transférer les requêtes aux services déployés dans le système. Différentes politiques de répartition de charge peuvent être appliquées en fonction de l'application. Une interface Java permet l'implémentation de nouvelles stratégies de répartition. L'utilisateur précise ensuite le nom de la classe implémentant la politique de répartition voulue dans la configuration du *répartiteur* pour l'appliquer sur le service qu'il souhaite répartir. La figure 3.2 montre le modèle utilisé pour implanter ces stratégies. La classe abstraite Policy fournit la méthode virtuelle *loadBalance* qui renvoie une machine cible après avoir pris en paramètre la liste des machines disponibles et les paramètres de la requête en cours. Lorsqu'une requête concernant un service déployé sur la plateforme arrive au niveau du *répartiteur*, la servlet CloudFrontend est appelée et exécute la méthode *loadbalance* définie pour le service cible.

Les stratégies sont chargées lors de l'initialisation du répartiteur, en fonction de la configuration définie par l'utilisateur. Lorsque le répartiteur reçoit une nouvelle requête, il vérifie que la requête vise un service pour lequel des machines sont disponibles en interrogeant le

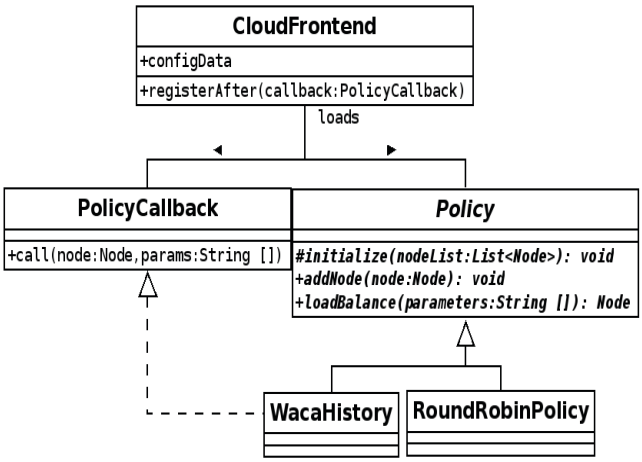


FIGURE 3.2 – Modèle des stratégies de répartition de charge

contrôleur. Si aucune machine n'est disponible, une erreur est renvoyée. Si des machines sont disponibles pour le service demandé, le répartiteur peut procéder au transfert de la requête de deux manières différentes : à la volée, c'est à dire que chaque requête reçue est immédiatement transférée à la machine implantant le service, soit en parallèle : la requête et ses paramètres sont découpés de manière à former un ensemble de plusieurs requêtes distinctes, qui sont envoyées en même temps et de manière asynchrone à un ensemble de machines pour être exécutées en parallèle.

Exécution requête par requête Pour chaque requête reçue, le répartiteur applique la stratégie de répartition sélectionnée par l'utilisateur pour choisir une machine cible, puis transfère directement la requête à la machine sélectionnée.

Exécution des requêtes en parallèle Un service peut être appelé de plusieurs machines / clients à la fois. Toutefois, certains services peuvent être appelés en parallèle sur plusieurs machines en cas de besoin, ce qui permet de réduire le temps d'exécution total. Le résultat sera donc l'agrégation de plusieurs résultats produits sur différentes machines. Pour cela deux modes d'appels sont possibles : un mode d'appel parallèle simple où chaque nœud disponible reçoit une part égale de requêtes à exécuter pour le service appelé, et un mode avec équilibrage de charge où les requêtes d'exécution à agréger sont réparties en

suivant une politique de distribution choisie par l'utilisateur.

3.2.3 Déploiement de services

Cloudizer permet de superviser et déployer des services web. Lorsqu'un nœud Cloudizer est déployé, le service est configuré via un fichier de propriétés décrivant les différentes informations nécessaires à l'identification du service. Un service est identifié par un nom et un numéro de port. Lors du lancement du nœud cloudizer, la librairie enregistre la machine et le service correspondant auprès du Contrôleur, qui met à jour le statut du service dans sa base de données et fait apparaître cette instance comme disponible.

3.3 Intégration au projet MCube

La plupart des algorithmes de traitement d'images ou de sons destinés à la plateforme MCube sont écrits sans considération pour le traitement de gros volumes de données. Or, dans le contexte de ce projet, les programmes de traitement des fichiers multimédias doivent s'exécuter sur un nombre important de fichiers à la fois, qui plus est sur une architecture de stockage distribuée. Afin d'adapter de manière transparente les algorithmes utilisés à ce contexte, le projet Cloudizer a été utilisé pour déployer et exécuter les programmes à la demande en fonction des besoins. Ainsi, un binaire exécutable peut être converti en service web à l'aide d'une description XML du programme suivant le schéma représenté en figure 2.4. Il est plus efficace d'envoyer le programme exécutable vers les données plutôt que de lui faire lire les données depuis le réseau. Ce principe est au centre des plateformes de traitement de données massives comme le système Hadoop[Whi09].

Un fichier de description XML comme celui figurant dans le listing 2.1 doit être créé pour représenter un programme et le déployer dans le système. La librairie Cloudizer contient un utilitaire permettant de créer une arborescence où seront stockés les binaires du programme et tous les dossiers et dépendances nécessaires. Deux autres dossiers sont créés par défaut : un dossier INPUT et un dossier OUTPUT. Le dossier INPUT est destiné à stocker les fichiers en entrée du programme, et le dossier OUTPUT à stocker les fichiers en sortie, à la suite d'une exécution. Une fois l'arborescence créée, il est possible d'y exécuter un serveur

Méthode	Paramètres	Action
GET	- Nom du service, - Paramètres tels que décrits dans le fichier description.xml	Renvoie le contenu d'un fichier de résultat correspondant aux paramètres donnés. Renvoie une erreur 404 si le fichier n'existe pas.
POST	Nom du service, Paramètres	Exécute le programme représenté par le service. Renvoie une erreur 500 en cas de problème à l'exécution.
PUT	Nom du service, fichier à transférer	Copie le fichier donné en paramètre dans le répertoire INPUT du service.
DELETE	Nom du service, nom de fichier	Supprime le fichier indiqué en paramètres des répertoires du service

TABLE 3.1 – Protocole HTTP : exécution de services de traitement de données

HTTP, qui sera identifié par le nom du programme précisé dans le fichier XML. Ce serveur répond aux requêtes HTTP GET, POST, PUT et DELETE avec la sémantique détaillée en tableau 3.1.

Sécurité des exécutions Les exécutables convertis de cette manière en services web peuvent être exposés à des attaques par des utilisateurs malicieux. Limiter le risque d'attaques est primordial, dans la mesure où les programmes déployés sur la plateforme sont des boites noires dont le fonctionnement est inconnu. Le serveur utilise donc l'utilitaire "Jail" [KW00] qui permet d'exécuter des binaires dans un environnement limité au strict minimum, ce qui réduit les risques d'opérations illégales ou accidentelles sur des fichiers de configuration ou système. De plus, le fichier de description précise rigoureusement les types de données attendus pour les paramètres du programme, ce qui permet au serveur de refuser les requêtes qui ne respectent pas la signature attendue de l'appel. La dernière contre-mesure est de limiter autant que possible l'exposition de ces services en leur interdisant l'accès au réseau, qui n'est donc obtenu que par le serveur généré par Cloudizer.

Exécution asynchrone L'exécution du service en mode asynchrone permet d'alléger la charge de la machine appelant le service en raccourcissant le temps de la connexion entre le client et le serveur, via l'utilisation d'une fonction de rappel distribuée. Ceci est particulièrement utile lorsque le service demandé a une latence élevée et met donc un temps

significatif à s'exécuter : maintenir une connexion ouverte pendant plusieurs secondes sans que des données ne circulent n'est pas utile. La contrepartie est que la machine appelante doit rester à l'écoute du serveur qui lui enverra le résultat d'une requête GET sur les mêmes paramètres une fois le service achevé. Ce fonctionnement est similaire à celui d'une fonction de rappel distribué. Ce rôle de client appelant peut être assuré par le répartiteur ou le contrôleur en cas de besoin. Ce mécanisme est particulièrement utile en conjonction de l'invocation parallèle décrite en section 3.2.2, car cela permet d'alléger la charge de l'entité appelante. La gestion des erreurs et des résultats retournés fonctionne de la même manière que pour un appel synchrone.

3.4 Considérations sur l'implémentation

Au cours de cette thèse, l'effort principal de développement s'est concentré sur la réalisation du répartiteur du système Cloudizer. Les premières versions de ce système de répartition étaient implantées directement dans l'interface web du système et souffraient des problèmes de performance du langage Groovy. Le principal facteur d'amélioration de la performance de ce système a donc été déplacer les classes assurant la répartition des requêtes dans une librairie séparée, permettant de découpler l'interface Web du répartiteur.

Le répartiteur en lui-même est un projet de taille modeste, représentant environ 400 lignes de code Java, et 964 lignes de code Groovy pour l'interface Web. Les politiques de répartitions implémentées sur le répartiteur représentent quant à elles environ 2000 lignes de code. Cet aspect quantitatif ne permet pas de mesurer avec certitude l'effort total réalisé car l'ensemble de cette base de code a subi de multiples changements au cours du temps, principalement concernant l'optimisation ou la correction de bogues détectés lors des simulations. Parmi les optimisations réalisées, le passage en pur Java a été le plus efficace en terme de gains de performance brute. De même, le passage d'un modèle de traitement des requêtes événementiel plutôt que par fil d'exécution a permis un fort gain dans la capacité de traitement du répartiteur.

Le générateur de service Web représente environ 1800 lignes de code Java entre l'implémentation du serveur, l'intégration avec Cloudizer et un module d'intégration avec le

système Hadoop permettant d'utiliser ce système pour répartir les traitements. Ces développements spécifiques et de petite taille n'ont pas fait l'objet d'une méthode de développement particulière. Un soin particulier a été apporté à là modularité de ces projets, en permettant une extension facile de leurs fonctionnalités respectives : ajouts de nouvelles politiques de distribution pour le répartiteur ou de nouveaux environnements d'exécution pour le générateur de services.

3.5 Discussion

L'architecture de Cloudizer permet donc de déployer et d'exécuter un grand nombre d'applications différentes en déléguant la gestion des données à d'autres systèmes. Les services pouvant tirer partie de Cloudizer vont de l'application Web standard ou la Web Archive. À cela s'ajoute la possibilité d'utiliser un binaire comme service web ce qui peut être particulièrement utile dans le cadre d'applications héritées. Cette capacité a été développée pour permettre une intégration simple de ce projet dans la plateforme MCube afin d'assurer la répartition des requêtes de services de traitement de données. Ce service est suffisamment générique pour assurer à la fois la distribution des services de traitements de données de la plateforme MCube mais aussi de la partie Web fournissant l'interface utilisateur et la communication avec les passerelles.

Le fait de pouvoir sélectionner la stratégie de répartition la plus appropriée pour un service donné dans la plateforme Cloudizer permet d'évaluer et de comparer la performance des différentes stratégies. Cependant le comportement des stratégies mises au point ainsi que leur tests en environnement réels sont longs et fastidieux. Les nouveaux environnements de déploiements tels que les plateformes de Cloud Computing permettent d'utiliser de grands nombres de machines, il est donc nécessaire d'utiliser des outils de simulations appropriés afin de faciliter le développement de ces stratégies.

Chapitre 4

Simizer : un simulateur d'application en environnement cloud

4.1 Introduction

La conception du simulateur Simizer a commencé comme un simple programme de test pour les stratégies de répartition de charge en amont de leur déploiement dans le projet Cloudizer. Ses fonctionnalités ont été étendues par la suite, dans le but de fournir des résultats de plus en plus réalistes et de permettre la réalisation de simulations plus diverses, comme des simulations de protocoles de synchronisation de processus dans les systèmes distribués.

Le développement des grilles de calcul, et plus récemment du cloud computing, a donné naissance à de nombreux simulateurs de plateformes de traitement à large échelle. Les simulateurs de grilles, conçus pour simuler de grands nombres de machines partagées par plusieurs utilisateurs, ont été naturellement adaptés pour prendre en compte la virtualisation et simuler les plateformes de calcul à la demande : les "clouds". Par exemple le Grid Sim toolkit [MB02] a servi de base au développement du simulateur CloudSim [RNCB11], et l'utilitaire SimGrid [CLQ08] intègre aujourd'hui de nombreuses options permettant de simuler le fonctionnement des centres de virtualisation. Simizer a donc été conçu de manière à combler un vide dans les solutions de simulation existantes se rapportant au Cloud Computing.

4.1.1 Simulateurs de Cloud existants

Les principaux représentants des simulateurs de clouds sont les projets SimGrid[CLQ08] et CloudSim[RNCB11]. Ces projets permettent de simuler entièrement une infrastructure de type cloud en écrivant un programme utilisant les objets mis à disposition par l'interface de programmation fournie par la librairie. Ces deux utilitaires sont principalement destinés à la recherche en matière de conception et d'évaluation de l'architecture sous-jacente des plateformes de services informatiques à la demande. I ls permettent d'obtenir un modèle du comportement du centre de données dans son ensemble. Par exemple, des travaux menés avec CloudSim par Beloglazov et Buyya [BB12] ont permis d'analyser l'influence de différentes stratégies d'allocation de ressources sur la qualité de service rendue aux utilisateurs et la consommation électrique d'un centre de données. L'utilitaire SimGrid [BLM+12] a été conçu pour simuler de manière transparente le comportement d'applications de calculs distribués massifs, avant leur déploiement effectif sur une grille. Cependant, l'application de ce logiciel au domaine du Cloud Computing se limite à la simulation du fonctionnement interne des infrastructures. Le simulateur GreenCloud [KBAK10] est destiné à modéliser la consommation électrique des centres de données. Il se fonde sur le simulateur de protocoles réseaux NS-2[MFF], ce qui rend son modèle de simulation de réseau extrêmement fiable. Le projet iCanCloud[NVPC+12] utilise un modèle mathématique pour simuler les coûts engendrés par l'utilisation d'une plateforme de Cloud Computing comme le service EC2 d'Amazon [1].

Cependant, ces simulateurs sont destinés aux fournisseurs de services Cloud, et permettent de modéliser facilement des prototypes de plateformes ou les coûts engendrés par les infrastructures en fonction d'un modèle d'utilisation du service. Il n'existe pas à ce jour de simulateurs d'infrastructure *cloud* permettant de simuler le comportement d'une application déployée dans ce type d'environnement [SL13]. Bien que des efforts aient été récemment faits dans cette direction, comme le simulateur CDOSim[FFH12] ce logiciel n'est pas publiquement disponible. Il n'existe donc pas à ce jour de simulateur permettant d'établir finement l'influence de protocoles de niveau applicatif utilisés par un système. Ce besoin existe car il a été montré à plusieurs reprises que les plateformes d'infrastructures à

1. `http://aws.amazon.com`, consulté le 5 octobre 2013

la demande souffrent par leur nature partagée, d'une dépendance entre le niveau d'utilisation global de la plateforme et la performance des applications. Pouvoir évaluer à l'avance le comportement d'une application dans ce type de contexte d'exécution présente donc un avantage certain. La conception de Simizer vise donc à remplir ce besoin en permettant l'évaluation de la performance théorique d'applications de type services web déployées sur des plateformes de *cloud computing*.

L'architecture du simulateur Simizer et le fonctionnement de ses différents composants sont décrits en section 4.2, puis l'utilisation pratique du simulateur est détaillée en section 4.3. La section 4.4 fournit à travers deux expérimentations une validation du fonctionnement du simulateur par rapport à la librairie CloudSim, et la section 4.5 discutera des futurs développements de l'utilitaire.

4.2 Architecture de Simizer

Simizer est un simulateur à événements discrets écrit en JAVA. Il est fondé sur une architecture à trois couches : une couche de bas niveau fournissant les classes de gestion des événements, une couche Architecture utilisant les classes de gestion d'événements pour simuler le comportement des machines virtuelles et des réseaux, et une couche de niveau Application, qui permet d'implémenter les protocoles à tester dans le simulateur.

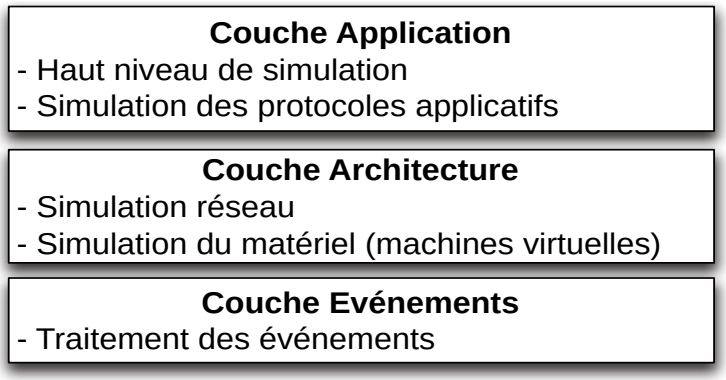


FIGURE 4.1 – Architecture de Simizer

TABLE 4.1 – Lois de probabilité disponibles dans Simizer

loi	paramètres
Exponentielle	espérance (λ)
Gaussienne	moyenne (μ)
Poisson	espérance (λ)
Uniforme	densité
Zipf	biais (s)

4.2.1 Couche Evénements

La couche de traitement des événements est la couche de plus bas niveau du simulateur. Elle fournit une boucle de traitement d'événements permettant de garantir l'ordre d'exécution des événements et d'optimiser le traitement de ces objets. Elle fournit aussi un ensemble de classes utilitaires permettant la génération de nombre aléatoires selon différentes lois de distribution.

Génération de nombres aléatoires La génération des événements produisant l'exécution de la simulation est régie par les lois de distribution aléatoires décrites dans le tableau 4.1. Le générateur de nombre aléatoire utilisé est celui de la Java Virtual Machine. Grâce à cette classe il est possible de générer trois types de valeurs aléatoires :

- des entiers distribués uniformément sur les valeurs comprises entre 0 et un nombre donné en paramètre,
- des flottants double précision : distribués uniformément sur les valeurs comprises entre 0 et un nombre donné en paramètre,
- des flottants double précision distribués selon une loi Gaussienne d'espérance 0 et d'écart type 1.

Les distributions utilisées sont donc générées à partir de ces trois types de valeurs. Deux contraintes principales sont posées : les valeurs obtenues pour la simulation doivent être entières, et l'intervalle des valeurs possibles est limité entre 0 et un paramètre spécifié. Les lois Gaussiennes, Uniformes et Zipfiennes sont utilisées principalement pour décrire les distributions de requêtes utilisateurs. La distribution Zipfienne décrit correctement divers phénomènes utilisés dans les simulations informatique [BCC+99, Fei02] tels que la distribution des tailles des fichiers d'un système, ou la distribution des pages les plus fréquemment

accédées sur un site Web. Il est donc intéressant d'étudier le comportement de différentes stratégies de répartition par rapport à des requêtes distribuées selon cette loi. Plus récemment il a été observé que les effets dus à la mise en cache des données les plus fréquemment accédées, en particulier dans les systèmes web, avait des conséquence visibles sur la distribution des requêtes observées, ces dernières ne suivant plus strictement une loi de Zipf [DCGV02, GTC+07]. Il est donc nécessaire de pouvoir tester le comportement des politiques de répartition par rapport à d'autres distributions possibles comme loi Gaussienne ou Uniforme.

Les lois de Poisson et Exponentielles sont utilisées pour décrire le comportement des utilisateurs. La loi de Poisson est utilisée pour générer les arrivées de nouveaux clients du système simulé et la loi Exponentielle est utilisée pour déterminer la durée d'utilisation du système par chaque utilisateur, ainsi que l'intervalle de temps séparant les réponses et les requêtes d'un utilisateur [Fei02].

Traitement des événements Le diagramme 4.2 montre les classes composant l'architecture du moteur d'événements utilisé par Simizer. Une simulation est constituée de plusieurs entités appelées "Producteurs d'événements" (EventProducer) qui produisent chacun un ensemble d'événements particuliers (classe ConcreteEvent). Un événement est caractérisé par une donnée, une cible et un horodatage, qui indiquent respectivement à quel instant et à quel objet de la simulation cet événement doit être signalé. Les producteurs d'événements servent à implanter les divers modèles de systèmes nécessaires à la simulation. Chaque Producteur crée des événements et les planifie en choisissant l'horodatage selon les règles du système simulé. C'est la production de nouveaux événements par les Producteurs d'événements qui fait avancer la simulation. Les événements ainsi produits sont stockés dans une file d'attente ordonnée : le Canal (Channel). À intervalles réguliers au cours de la simulation, le Répartiteur d'événements (EventDispatcher) interroge le Canal pour savoir si il y a des événements en attente. Le cas échéant, le Répartiteur retire de la file l'événement ayant la datation la moins avancée, et le signale à la cible correspondante. Les événements sont signalés l'un après l'autre dans l'ordre croissant des horodatages. Ce choix permet d'éviter les conflits dits de causalité qui peuvent se produire lorsqu'un événement

qui n'est pas encore "arrivé" est exécuté avant ou en même temps qu'un événement qui le précède, ce qui peut provoquer un résultat incohérent voire un blocage de la simulation. Les événements sont spécialisés en fonction du type d'action à réaliser. Ces actions sont

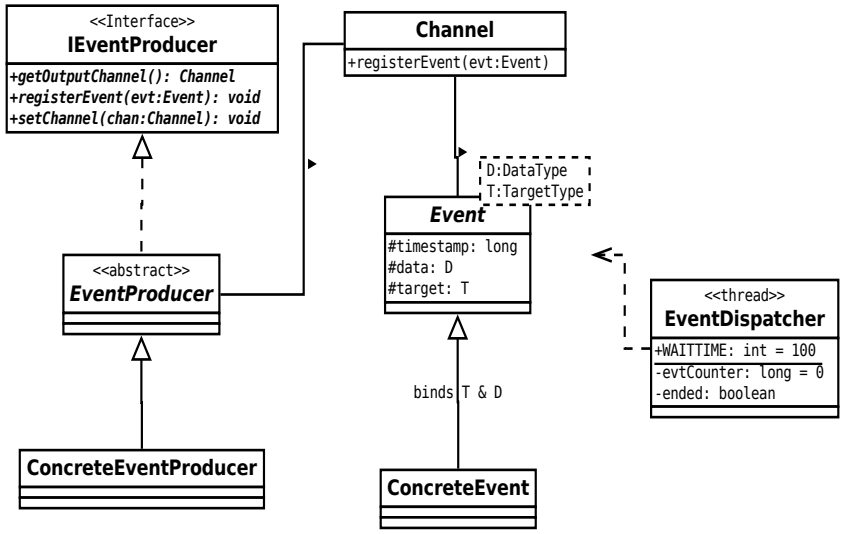


FIGURE 4.2 – Entités de gestion des événements

implantées dans la couche supérieure du simulateur : la couche d'architecture.

4.2.2 Couche architecture

La couche d'architecture de Simizer représente les différentes entités faisant l'objet de la simulation. En l'occurrence, Simizer est conçu pour simuler des systèmes informatiques distribués. Les entités simulées représentées dans le diagramme de classes 4.3 peuvent être définies ainsi : Le Réseau (Network) permet de simuler les délais et erreurs inhérentes au fonctionnement des réseaux modernes comme internet ou les réseaux locaux. Les Nœuds (Nodes) correspondent aux machines composant le système simulé. Ils communiquent via les Réseaux (Network). Il existe trois catégories de Nœuds : les Clients (ClientNode) représentent les machines clientes de services disponibles sur les Serveurs (ServerNode). Lorsque les serveurs et les clients sont sur des réseaux différents, un Répartiteur (LBNode) sert d'intermédiaire entre les clients et les serveurs. Le comportement des clients est géré par la classe ClientManager, qui ordonne la création de nouveaux clients selon une loi de pro-

babilité configurable par l'utilisateur. Lorsqu'un client est créé, il envoie une "Requête" (Request) via un "Message" vers un serveur en passant par le réseau (Network) qui le contient. Le réseau applique ensuite un certain retard au message, proportionnel à la taille du message, pour simuler le délai impliqué par l'utilisation du réseau. Un nouvel événement est planifié pour signifier l'arrivée de la requête sur la machine cible. Une "Requête" (Request) est caractérisée par une liste de ressources (classe Resource) et un nombre d'instructions. Les ressources sont les objets stockés sur le serveur dont la requête a besoin pour son exécution. Le nombre d'instructions est la quantité de code à exécuter par le serveur pour pouvoir générer la réponse à la requête avant de l'envoyer au nœud client. Les entités de la couche d'architecture dérivent les Producteurs de la couche de gestion d'événements.

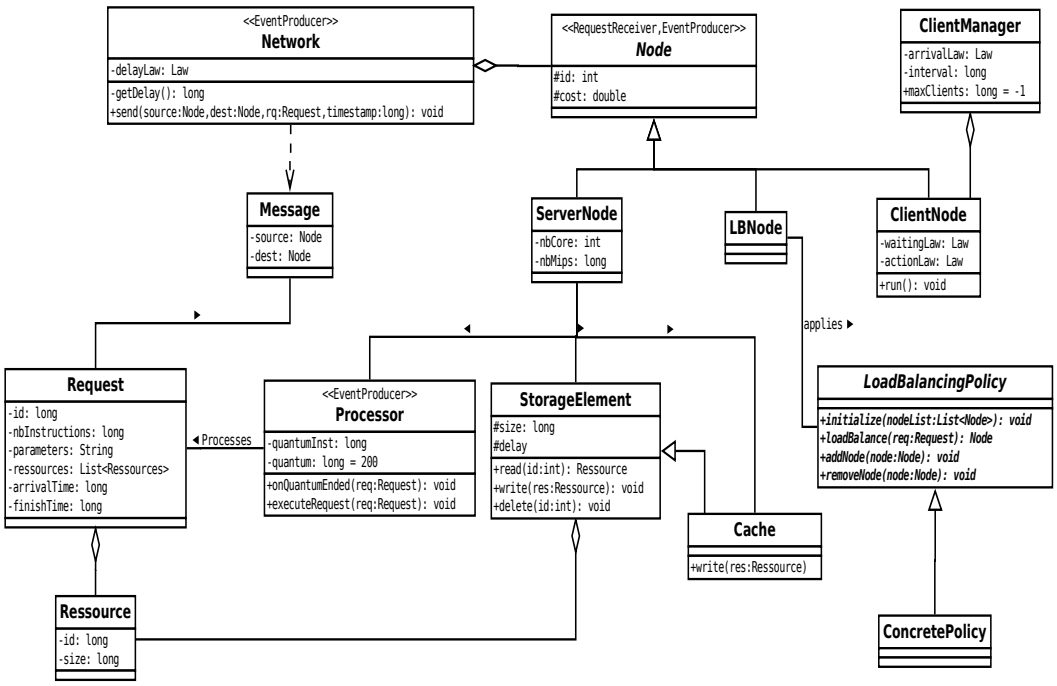


FIGURE 4.3 – Entités de simulation système

Sur chaque serveur, un Processeur (Processor) simule l'exécution des tâches affectées à la machine. Les caractéristiques du processeur sont son nombre de cœurs et la quantité d'instructions que chaque cœur peut exécuter en une seconde, définie en millions d'instruc-

tions par seconde (MIPS).

Le stockage disque et l'espace mémoire sont respectivement simulés par les classes StorageElement et Cache. Ce modèle de simulation est très abstrait et correspond à une architecture de ferme de serveurs web. Lorsqu'une requête arrive sur un serveur, le serveur passe le relais au processeur. Le processeur évalue alors la taille des ressources afin d'évaluer à quel moment la requête peut commencer à être exécutée. Les délais typiques de chargement de données en mémoire depuis les disques d'une machine sont régulièrement évalués et disponibles en ligne [2]. Ce sont ces délais qui servent de référence pour Simizer. Lorsque les données sont présentes en mémoire, la requête est considérée comme prête pour l'exécution.

4.2.3 Fonctionnement du processeur

La classe Processor est une entité particulière en cela qu'elle implémente la politique d'exécution des tâches appliquée sur la machine simulée. Il s'agit donc plus d'une simulation du fonctionnement du système d'exploitation que du matériel proprement dit.

Plus un processeur fonctionne rapidement, plus il exécute un grand nombre d'instructions par seconde. Dans les systèmes d'exploitation modernes, cette capacité de traitement est partagée entre les fils (threads) d'exécution des différents programmes s'exécutant sur la machine. Ce partage se fait en allouant le processeur à chaque fil d'exécution durant un intervalle de temps limité appelé "quantum". Un quantum est un intervalle de temps fixé par le système d'exploitation de la machine. Sous le système Linux, par exemple, cet intervalle correspond à 4 millisecondes [Mol07]. Simizer utilise cette notion pour simuler l'exécution parallèle de plusieurs requêtes sur un même processeur. Par conséquent, durant un quantum, seul un nombre limité d'instructions peut être exécuté, en fonction de la puissance du processeur. Lorsque l'intervalle est fini, ou que le code en cours d'exécution est interrompu ou bloqué, le processeur change de contexte et commence ou continue l'exécution d'un autre processus. Pour simuler ce comportement, un événement est utilisé pour signaler à chaque Processeur du système la fin d'un quantum. Ainsi, lorsqu'un quantum

2. `http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html`, consulté le 5 octobre 2013

est terminé pour une requête, le processeur met à jour la quantité d'instructions restantes dans la requête et passe à la requête suivante.

Cette approche permet de définir finement les différences de performances entre différents modèles de processeurs. La classe processeur simule l'ensemble des cœurs de la machine fonctionnant en parallèle. Pour optimiser ce fonctionnement il existe plusieurs politiques d'ordonnancement des tâches, qui peuvent être implantées en dérivant la classe Processeur. Le modèle de processeur utilisé tout au long de ce document est un modèle multitâche préemptif, utilisant la politique du premier arrivé, premier servi (First Come First Served). Cette politique présente l'avantage d'être simple à implémenter. En l'état actuel de la simulation, toutes les tâches sont considérées comme issues d'un seul et même processus : l'application en cours d'exécution. Lorsque le quantum actuel est expiré, la tâche en cours d'exécution est remplacée par la suivante dans la liste.

Simizer diverge de la librairie CloudSim au niveau de la simulation des processeurs à multicœurs : dans Simizer, les cœurs des processeurs ne peuvent être que partagés en espace (Space Shared) et pas en temps (Time Shared). Une tâche n'est donc affectée qu'à un seul cœur à la fois. Cependant, le modèle de Processeur étant dérivable, il sera possible d'implémenter des fonctionnements plus détaillés dans le futur.

4.2.4 Exécution d'une requête

Dans le simulateur Simizer, une requête représente une requête HTTP. Une requête déclenche donc un traitement sur la machine sur laquelle elle est reçue. Ce traitement est défini par la liste des ressources à accéder pour mener la requête à bien, ainsi que par le nombre d'instructions à exécuter pour compléter la requête. L'exécution d'une requête dans Simizer commence lors de la réception d'un Message par un nœud serveur. Le serveur examine la liste des ressources demandées par la requête et calcule en fonction des propriétés du disque attaché à ce serveur le temps nécessaire pour accéder à ces ressources. Lorsque ce temps est écoulé, un événement est signalé au processeur, indiquant que les données de la requête sont prêtes. La tâche en cours d'exécution sur le processeur est alors préemptée par la nouvelle requête, et l'exécution de la requête se poursuit selon la stratégie d'ordonnancement utilisée par le processeur. Lorsqu'une requête est terminée (il n'y a plus

d'instruction à exécuter) un événement est signalé au nœud serveur. Le serveur encapsule alors la requête dans un message de réponse à la machine cliente. À la réception de ce message, le client compare l'horodatage d'arrivée et de création de la requête pour évaluer le temps d'exécution simulé. Le résultat est collecté et affiché sur la sortie standard au format CSV, en indiquant le numéro de la requête l'identifiant des paramètres utilisés, le temps de réponse mesuré, le temps de transfert sur le réseau, et l'identifiant du client ayant émis la requête.

4.3 Utilisation du simulateur

Le simulateur peut fonctionner selon trois modes différents : premièrement il est possible de lui faire générer une trace, c'est à dire une séquence prédéterminée d'arrivées de clients et de requêtes à partir des différentes lois de probabilités disponibles dans le simulateur. L'utilisateur peut par exemple choisir la loi de distribution des arrivées de nouveaux clients dans le système en sélectionnant une loi de Poisson ou une loi Normale et en définissant leurs paramètres. La durée de vie des clients peut aussi être définie par une loi, ainsi que la distribution des différentes requêtes que les clients peuvent envoyer au système simulé. C'est le mode "génération". Il ne s'agit pas à proprement parler d'une simulation. Les différentes lois disponibles pour simuler le comportement des utilisateurs figurent dans le tableau 4.1.

À partir d'une trace générée de cette manière, il est possible de lancer une simulation en précisant les quantités de serveurs disponibles dans le système et en choisissant le répartiteur approprié. C'est le mode de "simulation de trace". Les traces peuvent être écrites par l'utilisateur dans un simple fichier CSV. Cette approche permet de tester des séquences de requêtes précises, ce qui peut être utile dans le cadre de la simulation de protocoles applicatifs, afin de tester le comportement du système face à des séquences de requêtes problématiques. Par exemple, ceci peut être utilisé pour vérifier la cohérence des données lors de la simulation d'un système de stockage distribué, en créant une séquence de requêtes déclenchant des accès concurrents et conflictuels à une même ressource. Le troisième mode de simulation est le mode "génératif". L'utilisateur définit les différentes lois régissant le comportement des clients comme lors de la génération d'une trace, ainsi que le

nombre de machines à utiliser et les caractéristiques de l'application simulée. Pour lancer une simulation, l'utilisateur doit fournir au simulateur trois fichiers de configuration :

Fichier de définition des requêtes : Il s'agit d'un fichier au format CSV répertoriant les propriétés des différentes requêtes que les nœuds clients peuvent envoyer au système simulé. Les requêtes sont définies par un identifiant, la liste des ressources (les données) qu'elles accèdent, et la quantité d'instructions nécessaires à leur exécution.

Fichier de définition de la charge : Ce fichier est un fichier de propriétés permettant de définir les différentes lois gouvernant le comportement des clients du système simulé. Tels que définis dans plusieurs travaux sur les modèles de comportements des clients d'un système distribué, le comportement des clients peut être régi par quatre lois :

- *Loi de sélection des requêtes :* Chaque client est un agent indépendant, qui a la possibilité de choisir d'envoyer un certain nombre de requêtes de service au système. A chaque nouvel envoi de requête, le client sélectionne les paramètres parmi ceux figurant dans le fichier de paramètres des requêtes décrit au paragraphe précédent.
- *Loi d'arrivée des clients :* le processus d'arrivée des clients est paramétré par un intervalle de temps et le nombre moyen de nouveaux clients arrivant durant cet intervalle. Une supposition raisonnable [Fei02] consiste à utiliser une loi de probabilité discrète telle qu'une loi de Poisson ou une loi uniforme pour déterminer le nombre d'arrivées à chaque intervalle.
- *Loi de durée de vie des clients :* Une durée de vie est affectée à chaque client lors de sa création. La durée de vie d'un client est le temps pendant lequel il est actif et communique avec le système par l'envoi ou la réception de requêtes. Il est d'usage d'utiliser une loi exponentielle pour modéliser les différentes durées de vies possibles des clients.
- *Loi d'attente des clients :* Un client peut envoyer des requêtes tout au long de sa durée de vie. Cependant, il est raisonnable de supposer que les envois de requêtes se font après la réception de la réponse à la requête précédente. Pour modéliser ce fonctionnement, après chaque réception de requête, un client attend un temps déterminé par une loi de probabilité, qui est la loi d'attente des clients.

Selon le modèle de comportement utilisé, les lois peuvent être configurées avec différents paramètres comme précisé dans le tableau 4.1. Une classe abstraite, la classe Loi (Law), permet à l'utilisateur qui le souhaite d'ajouter les lois de distribution qui lui sembleraient manquantes. Il lui faut ensuite préciser dans le fichier de configuration de la charge les noms des différentes lois qu'il souhaite utiliser pour chaque processus ainsi que les paramètres voulus.

Listing 4.1 – Exemple de fichier de description des serveurs

```
[
        {"nb":2,
        "memorySize": 1048576,
        "diskSize":512000000,
        "cost":10,
        "ProcessorName":"simizer.processor.LinuxProcessor",
        "cpuSlots":2,
        "nbMips":2500.0,
        }
        {"nb":2,
        "memorySize": 2097152   ,
        "diskSize":512000000,
        "cost":15,
        "ProcessorName":"simizer.processor.LinuxProcessor",
        "cpuSlots":4,
        "nbMips":3200.0,
        }

]
```

Fichier de définition des serveurs : Ce fichier de configuration au format JSON (Java Script Object Notation) permet aux utilisateurs de décrire les propriétés matérielles du système simulé à l'aide d'une notation simple et claire. Les serveurs sont décrits par le nombre de machines ayant les mêmes caractéristiques, la quantité de cœurs pour chaque machine, la puissance individuelle de chaque cœur, notée en millions d'instructions par secondes, ainsi que le type de processeur utilisé, l'espace de stockage disponible et la quantité de RAM disponible sur chacune des machines. Le listing 4.1 montre un exemple de fichier de configuration des serveurs. Le système simulé sera constitué de 5 serveurs : 2 machines double cœur d'une puissance de 1000 MIPS/cœur (donc 3000 MIPS en tout) et

de 3 machines quadruple cœur d'une puissance de 1200 MIPS/cœur. Chaque machine a 1 ou 2 gigaoctets de mémoire vive ("memorySize") et peut stocker 512 gigaoctets de données ("diskSize"). Le coût horaire de la machine simulée est aussi indiqué ("cost"), de manière à pouvoir fournir des analyses de coût de location de machines.

Au cours d'une simulation avec Simizer, toutes les requêtes envoyées et les réponses reçues sont collectées et stockées dans un fichier de type CSV. Ces résultats peuvent ensuite être analysés à l'aide d'outils spécialisés comme R [R C13]. Le simulateur ne possède pour le moment pas d'interface graphique.

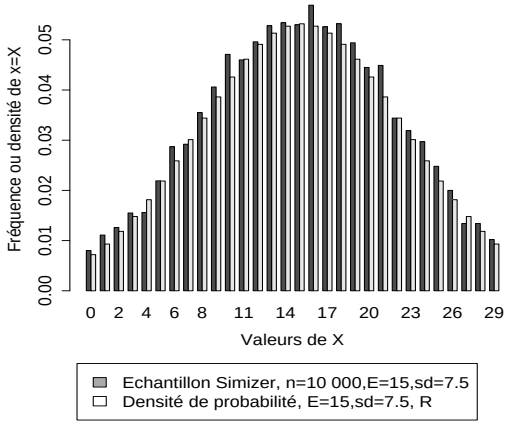
4.4 Validation

Cette section présente des travaux d'évaluation du simulateur Simizer, sur deux domaines : premièrement la précision des générateurs de nombres aléatoires utilisés, et deuxièmement le fonctionnement de la simulation des processeurs des machines par rapport à la librairie de simulation CloudSim.

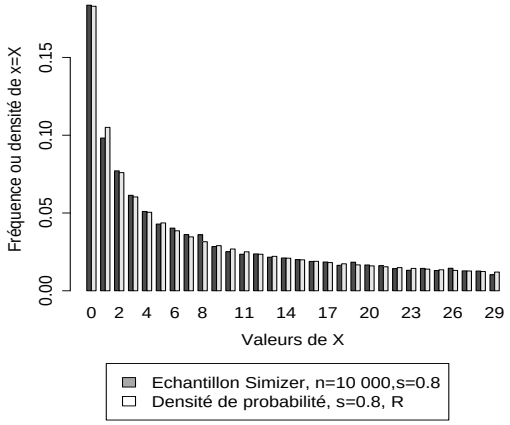
4.4.1 Génération de nombres aléatoires

Nous avons évalué le fonctionnement de trois lois implantées dans le logiciel Simizer : la loi Gaussienne, la loi Exponentielle et la loi de Zipf. La loi uniforme n'est pas représentée ici car l'implémentation de la Java Virtual Machine est directement utilisée pour cette loi, et ne nécessite donc pas de vérification. Des échantillons de 10000 valeurs sont tirés de chacune des lois de distribution implantées dans le simulateur, avec pour ensemble de résultats les valeurs de 0 à 29. La distribution des valeurs de ces échantillons est comparée aux densités de probabilité de lois obtenues par le logiciel R [R C13] dans les histogrammes en figure 4.4.

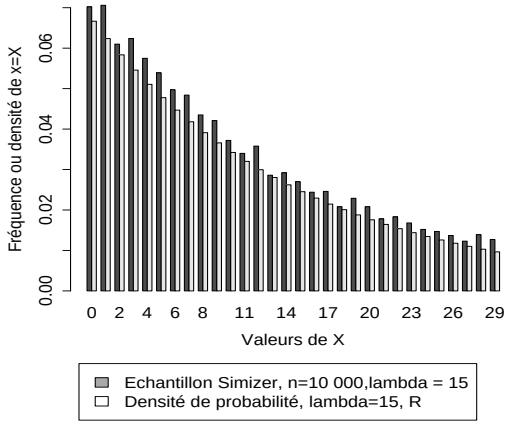
Trois lois sont représentées sur ces diagrammes : une loi Gaussienne, de moyenne 15 et d'ecart type 7.5 (4.4a), une loi de Zipf à 30 rangs et de biais 0.8 (4.4b) et une loi Exponentielle (4.4c). Un examen visuel de ces histogrammes montre que la distribution des valeurs obtenue par la génération de nombres aléatoires avec Simizer et les densités de probabilités attendues pour chaque loi sont très proches. Il est aussi possible de noter



(a) Distribution Gaussienne



(b) Distribution Zipfienne



(c) Distribution Exponentielle

FIGURE 4.4 – Comparaison des distributions aléatoires de Simizer

que la marge d'erreur est légèrement plus importante pour la distribution Gaussienne, car la loi est "tronquée" pour fournir des valeurs comprises dans l'intervalle souhaité par l'utilisateur. La conséquence est que la somme des fractions observées dans l'échantillon représenté sur l'histogramme 4.4a est égale à 1, alors que la somme des densités représentées sur le même diagramme est égale à 0.96. Ceci est dû au fait que les valeurs doivent rester dans le domaine spécifié par l'utilisateur, ici l'intervalle $[0, 29]$, ce qui dans le cas d'une loi continue telle que la loi normale, peut provoquer de légères erreurs.

Loi d'ajustement	χ^2	Degrés de libertés	p-valeur
Gaussienne	32.35	29	0.30
Zipf	23.84	29	0.74
Exponentielle	26	29	0.61

TABLE 4.2 – Résultats des tests du Chi-2

Le tableau 4.2 montre les résultats des tests d'ajustements par la méthode du χ^2 menés sur les différents échantillons représentés en figure 4.4. Ces résultats corroborent les observations empiriques en confirmant, avec un taux d'erreur de 5%, l'hypothèse selon laquelle l'échantillon généré suit bel et bien la loi qui lui est associée. Il est aussi possible de noter la p-valeur plus faible du test de la loi Gaussienne, qui confirme le problème observé sur les figures.

Le simulateur Simizer permet donc de générer correctement des nombres aléatoires selon les distributions voulues par l'utilisateur. Une interface de programmation et de chargement dynamique des classes définissant les distributions a été implantée dans le simulateur pour permettre aux utilisateurs d'utiliser des lois plus complexes, ou plus appropriées au contexte que celles disponibles par défaut dans le logiciel. Le choix d'utiliser nos propres classes de génération de distribution a été fait pour réduire le nombre de dépendances de la librairie.

4.4.2 Fonctionnement des processeurs

Afin de vérifier la pertinence de l'implémentation de la simulation, le fonctionnement des processeurs dans Simizer, décrit en section 4.2.3 de ce chapitre, a été comparé avec le fonctionnement des processeurs de la librairie CloudSim. CloudSim définit les tâches à exécuter par un nombre d'instructions à exécuter, défini en Millions d'Instructions (MI) et la capacité de calcul des processeurs simulés est définie en MI par Seconde (MIPS). La comparaison de ce niveau de simulation est donc possible entre Simizer et cette librairie, car les paramètres sont définis de la meme manière.

Par défaut, deux modèles de fonctionnement sont disponibles dans la librairie CloudSim : un fonctionnement en espace partagé (*Space Shared*), et un fonctionnement en temps partagé (*Time Shared*). Le fonctionnement de Simizer étant fondé sur le temps partagé, c'est ce modèle qui a été choisi pour la comparaison. De plus, le modèle d'utilisation des

Nombre de tâches	Nombre de cœurs	Temps par tâche	
		Simizer	CloudSim
1	1	10	10
2	1	20	20
3	1	30	30
4	1	40	40
1	2	10	10
2	2	10	10
3	2	15	15
4	2	20	20
1	4	10	10
2	4	10	10
3	4	10	10
4	4	10	10

TABLE 4.3 – Comparaison de Simizer et CloudSim, temps d'exécution de tâches

ressources Processeur retenu pour CloudSim est le modèle d'utilisation total : la tâche en cours d'exécution utilise 100% des ressources disponibles sur la machine.

Le test consiste à simuler le fonctionnement d'une seule machine et de lui soumettre différentes tâches à exécuter, selon des paramètres équivalent dans les deux simulateurs. Par exemple le premier test consiste à soumettre une tâche de 10000 Millions d'Instructions (MI) à exécution sur une machine équipée d'un processeur capable d'exécuter 1000 Millions d'Instructions Par Seconde (MIPS), ce qui devrait fournir un temps d'exécution de dix secondes pour les deux simulateurs. Le tableau 4.3 résume les temps obtenus pour différentes configurations de tests sur les deux simulateurs. Pour les deux simulateurs, les cœurs de la machine sont dotés d'une capacité de traitement de 1000 MIPS, et la taille des tâches est fixée à 10000 MI. Il apparait donc que le fonctionnement de la simulation de processeurs de Simizer procure des résultats cohérents par rapport à un simulateur de référence dans ce domaine.

4.5 Discussion et travaux futurs

Simizer est un simulateur à événements discrets conçu pour émuler le fonctionnement de systèmes distribués à large échelle. L'objectif de ce développement est de parvenir à distinguer et évaluer finement les effets de l'utilisation de différents protocoles de répartition

de cohérence ou de synchronisation sur un grand ensemble de machines. Les résultats obtenus à ce jour permettent d'établir des tendances cohérentes avec les expérimentations dans les systèmes réels (cf. section 6.7).

Ce projet représente un développement conséquent avec environ 10 000 lignes de code Java. La gestion des événements et des classes de gestion des processeurs et des Clients (8000 lignes) ont été développées en parallèle des travaux de recherche décrit dans cette thèse. Le projet a subi de multiples révisions de son fonctionnement, notamment dans le but de supporter de nouveaux cas d'utilisations autres que l'exécution d'algorithmes de répartition de charge, et afin de réduire le temps d'exécution des simulations.

Les travaux d'amélioration du simulateur se porteront à l'avenir sur trois aspects principaux :

- Utilisation : il s'agit de simplifier l'utilisation du simulateur pour un développeur externe souhaitant tester ses idées. Bien que le code source du logiciel soit disponible [3], son utilisation demeure encore difficile pour les utilisateurs n'ayant pas participé au développement du logiciel. De plus, la configuration des simulations peut encore faire l'objet de multiples simplifications.
- Performance : Le cœur de l'application, le moteur de traitement d'événements, peut être amélioré afin de permettre l'exécution en parallèle de plusieurs événements, ce qui permettra de réduire considérablement le temps de simulation total.
- Couche simulation : Il serait utile pour la simulation de protocoles particuliers, d'implanter la gestion de la sémantique des requêtes, et que les nœuds puissent communiquer entre eux de manière directe, sans passer par la relation client-répartiteur-serveur. Un autre élément utile serait la simulation des phénomènes de contention pouvant apparaître sur les plateformes de *cloud computing* existantes [BS10, IOY+11], qui peuvent influencer la performance des processeurs et du réseau.

Simizer est donc utilisé pour développer et tester le comportement et l'influence de stratégies de répartition de charge, car il facilite leur développement et leur mise en place. La meilleure des simulations ne remplacera cependant pas une évaluation en conditions réelles. L'évaluation de la pertinence des simulations fournies par Simizer sera montrée dans les sec-

3. `https://simizer.forge.isep.fr/scm/?group_id=615`, consulté le 5 octobre 2013

tions d'évaluation des stratégies de répartition mises au point durant ces travaux. Simizer ne prétend pas prévoir exactement le comportement des applications lorsqu'une nouvelle stratégie de répartition est choisie mais permet de déterminer les tendances qui se dégageront des tests à venir.

4.6 Conclusion

Le projet MCube est un projet innovant, mais large dans ses objectifs et dans les différentes problématiques de recherche qu'il aborde. L'axe retenu à la suite de l'étude des plateformes de gestion de données multimédia existantes est celui de la distribution des traitements. La plateforme Cloudizer a donc été développée pour permettre le déploiement et la distribution des différents traitements utilisés dans la plateforme Mcube, sous forme de services Web. Les plateformes Cloudizer et Simizer sont liées par leur conception car le premier a été conçu à l'origine comme un simulateur du second. Cette relation entre les deux logiciels permet une adaptation facile des politiques de répartition à la librairie Cloudizer, facilitant d'autant plus les tests de nouvelles stratégies de distributions de requêtes, en amont de leur déploiement. De plus, la mise au point de ce simulateur tache de combler un vide dans l'offre actuelle des outils de simulations d'infrastructure de *Cloud Computing* en fournissant des interfaces permettant de simuler des protocoles applicatifs. Le reste de ce travail se concentre sur la mise au point de deux nouvelles stratégies de répartition de charge pour les services web déployés dans les environnements Cloud. Le chapitre suivant établit l'état de l'art actuel en matière de stratégies de distribution de requêtes et de répartition de charge.

Chapitre 5

Algorithmes de répartition de charge

5.1 Introduction

La répartition de charge consiste à affecter des tâches, ou des morceaux de travail, à différentes machines d'un système informatique distribué. Ces techniques permettent d'obtenir la meilleure utilisation possible des ressources disponibles sur chaque machine du système. Le but étant, soit de minimiser le temps d'exécution d'un calcul distribué, soit de maximiser le nombre de taches réalisées dans un temps donné. Il s'agit donc du point de vue théorique d'un problème d'optimisation des ressources informatiques disponibles en fonction des besoins du calcul à effectuer ou des caractéristiques des tâches à réaliser. Pour résoudre ce problème, il est aussi important de prendre en compte les caractéristiques du système étudié, à savoir :

- Type de système : Quel est le type de l'application étudiée ? Est-ce un système de fichiers distribué ? Une base de données ? Un système de traitements parallèle ?
- Ressources utilisées : Quelles sont les ressources principalement consommées par les unités de travail réparties ? Est-ce un travail effectuant des calculs intensifs ou bien chargeant la mémoire de manière importante ?
- Contraintes externes : Y a-t-il une limite quant à l'utilisation de certaines ressources ? Le travail doit-il s'effectuer en un temps ou un budget limité en raison d'un contrat de service ?
- Granularité : Quelle est la taille des unités de travail à répartir ? Répartir des travaux de plusieurs heures entre différentes grilles distribuées géographiquement requiert un

algorithme différent de celui qui répartit l'exécution des tâches entre les multiples processeurs d'une machine.

Le choix de l'algorithme de répartition de charge dépend des réponses données aux questions ci-dessus. En effet, chacun des algorithmes qui seront présentés dans ce chapitre est adapté à un environnement ou une application spécifique. Les travaux de [CK88, OA03] ont permis de classer les algorithmes existants en trois grandes familles : les algorithmes statiques, les algorithmes dynamiques et les algorithmes adaptatifs. De plus, chaque algorithme de ces trois familles peut être décomposé en quatre sous-stratégies distinctes : la stratégie d'initiation, la stratégie de localisation, la stratégie d'information et la stratégie de sélection. Cette classification a été reprise par la suite dans plusieurs travaux significatifs sur le sujet ([YS07, GPS11, GJTP12]). Cette typologie, ajoutée aux questions du paragraphe précédent permettent d'identifier ou concevoir l'algorithme de répartition le plus approprié selon le contexte d'utilisation.

Le principe d'optimisation de la localisation des données dans les systèmes distribués, bien qu'ancien, comme le rappelle A. Rotem-Al-Goz dans [RGO12], permet de limiter la latence des temps de réponses en réduisant le nombre de communications réseaux. Ce type d'optimisation s'est beaucoup développé ces dernières années avec l'émergence des nouveaux systèmes à large échelle tels que le framework Hadoop [DG04, Aba12, DSASSLP12] et les plateformes partagées d'infrastructure à la demande, dans lesquelles la latence réseaux est imprévisible et peut donc fortement influencer la performance du système.

Les plateformes émergentes et les services d'infrastructure à la demande permettent de louer des machines virtuelles à l'heure [Ama12] ou à la minute [1]. Le concept de coût, au sens financier, est devenu plus tangible pour les services logiciels utilisant ces infrastructures, car directement mesurable. Pour tirer partie de cette information il faut donc concevoir des stratégies de répartition de charge permettant d'optimiser le coût opérationnel d'un service donné.

Une description détaillée de la classification des différentes stratégies de répartition de charge existante est fournie en section 5.2 de ce chapitre. Les sections suivantes analysent différentes familles de cette typologie en commençant par les stratégies de répartition uti-

1. https ://cloud.google.com/pricing/compute-engine

lisant la charge comme indicateur, décrites en section 5.3. Les stratégies existantes fondées sur la localité des données sont décrites en section 5.4 et les stratégies fondées sur l'évaluation du coût sont décrites en section 5.5.

5.2 Classification et composition des algorithmes de répartition de charge

5.2.1 Familles d'algorithmes de répartition de charge

Les algorithmes de répartition de charge se divisent en général en trois grandes familles. Les algorithmes statiques, les algorithmes dynamiques et les algorithmes adaptatifs. Cette séparation a été détaillée dans les travaux de [GPS11].

Algorithmes statiques Un algorithme statique a connaissance de tous les paramètres nécessaires à la répartition des tâches avant son exécution, comme par exemple le nombre de machines ou de processeurs présents dans le système. Les algorithmes statiques typiques sont les algorithmes de planification : les durées et le nombre de ressources requises par chaque tâche sont connus a priori par l'algorithme. Un autre exemple est l'algorithme dit "Round Robin" (tourniquet) qui consiste à affecter les tâches à chaque machine de manière circulaire. Cet algorithme est considéré comme statique car son principe d'affectation des tâches ne change pas avec l'évolution du système. Le principal avantage de ces algorithmes est leur vitesse d'exécution rapide, qui leur permet de traiter un grand nombre de requêtes dans un intervalle de temps court. Ces algorithmes sont donc appropriés pour être exécutés à la volée sur un grand nombre de petites tâches, par exemple pour répartir des requêtes de services web [KR04].

Algorithmes dynamiques Les algorithmes dynamiques peuvent affecter de nouvelles ressources à des tâches en cours d'exécution. Concrètement, le système permet aux tâches en cours d'exécution d'être transférées d'une machine chargée à une machine moins chargée. Les algorithmes dynamiques sont particulièrement appropriés aux systèmes massivement parallèles à mémoire partagée, en raison de la facilité procurée par ces systèmes pour migrer une tâche en cours d'exécution. La migration dynamique de tâches dans d'autres types de

systèmes impose le développement de mécanismes spécifiques au niveau du système d'exploitation. L'exemple typique d'algorithme dynamique est celui de l'algorithme dit "central" développé par W.D. Hillis [Hil86] et décrit succinctement dans ceux de A. Osman [OA03]. Dans cet algorithme, la répartition s'effectue entre les différents processeurs d'une machine massivement parallèle. À intervalles réguliers, chaque processeur communique son taux d'utilisation à tous les autres processeurs de la machine, et se classe en fonction de sa charge dans une des trois catégories suivantes : IDLE (inoccupé),OVERLOADED (suchargé), OTHERS (ni surchargé, ni inutilisé). Suite à ce classement, les tâches précédemment affectées aux processeurs de la catégorie OVERLOADED sont affectées aux processeurs de la catégorie IDLE, de sorte qu'un minimum de processeurs restent surchargés à la fin de l'algorithme. De plus en plus, ces algorithmes dynamiques sont utilisés dans les plateformes de virtualisation à la demande, pouvant faire migrer des machines virtuelles en cours d'exécution d'une machine physique à une autre [BKB07].

Algorithmes adaptatifs Les algorithmes adaptatifs sont des algorithmes de répartition dont le nombre ou la nature des paramètres pris en compte pour allouer les ressources aux travaux change avec l'évolution du système. Nakrani et Randles [Nak04, RLTB10] ont conçu un système imitant le comportement des abeilles butineuses. Pour indiquer la quantité de ressources disponible dans une certaine zone géographique, les abeilles ouvrières exécutent une danse devant la ruche. Les autres abeilles ont une certaine probabilité de regarder l'abeille danser. Les ouvrières peuvent alors quitter leur tâche courante pour suivre l'abeille "danseuse". Les travaux de Randles et al. [RLTB10] appliquent ce principe à une ferme de serveurs. Les machines partagent un tableau d'affichage où elles indiquent après chaque requête le service exécuté et en combien de temps cette tâche a été réalisée. Chaque machine estime un score de "profitabilité" en fonction de la demande pour le service qu'elle opère. Si le score est suffisamment bas, la machine interroge le panneau d'affichage et sélectionne un service qui a besoin de nouvelles ressources parmi les services affichés. Après avoir consulté le tableau, une machine se reconfigure pour exécuter le service affiché si il est différent du sien.

Ces trois familles d'algorithmes font toutes appel à des stratégies spécifiques en fonction

de leur environnement ou du type de calcul à effectuer ou répartir. Ces stratégies ont d'abord été détaillées par A. Osman dans [OA03] pour décrire les algorithmes dynamiques, mais ce modèle est suffisamment générique pour englober les trois familles décrites ci-dessus.

5.2.2 Composition d'algorithmes de répartition de charge

Les algorithmes de répartition de charge résultent d'une combinaison de quatre stratégies principales, décrites par Osman et Hammar [OA03], et reprises dans plusieurs travaux depuis ([YS07, GPS11, GJTP12]).

Stratégie de déclenchement ou d'initiation : Cette stratégie détermine le moment auquel s'exécute l'algorithme de répartition. Deux approches se distinguent dans les choix possibles pour cette stratégie. Les stratégies de déclenchement périodique exécutent la répartition de charge à certains intervalles réguliers ou calculés dynamiquement. Par exemple, le système de stockage de fichier distribué HDFS [SKRC10], calcule périodiquement une nouvelle répartition des blocs de données qu'il stocke pour répartir équitablement les quantités sur les machines du système. La seconde approche consiste à déclencher l'exécution de l'algorithme de répartition en réponse à un événement comme l'arrivée d'une nouvelle requête sur le système. Ce modèle est fréquemment utilisé pour les grappes de serveurs web. Un événement de déclenchement peut aussi être signalé par le dépassement d'un certain seuil sur une métrique observée par le système. par exemple lorsque le taux d'utilisation du processeur dépasse 80%. Dans ce cas, la stratégie de déclenchement est fortement dépendante de la stratégie de sélection, et du système de surveillance des métriques utilisé.

Stratégie de localisation : Cette stratégie détermine quelle(s) machine(s) exécute(nt) l'algorithme. Il peut s'exécuter de manière centralisée : dans ce cas une machine du système est désignée comme répartiteur et exécute seule l'algorithme de répartition. L'avantage de ce modèle est la réduction du nombre de communications entre les machines et sa simplicité d'implémentation, mais cette approche présente le risque de voir la répartition de charge inopérante quand le répartiteur est sujet à une panne. À l'opposé, dans le cas distribué, plusieurs nœuds du système exécutent un algorithme de répartition de charge de

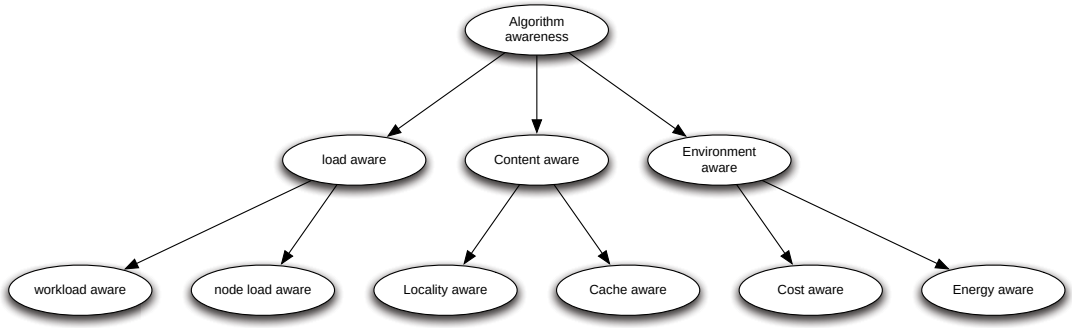


FIGURE 5.1 – Stratégies de sélection possibles [GPS11, GJTP12]

manière autonome. Cette exécution peut se faire de manière synchrone : toutes les machines exécutent l'algorithme de répartition au même moment (à la fin d'un travail par exemple) ou asynchrone. Dans ce cas chaque machine exécute l'algorithme en fonction de la stratégie de déclenchement choisie.

Stratégie d'information : Cette stratégie détermine comment l'information sur la charge est transmise entre les machines. Les machines peuvent transmettre l'information relevant de leur état ou de leur charge à un nœud central ou bien à quelques voisins déterminés au hasard ou selon des règles précises. La diffusion de rumeurs (gossiping), par exemple, est une stratégie d'information utilisée dans les systèmes pairs à pairs pour informer les voisins d'un nœud de l'état de ce dernier [BGPS05].

Stratégie de sélection : La décision de transférer une tâche précise à machine spécifique peut se fonder sur différents critères. Il est ainsi possible de classer les algorithmes de répartition en fonction des informations utilisées par le répartiteur pour sélectionner une machine à laquelle transférer la charge. La figure 5.1 représente cette classification. Les critères de sélection se répartissent entre trois familles : les informations sur la charge (*load aware*) qui peuvent être la charge totale de travail ou celle de chaque nœud. Les informations sur les propriétés et arguments de la tâche à exécuter (*content aware*), enfin il est possible de répartir la charge en fonction de facteurs externes qu'on désignera par la suite comme l'environnement (*environment aware*) de l' algorithme. Ces stratégies peuvent se composer entre elles pour permettre la sélection des machines sur la base de plusieurs

critères différents. De plus, un même critère peut être évalué de différentes manières : la charge d'une machine peut s'évaluer sur le taux d'utilisation du processeur, de la mémoire ou d'une composition de ces deux critères. Par exemple certaines stratégies utilisent une combinaison d'évaluation de la charge et d'optimisation de la localité.

5.3 Algorithmes de répartition par évaluation de la charge

Les algorithmes de répartition par évaluation de charge utilisent un ou plusieurs indicateurs pour mesurer l'occupation des ressources. Cela peut aller d'une interrogation régulière des machines pour connaître le taux d'utilisation Processeur [Hil86] à une combinaison de plusieurs indicateurs, comme une proportion entre le processeur, la bande passante disque et la mémoire, telle qu'utilisée par [YS07] pour obtenir une information précise sur l'occupation d'un ensemble de machines.

Une des stratégies les plus simples est le Join Shortest Queue (JSQ) mise au point par Bonomi [Bon90]. Dans cette stratégie, un routeur central envoie les tâches vers la machine qui a le moins de tâches en attente ou en cours d'exécution. La localisation centrale du répartiteur lui permet de maintenir la liste des requêtes en cours d'exécution.

Une amélioration de cette stratégie a été proposée par Lu et al.[LXK$^+$11]. Nommée "Join Idle Queue" (JIQ) cette stratégie est conçue pour fonctionner sur plusieurs répartiteurs à la fois (localisation distribuée). La problématique de répartition est découpée en deux parties : un problème d'association des tâches à des machines disponibles, et un problème d'association des machines à un répartiteur. Les répartiteurs maintiennent une file d'attente listant les processeurs qui n'ont pas de tâches en cours d'exécution et choisissent le premier processeur dans la file pour exécuter la prochaine tâche. Si la file est vide, le répartiteur envoie la tâche à un processeur au hasard. Lorsqu'une machine n'a plus de tâches à traiter, elle peut s'associer à un nouveau répartiteur en s'ajoutant à la liste des machines de ce dernier. La sélection du nouveau répartiteur peut se faire selon deux stratégies : soit au hasard, soit en choisissant le répartiteur qui a le moins de machines dans sa file, parmi un tirage aléatoire de répartiteurs. Ces deux techniques sont fondées sur les tâches en cours sur chaque machine ce qui ne donne qu'un aperçu succinct de la charge car cela suppose

l'homogénéité des différentes tâches.

Dans [ZRS+05], les auteurs présentent ADAPTLOAD, une stratégie de répartition évaluant en temps réel la taille des tâches en cours sur le système et adaptant la répartition de ces tâches au fil de l'exécution. Pour cela les auteurs partitionnent les requêtes en différents intervalles de tailles. La stratégie étant étudiée dans le contexte d'une grappe de serveurs web, la "taille" des requêtes est en réalité la taille du fichier demandé et transmis au client. Afin de répartir la charge de manière équitable entre les machines, l'algorithme crée à intervalles réguliers un histogramme des tailles des requêtes passées, puis alloue à chaque machine une portion des requêtes identifiées de sorte que la somme des tailles des requêtes soit équivalente pour toutes les machines du système. Cependant cet algorithme semble plus adapté pour servir des contenus statiques que dynamiques. Les auteurs ont donc modifié leur stratégies pour servir les requêtes dynamiques en suivant la stratégie JSQ, qui fournit de meilleures performances dans ce contexte. L'algorithme ADAPTLOAD a des caractéristiques similaires aux algorithmes de répartition utilisant la localité, en dépit du fait que ce concept ne soit pas explicitement utilisé dans l'algorithme.

5.4 Algorithmes de répartition fondés sur la localité

Le but de ces techniques est de limiter les latences réseau en évitant les communications et les transferts de données inutiles entre les machines du système. L'optimisation de la localité permet aussi de limiter les accès disques si le système stocke les données dans un cache. Ces techniques sont appelées "cache-aware" et sont de plus en plus utilisées dans de nombreux systèmes à large échelle. Trois approches sont possibles pour obtenir une répartition de ce type. Les techniques qualifiées ici d' "exhaustives" désignent les systèmes qui ont une connaissance exacte et explicite de la localisation des données. Ces systèmes sont souvent des systèmes de stockage de données comme les bases de données distribuées ou les systèmes de fichiers distribués. Les techniques fondées sur les tables de hachage regroupent un ensemble d'algorithmes utilisés dans les systèmes pairs à pairs pour localiser efficacement les données. Les algorithmes fondés sur les résumés ne possèdent qu'une connaissance approximative de l'emplacement des données et sont soumis à une certaine marge d'erreur dans la localisation des données.

5.4.1 Techniques exhaustives

Dans [PAB+98], les auteurs proposent un répartiteur qui envoie les requêtes concernant la même donnée au même nœud, en enregistrant pour chaque requête le nœud auquel elle est envoyée. La charge est équilibrée en évaluant le nombre de requêtes en cours sur chaque nœud du système. Chaque requête est associée à une liste de serveurs potentiels et le moins chargé de la liste est utilisé pour traiter la requête suivante. Lorsque tous les serveurs de la liste sont surchargés, alors un serveur est ajouté à cette liste. Cette technique présente l'avantage de procurer une répartition très équilibrée de la charge sur tous les serveurs. L'algorithme définit cependant statiquement un nombre maximum de requêtes à affecter pour toutes les machines du système, et ne considère donc pas que des machines de différentes capacités puissent être utilisées dans le système. Le second désavantage de cette approche est la consommation mémoire nécessaire pour enregistrer la machine cible de chaque requête.

Le système Hadoop [Whi09] est une implémentation open-source du système décrit dans les travaux de [DG04]. Hadoop est un framework de traitement de données distribué, permettant de simplifier le travail du programmeur en le déchargeant de la gestion de la distribution des traitements, ce qui permet de se concentrer uniquement sur les aspects de manipulation des données. Hadoop utilise conjointement deux composants : le premier est Hadoop Distributed File System (HDFS, [SKRC10]) un système de fichiers distribué permettant de stocker et répartir des données de grande taille sur plusieurs machines. Le second composant est le framework Map/Reduce lui même, permettant de gérer la répartition des tâches de traitement sur les données. Cette répartition est rendue possible par l'interrogation du registre des fichiers qui possède la localisation exacte de toutes les données stockées dans HDFS. Dès lors, au moment de l'exécution d'un travail, les tâches sont réparties sur les fichiers cibles en interrogeant le NameNode. L'avantage de cette technique est sa grande précision dans la répartition des tâches. Cependant, elle présente deux désavantages. Premièrement les données accédées ne sont peut être pas réparties équitablement sur le système de fichiers distribué. Ceci entraine un déséquilibre dans la répartition des tâches, qui peut augmenter le temps nécessaire pour terminer l'ensemble des tâches en cours. Deuxièmement dans le cadre de ce canevas, les tâches à effectuer sont

directement associées avec les données à traiter : l'utilisateur fournit le code à exécuter et la liste des fichiers sur lesquels le traitement s'effectue. Cependant pour bénéficier de la localité des données il est nécessaire de respecter scrupuleusement l'interface de programmation fournie par le logiciel.

5.4.2 Techniques de hachage

Évoquée dans les travaux de [PAB+98], une technique de répartition simple consiste à partitionner les ressources du système à l'aide d'une fonction de hachage. Les ressources nécessaires à l'accomplissement d'une tâche sont alors identifiées par une clef et réparties sur les différentes machines du système, via une opération de modulo entre la clef et le nombre total de partitions. Le résultat de cette opération donne l'identifiant de la partition à laquelle la ressource appartient. Le désavantage de cette technique est qu'elle n'est efficace que si chaque tâche n'accède qu'à une seule ressource à la fois, autrement il est nécessaire d'implémenter des opérations parallèles. Deuxièmement, si les accès aux ressources ne sont pas uniformes, une des partitions présentera le risque d'être plus fréquemment accédée que les autres et la charge ne sera donc pas répartie équitablement entre les nœuds du système. Une solution possible pour palier à ce défaut consiste à répliquer les ressources ou les partitions sur plusieurs machines différentes de manière dynamique [BKB98], afin de diviser la charge progressivement avec l'utilisation du système. Depuis ces travaux, les systèmes sont devenus de plus en plus dynamiques, fournissant la possibilité d'ajouter ou supprimer des machines à la demande. Le partitionnement par fonction de hachage montre alors une limite, car l'ajout ou la suppression d'une machine du système modifie les résultats de l'opération de modulo, et l'information de localisation des requêtes est alors perdue.

Pour palier à ce problème, Karger et al. [KLL+97] ont introduit le principe du hachage cohérent (*consistent hashing*). Le principe repose lui aussi sur l'utilisation des fonctions de hachage. Chaque partition est identifiée par une clef donnée par une fonction de hachage. Les ressources sont ensuite identifiées par la même fonction de hachage et insérées dans la partition qui a l'identifiant supérieur le plus proche. La métaphore habituellement utilisée pour représenter cette technique est celle du cercle où chaque identifiant correspond à un point quelconque du cercle. Pour obtenir la partition correspondant à une donnée, son

identifiant est haché et la liste des identifiants est ensuite parcourue dans l'ordre croissant des clés jusqu'à trouver le premier identifiant correspondant à une partition. Lorsqu'une machine est ajoutée ou enlevée du système, elle est associée à un certain nombre de partitions. Seule la portion des ressources du système se trouvant nouvellement associée à cette machine sera affectée par le changement. Cette technique a été adoptée dans une grande variété de systèmes de stockage [LM09] et de cache [2], pour lesquels elle est particulièrement appropriée. Un autre avantage de cette technique est la possibilité de pouvoir exécuter l'algorithme depuis n'importe quelle machine à condition que les identifiants des partitions lui soient communiqués. La problématique du biais dans la distribution des requêtes n'est cependant toujours pas réglée par ces techniques [YL11].

5.4.3 Techniques de résumé de contenus

La troisième approche utilisée pour optimiser la localité des données dans un système distribué consiste à utiliser des techniques de résumé de contenu. Il s'agit de trouver des représentations compactes des ressources disponibles sur les machines du système. Cependant, ces représentations compactes nécessitent de réduire leur précision. Il est donc nécessaire de prévoir l'utilisation de techniques visant à corriger les éventuelles erreurs d'allocation des tâches.

Les filtres de Bloom [Blo70] ou des techniques similaires sont régulièrement employés pour produire des résumés compacts de grands ensembles de données. Les filtres de comptage sont une extension des filtres de Bloom qui ont été décrits par Fan et al. [FCAB98]. Un filtre de comptage est un tableau de petits compteurs (4 bits), qui permet de résumer un ensemble de données. Lorsqu'un élément est ajouté ou supprimé de l'ensemble, plusieurs fonctions de hachage sont appliquées à l'élément. Chaque clef obtenue désigne un compteur du tableau. Les compteurs ainsi désignés sont incrémentés lors de l'insertion d'un élément dans le filtre et décrémentés lors de la suppression. Cette structure peut ensuite être interrogée pour déterminer la présence d'un élément dans l'ensemble. Une description détaillée du fonctionnement de cette structure de données est fournie au chapitre 6.2. Dans le système décrit par [FCAB98], chaque machine du système maintient un filtre de comptage pour

2. `http//memcached.org`, consulté le 5 octobre 2013

résumer le contenu de sa mémoire cache, et transmet régulièrement cette représentation à ses voisins. Chaque nœud exécute la répartition lorsqu'une requête pour une ressource arrive sur une machine, elle examine en priorité son filtre local, puis si la donnée est absente de son cache, l'algorithme examine les résumés des caches voisins. Si la donnée est trouvée dans un des filtres des voisins, elle est recopiée dans le cache local par le serveur puis le résultat est renvoyé au client. Ce protocole est donc utile pour implémenter un service de cache coopératif. Cette approche échoue cependant à prendre en considération des tâches nécessitant l'accès à un certain nombre de ressources.

[DSSASLP08] ont démontré qu'un protocole d'échange de résumés de caches peut être efficacement utilisé pour la répartition de charge. Les Résumés Evolutifs avec Compteurs (Evolutive Summary Counters, ESC) décrits par ces travaux, consistent en une structure de données utilisant une liste chainée de filtres de bloom avec compteurs (FBC). Un ESC est tenu à jour par chaque machine du système. Dans chaque ESC, un premier FBC enregistre la liste des données accédées localement sur la machine. Lorsque ce FBC atteint sa capacité de stockage maximale, il est remplacé par le FBC suivant dans la liste. Ceci permet au système de conserver un historique des requêtes passées dans les filtres, de sorte que les données ne sont insérées dans le nouveau filtre que si elles sont absentes des autres filtres de la liste. A intervalles réguliers, un résumé des FBC de chaque machine est envoyé à toutes les machines du système. Le fait de conserver plusieurs FBC dans une liste chainée permet au système de connaître les données qui ont été accédées sur la machine mais qui ne sont plus dans le cache. Les auteurs ont comparé deux algorithmes de répartition utilisant leur structure de résumé.

Le premier, appelé Coût Probable (Probability Cost) calcule à chaque requête et pour chaque nœud, le coût probable en utilisation CPU et en nombre d'Entrées Sorties disques, en fonction de l'historique des requêtes. La requête est ensuite envoyée au nœud sur lequel le coût est le moins élevé.

La seconde approche ajoute aux calculs précédents un facteur d'affinité qui mesure la probabilité qu'un document demandé par la requête soit dans le cache de la machine, en fonction de la liste des accès récents obtenue depuis les résumés des machines. Ces deux approches combinées à l'utilisation des résumés ont montré de fortes améliorations dans

les temps de réponses observés, quelle que soit la distribution des requêtes.

L'algorithme de répartition utilisé dans cette approche parcourt toute la liste des machines à chaque exécution d'une requête. Cette solution est supportable tant que le nombre de machines du système ne dépasse pas un certain seuil. Une autre limite de cette approche est la spécificité du système décrit, qui est un système d'analyse de requêtes en langage naturel. Les mesures effectuées pour localiser les documents concernés dans le systèmes sont difficilement généralisables à d'autres environnements. Il doit être possible de trouver un juste milieu entre l'optimisation de l'utilisation des ressources et la vitesse d'exécution de l'algorithme de répartition de charge, car de plus en plus d'informations sont aujourd'hui disponibles et observables sur les machines. Par exemple, les nouvelles plateformes d'infrastructure à la demande permettent de prendre en compte de nouveaux indicateurs de performance comme le coût.

5.5 Stratégies basées sur le coût

L'introduction de services informatiques à la demande, et facturés à l'utilisation (cloud computing), fait émerger une problématique de coût opérationnel dans les systèmes déployés sur ces plateformes. Ainsi, de nombreux travaux de recherche se concentrent sur la mesure et l'optimisation des coûts d'exécution dans ces environnements.

Dans [AaC09] les auteurs évaluent le coût (monétaire) d'allocation de différents traitements parallèles sur un ensemble de machines virtuelles réparties entre une plateforme de virtualisation locale et un "cloud" de type infrastructure à la demande (en l'occurrence l'*Amazon Elastic Compute Cloud*, EC2 [Ama12]). L'allocation d'une machine virtuelle sur ce type de plateforme a un coût fixe en fonction du temps d'utilisation. Les machines virtuelles allouées sur la plateforme EC2 sont facturées à l'heure, arrondie à l'heure suivante. Par conséquent, si une machine est allouée mais n'est utilisée que pendant quelques minutes, l'opérateur n'a pas une allocation optimale de son budget. La planification des allocations de machines est donc particulièrement sensible dans la mesure où son impact sur le coût de l'infrastructure se ressent immédiatement. Les auteurs ont donc évalué différentes stratégies de planification des tâches permettant d'optimiser l'utilisation des ressources disponibles

dans le système utilisé. Le planificateur maintient la liste des travaux en attente, et fournit à chaque travail un créneau horaire pour s'exécuter selon la règle du premier arrivé, premier servi. Le graphique en figure 5.2 montre ce qui peut arriver à la suite d'une telle allocation. Les rectangles blancs symbolisent trois travaux en cours de réservation (J1, J2 et J3), caractérisés par deux chiffres : le nombre de processeurs à utiliser et le temps pendant lequel les processeurs seront occupés. Le travail J1, par exemple, utilise trois processeurs pendants deux heures. Il peut arriver qu'une allocation naïve des tâches fasse apparaître des trous dans l'occupation des machines (rectangle gris). Si le cas se présente, le planificateur réordonne les tâches en fonction de leurs besoins respectifs en ressources, en faisant remonter dans la liste les tâches qui peuvent remplir ces trous. Ce ré-ordonnancement peut

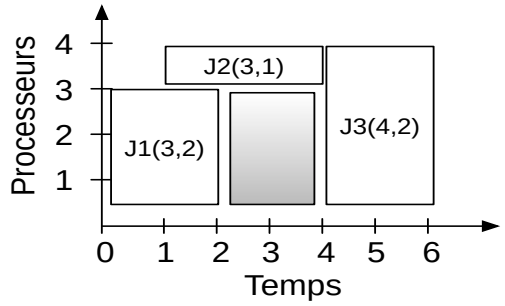


FIGURE 5.2 – Graphique d'ordonnancement, d'après Assunçao et al. [AaC09]

être plus agressif, dans la mesure où les auteurs permettent à l'ordonnanceur de repousser l'exécution de certaines tâches, tant que cela ne viole pas l'heure de fin attendue pour l'exécution des tâches repoussées. La comparaison de différentes techniques d'ordonnancement montre que les stratégies de ré-ordonnancement agressives permettent une meilleure utilisation globale du système.

Les travaux de Ming et Marty [MH11] se concentrent sur l'exécution de travaux complexes (workflow) sur une infrastructure de type Cloud. Le but du système est de permettre l'exécution de tâches intensives réparties sur plusieurs machines en respectant des contraintes horaires strictes. Pour cela les auteurs représentent chaque unité de travail comme un graphe orienté acyclique de tâches à réaliser accompagné d'un horaire de fin. Chaque tâche de ce graphe fait appel à un service déployé sur une machine virtuelle dans le cloud. Le système proposé fonctionne comme une boucle exécutée à intervalles réguliers.

À chaque itération de la boucle, le système établit une planification des tâches et des allocations de nouvelles machines, de sorte que le coût total d'exécution soit minimal. Une contrainte supplémentaire est que chaque unité de travail doit être complétée avant son horaire de fin. Pour chaque unité de travail, un horaire de fin est estimé pour chaque tâche du graphe, et chaque tâche est assignée à la machine sur laquelle elle s'exécute le plus économiquement. Il n'est cependant pas toujours optimal d'affecter les tâches à la machine sur laquelle elles s'exécutent le plus rapidement. Pour optimiser le coût, deux mécanismes de consolidation sont appliqués par les auteurs : premièrement lorsque certaines tâches peuvent être exécutées en parallèle, le système les exécute l'une après l'autre si la somme de leurs temps d'exécution est inférieure à l'unité de temps de location des machines. En effet, exécuter en parallèle trois tâches de 20 minutes sur trois machines différentes coûte plus cher que de les exécuter à la suite sur la même machine. De plus, les 40 minutes restantes sur les machines ainsi allouées ne seraient pas forcément utilisées. Cette optimisation est faite si elle n'a pas de conséquence sur l'horaire de fin de l'unité de travail. La seconde optimisation consiste à grouper des tâches sur des machines moins efficaces, afin d'éviter d'allouer des machines plus coûteuses. Cette approche tire donc partie de la subdivision des tâches dans le cadre de traitements de données intensifs, mais impose de connaître à l'avance les types de tâches et de services utilisés dans l'ensemble du système.

Dans des échelles de coûts plus réduites, les travaux de Rogers et al. [RPC10] visent à fournir une estimation des coûts de différents types de charge de travail pour une base de données distribuée dans un environnement Cloud. L'approche consiste à exécuter un ensemble de séries de requêtes SQL types sur les machines du système puis d'estimer le coût d'exécution de ces requêtes sur chaque machine. Les auteurs utilisent ensuite un programme d'optimisation linéaire minimisant le coût total d'exécution pour obtenir l'affectation des requêtes à chaque machine. Le coût des requêtes est évalué de deux manières. La première approche, dite "boite noire", mesure pour chaque série de requêtes envoyée au système le coût de chaque requête par rapport à son temps d'exécution. La seconde approche dite "boite blanche" quantifie avec précision les ressources disponibles pour chaque type de machine dans le système et évalue les besoins des requêtes en les exécutant dans un outil de profilage. Cette approche semble cependant difficile à mettre en pratique dans un système

applicatif. Le profilage des requêtes pour chaque type de machine peut se révéler prohibitif, et devra être refait à chaque mise à jour du système si les requêtes changent. Cependant, cette approche montre qu'il est possible d'utiliser l'optimisation des coûts pour répartir des requêtes SQL, qui ne sont pas des traitements parallèles lourds comme les traitements décrits dans les approches précédemment citées.

5.6 Discussion

La recherche en matière de stratégies de répartition de charge pour les systèmes distribués de traitement de données se concentre actuellement sur la combinaison des stratégies évaluant la charge et les stratégies optimisant la localité. Cependant, l'émergence des plateformes de cloud computing, de plus en plus utilisées pour ce type de système, nécessite de prendre en considération la notion de tarif pour permettre une allocation efficace des ressources en fonction de leur coût, et bénéficier au mieux des économies potentiellement fournies par ce type d'infrastructures.

Les stratégies se reposant seulement sur la localisation des données ne sont pas suffisantes pour parvenir à une bonne répartition de la charge, il est nécessaire de combiner ces approches avec les stratégies utilisant les informations sur l'occupation des machines.

De plus, les techniques de résumés de données peuvent être améliorées au niveau de la recherche de la machine cible, qui est rarement optimisée dans ces stratégies et passe par une énumération de l'ensemble des machines. Un juste équilibre doit être trouvé entre la vitesse d'exécution de la stratégie de répartition de charge et son efficacité pratique, de manière à garantir un débit important au niveau du répartiteur.

Par exemple, les stratégies fondées sur les filtres de Bloom décrites en section 5.4 et mises au point par Dominguez-Sal et al. [DSASSLP12], possèdent une complexité bien supérieure aux algorithmes de type Hachage Cohérent décrits dans les travaux de Karger [KLL+97, KR04]. Il est donc légitime de s'interroger sur les améliorations possibles de ce type d'algorithmes de répartition. La deuxième conclusion de ce travail, est que les travaux sur la répartition de charge utilisant l'information de coût monétaire, considèrent le coût comme une ressource. Or dans les systèmes de Cloud Computing existants aujour-

d'hui, l'information de coût est publiquement disponible et il est donc possible de mesurer avec précision le cout d'exécution d'un programme afin de s'en servir comme indicateur de performance. Les travaux détaillés dans le reste de ce document montrent le développement de deux nouvelles stratégies de répartition de charge, respectivement fondées sur la localisation des données (chapitre 6) et sur l'information tarifaire (chapitre 7).

Chapitre 6

WACA : Un Algorithme Renseigné sur la Charge et le Cache

6.1 Contexte et Motivation

Le domaine des traitements de données massives, notamment les données issues de réseaux de capteurs telles que celles utilisées dans le projet MCube (cf. chapitre 2), nécessite le croisement de données précises issues de larges fichiers peu structurés et dépourvus d'indexation. La mise en mémoire principale de ces données produit une nette amélioration des temps de traitements. La réduction du nombre de lectures sur disques, et donc l'utilisation efficace de la hiérarchie des mémoires est une des clés du passage à l'échelle. Une utilisation efficace de la mémoire cache permet en effet de réduire la contention sur les disques et par conséquent de réduire le temps de traitement des données. Cette technique est par exemple utilisée dans de nombreux systèmes d'analyses ou de stockage de données distribués "en mémoire" tels que le système Spark ([ZCF+10]).

6.1.1 Localité des données pour les traitements distribués

La localisation des données dans un système distribué, est l'emplacement des données dans le système, c'est-à-dire sur quelle(s) machine(s) du système les données sont-elles stockées. À l'heure de l'utilisation croissante des plateformes d'infrastructures "dans les nuages" (le "cloud"), dans lesquelles l'utilisateur ne maîtrise plus l'implantation physique des machines, et où les délais de communication dépendent de l'utilisation de la plateforme,

déterminer rapidement la localisation des données est un atout permettant de gagner un temps précieux. En dehors du domaine des bases de données, la gestion du cache par le système d'exploitation peut produire des effets de bord bénéfiques pour certains types d'applications. Dans [PLCKA11], un service web de traitement de données météorologiques est étudié. Le service utilise des grilles de mesures fournies par des partenaires tel que le service Météo France et génère à la demande des prévisions météorologiques pour des coordonnées géographiques précises. Le temps de calcul de ces prévisions est fortement corrélé au nombre de jours de prévision demandés. Ce nombre de jours correspond au nombre de fichiers de données à traiter pour calculer la prévision. La figure 6.1, montre une séquence d'appels à ce service web, numérotés de 1 à 11 sur l'axe des abscisses. Pour chacun de ces appels successifs la courbe en pointillés compte le nombre d'accès disques mesurés pendant cet appel sur l'axe de droite et la courbe pleine montre le temps d'exécution observé sur l'axe de gauche. Le temps de traitement apparait réduit lorsqu'il y a peu d'accès disques (appels 3 à 6 sur le graphique), et remonte lorsque la requête requiert à nouveau des accès disque (appels 2,7 et 8 sur le graphique). Pour ce type de services, la localisation des

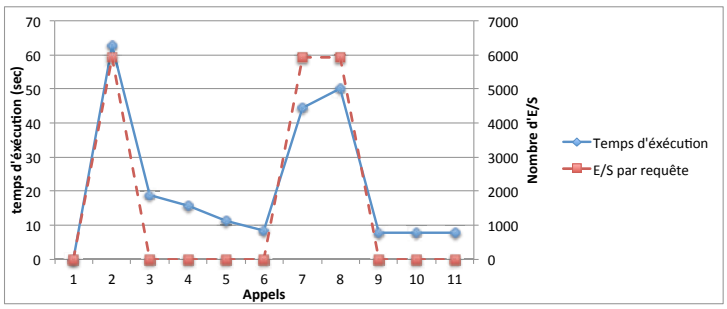


FIGURE 6.1 – Relation entre nombre d'entrées/sorties et temps d'éxécution

données en mémoire est donc un facteur clef de la performance. C'est pourquoi de nombreux systèmes distribués d'analyses de données massives tels que Hadoop ([Whi09, SKRC10]) ou Spark ([ZCF+10]) optimisent les accès disques et mémoire pour fournir des traitements plus rapides à leurs utilisateurs. Ces approches ont besoin d'algorithmes de répartition spécifiques remplissant trois conditions :

Résumer le contenu des noeuds des systèmes : Comment maintenir une information correcte sur la localité des données de manière compacte et fiable sans sacrifier l'effi-

cacité du système. Par exemple le système HDFS ([SKRC10]) est conçu pour contenir un nombre important de gros fichiers, mais si le même volume de données est divisé en un plus grand nombre de petits fichiers, alors le système commence à perdre en efficacité.

Maintenir une charge équilibrée sur les machines : Certaines charges de travail présentent des biais, c'est-à-dire qu'une fraction minime des données doit être accédée par la majorité des traitements. Dans ce cas, une stratégie de répartition naïve présentera un risque de surcharger une machine qui stocke les données appartenant à cette fraction majoritaire.

Prendre une décision pour l'affectation des requêtes rapidement : Un algorithme de répartition de charge dans le cadre de services de traitement de données à la demande doit traiter rapidement les requêtes de services de manière à pouvoir répartir ces dernières sur un nombre important de machines.

Dans le cadre de ces travaux, l'objectif consistait à mettre au point une stratégie de répartition qui permet d'affecter des requêtes de traitements à la machine qui a le plus de chances d'avoir les données dans son cache système. L'algorithme doit être générique, de manière à pouvoir être utilisé par un grand nombre de services de manière transparente. Il faut donc pouvoir construire des résumés de requêtes de services "boites noires" dont la structure et le fonctionnement sont inconnus, excepté le fait qu'ils suivent le principe REST [Fie00]. Il est aussi nécessaire d'établir une correspondance entre les requêtes reçues et les données reçues. Le nombre de ces requêtes est potentiellement grand, il faut donc identifier rapidement quel nœud a reçu une requête similaire dans le passé. Pour cela, l'algorithme mis au point au cours de ces travaux, appelé Workload And Cache Aware algorithm (WACA, Algorithme Renseigné sur la Charge et le Cache), utilise différentes variantes d'une structure de données appelée filtres de Bloom ([Blo70]) pour créer un résumé des requêtes envoyées à chaque machine. Le principal avantage des filtres de Bloom comme technique de résumé est sa facilité d'implémentation et sa complexité $O(1)$ pour la recherche d'élément. La plateforme d'application cible étant basée sur les services web, il est possible d'utiliser les URLs pour identifier les requêtes à résumer. Ceci ne nécessite aucune connaissance de la structure interne des applications web déployées.

Le besoin d'optimiser les accès disques via les caches systèmes est particulièrement intéressant pour les applications de traitement de données massives mais aussi dans le contexte du Cloud Computing, car les latences réseaux sur les plateformes d'informatique à la demande sont en général peu maitrisées de par la nature partagée de la plateforme [IOY+11, BS10].

Différentes variantes de filtres de Bloom sont décrites en section 6.2 de ce chapitre. L'algorithme WACA, dont le principe général est décrit en section 6.3, se décline en trois versions, la première version de l'algorithme (WACA 1) utilise des filtres de Bloom avec Compteurs (FBC). Cette première version a permis d'établir les bases des règles de dimensionnement des filtres utilisées par les deux versions suivantes, et son fonctionnement ainsi que sa comparaison avec d'autres approches sont fournies en section 6.4. La seconde version de WACA, appelée WACA-History ajoute un historique des requêtes envoyées à chaque nœud afin de répartir au mieux les données en mémoire. Cette approche est étudiée en section 6.5. La troisième version de l'algorithme intitulée Block-Based WACA (BB-WACA, section 6.6), est destinée à optimiser le fonctionnement de la stratégie WACA lorsque le nombre de machines dans le système augmente significativement, en ajoutant un index aux filtres de Bloom afin de réduire le temps de sélection de la machine cible. L'évaluation des performances de ces politiques dans différents environnements montre que la politique de répartition de charge WACA permet de réduire les temps de traitements de requêtes de services.

6.2 Filtres de Bloom

6.2.1 Filtres de Bloom Standards (FBS)

Un filtre de Bloom, décrit par B.H Bloom dans [Blo70], est une structure de données probabilistique très compacte qui permet de tester de manière rapide l'appartenance d'un élément à un ensemble de données. Cette efficacité en mémoire se traduit par la probabilité de *faux positifs*, c'est-à-dire qu'il existe une probabilité quantifiable qu'un élément soit considéré comme appartenant à l'ensemble alors qu'il n'y est effectivement pas. En revanche, il n'est pas possible d'avoir de faux négatifs. Le filtre de Bloom consiste en deux

composantes principales : un ensemble k de fonctions de hachage et un vecteur de bits d'une taille m fixée. Une opération de modulo est appliquée au résultat de chaque fonction de hachage, de sorte qu'il corresponde à la taille du vecteur. Par exemple, si le vecteur de bits est de taille 200, alors toutes les fonctions de hachage doivent retourner un nombre entre 0 et 199, grâce à l'application du modulo 200. Les fonctions de hachage sélectionnées garantissent que les adresses générées sont réparties de façon équiprobable sur toutes les valeurs possibles.

Pour introduire un élément dans un filtre de Bloom, on lui applique les différentes fonctions de hachage sélectionnées. Pour chaque valeur obtenue, le bit correspondant dans le vecteur est activé si il est égal à 0. Pour tester l'appartenance d'un élément e à un ensemble S d'éléments introduits dans le filtre de Bloom, on lui applique les fonctions de hachage. Si au moins un des bits est à 0 alors l'élément e n'appartient pas à S. Par contre, si tous les bits sont à 1 alors e appartient à S avec une certaine probabilité, la probabilité de *faux positif*. Ce taux évolue avec le nombre n d'insertions dans le filtre, d'après la relation suivante ([FCAB98]) :

$$f = (1 - e^{-kn/m})^k \tag{6.1}$$

m étant la taille du vecteur du filtre de Bloom, n le nombre d'éléments indexés et k le nombre de fonctions de hachage. Pour minimiser ce taux, il est possible de choisir le nombre optimal de fonctions de hachage à utiliser à partir du rapport entre le nombre d'éléments insérés n et la taille souhaitée du filtre en utilisant la formule :

$$k = ln(2) \times (m/n) \tag{6.2}$$

En raison de sa compacité et de sa rapidité de réponse, le filtre de Bloom est une structure idéale pour fournir des résumés. Mais un résumé doit être tenu à jour : si des éléments sont supprimés de l'ensemble, ce changement doit être répercuté sur le résumé. Or, les FBS ne supportent pas la suppression d'un élément : il se peut en effet que la même position dans le filtre soit utilisée pour représenter plusieurs éléments de sorte que la remise à zéro d'une position donnée risque de faire perdre l'information concernant d'autres éléments du filtre. Les travaux de [FCAB98] fournissent une solution à ce problème avec la mise au point des filtres de Bloom avec Compteurs (FBC).

6.2.2 Filtres de Bloom avec compteurs (FBC)

Le FBC, décrit dans [FCAB98] est une extension du filtre de Bloom standard qui fournit la possibilité de supprimer des éléments du filtre. Le vecteur de bits y est remplacé par un tableau d'entiers,où chaque position est utilisée comme compteur. Les formules de calcul du taux de faux positifs 6.1 et du nombre de fonctions de hachage k (6.2) s'appliquent. L'insertion d'un élément est réalisée en incrémentant de 1 les entiers aux positions renvoyées par les fonctions de hachage. Le retrait est réalisé en décrémentant de 1 ces entiers. La question d'appartenance d'un élément au filtre est traitée en regardant si tous les entiers aux positions renvoyées par les k fonctions de hachage sont strictement positifs. Ainsi les filtres avec compteurs permettent de tenir les filtres à jour au prix d'un espace mémoire plus important. Il a été montré par les auteurs de [FCAB98] que limiter la taille des compteurs à 4 bits fournit un bon compromis entre précision et espace mémoire.

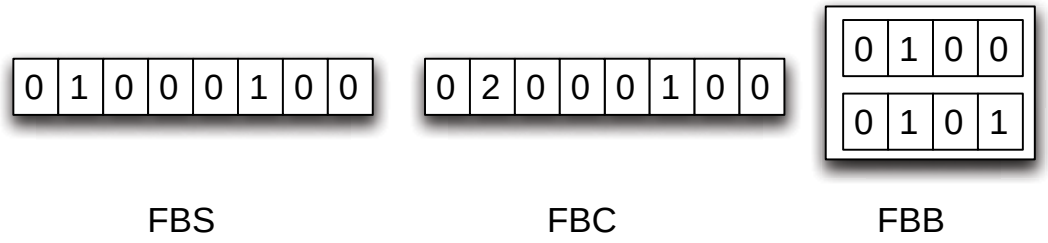


FIGURE 6.2 – Exemples de Filtres de Bloom

6.2.3 Filtres de Bloom en Blocs (FBB)

Un filtre de Bloom en blocs (FBB), est un FBS dont le vecteur est divisé en k tranches ou blocs de tailles égales. Chaque bloc est associé à une des k fonctions de hachage du filtre, et chacune des k fonctions de hachage indiquera une position au sein de son bloc respectif. L'avantage de cette structure est d'augmenter l'utilisation des bits au sein du filtre par rapport aux filtres standards et de faciliter le calcul des taux de faux positifs ([ABPH07]). Le travail de [ABPH07] montre que le maximum d'objets n insérables dans un bloc d'un FBB, sans faire augmenter la probabilité de faux positifs, dépend du taux de remplissage p et de la taille m du bloc.

$$n \approx -m \ln(1-p) \tag{6.3}$$

Dans ces mêmes travaux, les auteurs démontrent que le taux de remplissage optimal d'un bloc de FBB est $p = 1/2$. Sous cette hypothèse, il est possible d'utiliser la relation 6.3 pour estimer la taille m du bloc en fonction du nombre d'objets voulus avec le calcul :

$$m \approx \frac{n}{-\ln(1/2)} \tag{6.4}$$

Il est donc possible d'obtenir la taille totale du filtre M par la relation $M = km$ où k est le nombre de fonctions de hachage à utiliser. Cette valeur ne dépend alors que du taux de faux positifs f souhaité par l'utilisateur, d'après la relation suivante :

$$k \approx \log 2(\frac{1}{f}) \tag{6.5}$$

Pour illustrer la différence entre ces trois versions de la même structure, le schéma 6.2 montre l'état après trois insertions d'un filtre de taille $M = 8$ avec $k = 2$ fonctions de hachage, pour chacune des structures décrites ci-dessus.

6.3 Principe Général de l'algorithme WACA

L'algorithme WACA est un algorithme statique centralisé initié par l'envoyeur. Il a été développé dans le contexte des services web et s'appuie sur une architecture composée d'un répartiteur qui transmet les requêtes à un ensemble de serveurs d'applications. La figure 6.3 montre une vue abstraite de la stratégie proposée. Cet algorithme est déclenché chaque fois qu'une requête est reçue. Entre les requêtes, l'algorithme maintient la liste des machines disponibles et trois structures de données sont attachées à chaque machine : Un filtre de Bloom avec compteur, tel que décrit en section 6.2, une File et un Compteur de Requêtes. Le filtre de Bloom est utilisé pour tracer les requêtes exécutées par chaque noeud (il correspond à l'historique du contenu des requêtes), la File permet de tenir le filtre à jour par rapport aux évolutions des requêtes, et le compteur de requêtes est utilisé pour identifier la machine la moins chargée.

Dès qu'une nouvelle requête arrive, l'algorithme reçoit la liste des machines disponibles, et examine le filtre de bloom de chaque nœud afin de vérifier si il a déjà reçu cette requête. Parmi les machines ayant déjà reçu la requête, l'algorithme choisit la moins chargée. Dans

le cas où aucune machine n'a enregistré la requête dans son filtre, c'est alors le nœud le moins chargé de toutes les machines qui sera sélectionné comme cible.

Comme son nom l'indique, l'algorithme WACA prend aussi en compte la charge des machines, c'est à dire la quantité de travail en cours d'exécution sur chaque serveur du système. Dans le contexte des services web, il est courant d'utiliser un compteurs de requêtes comme évaluateur de la charge. Cette définition de la charge ne donne qu'une estimation partielle de la charge réelle de la machine, mais l'algorithme WACA peut être dérivé de manière à utiliser d'autres indicateurs tels que le niveau d'utilisation mémoire ou processeur.

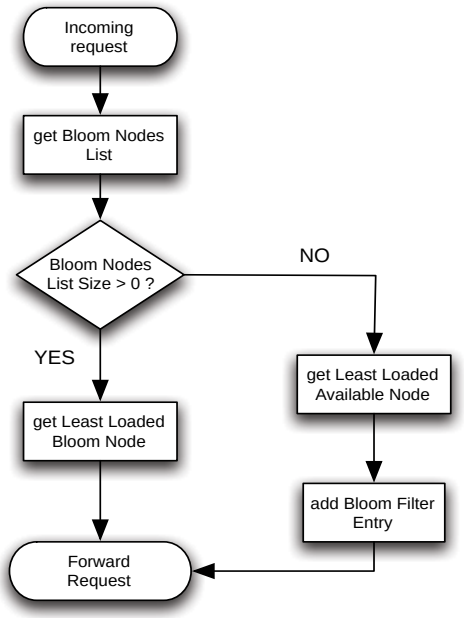


FIGURE 6.3 – Description de l'algorithme

6.3.1 Dimensionnement des filtres

Comme la taille du cache est limitée, l'algorithme doit maintenir à jour les filtres des nœuds en ne gardant que les entrées les plus récentes. Cette suppression doit permettre de conserver une faible probabilité de faux positif, car plus il y a de bits activés dans le filtre, plus la probabilité d'obtenir des faux positifs augmente. Ainsi, un seuil, appelé la Capacité C_m, est associé à chaque noeud, représentant le nombre maximum d'entrées que le cache

de la machine est supposé contenir. Cette valeur est calculée à partir de l'équation 6.6, en divisant la mémoire disponible (M_m) de la machine m par une estimation de la quantité de données mise en mémoire par l'exécution d'une requête test S_r.

$$C_m \approx \frac{M_m}{S_r} \tag{6.6}$$

La *Capacité* est la valeur qui permet de déterminer, en fonction du taux de faux positifs souhaité par l'utilisateur et des formules citées en section 6.2, le nombre k de fonctions de hachage optimal à utiliser pour dimensionner le filtre d'une machine spécifiée. Cependant, le taux de faux positifs possible augmente avec le nombre d'éléments présents dans le filtre. Il est donc nécessaire de maintenir le taux de faux positifs à la valeur configurée par l'utilisateur. Pour cela, une File d'attente est associée à chaque filtre. La taille Q_m de cette file est égale à la *Capacité* multipliée par le nombre k de fonctions de hachage utilisées dans le filtre :

$$Q_m = C_m \times k \tag{6.7}$$

Dès qu'une donnée est insérée dans le filtre, les positions insérées sont ajoutées à la fin de la file. Lorsque le nombre d'éléments insérés dans le filtre atteint ou dépasse la capacité de ce filtre, les k plus vieilles positions enregistrées dans la file sont retirées, et les positions correspondantes dans le filtre décrémentées, ce qui équivaut à une suppression. Ce mécanisme permet de maintenir la probabilité de faux positif du Filtre de Bloom avec Compteur constante en supprimant les entrées trop vieilles du filtre. Cette approche permet de conserver la liste des positions récemment mises à jour, même si les fonctions de hachage utilisées n'ont aucune relations entre elles.

6.3.2 Choix de la fonction de hachage

L'efficacité d'un filtre de Bloom repose sur le nombre de collisions possibles lors du hachage des éléments insérés. Si la fonction de hachage utilisée produit un grand nombre de collisions alors la probabilité de faux positifs augmentera sensiblement. Il est donc important de considérer les choix possibles en matière de fonctions de hachage. Deux algorithmes connus sont le Message Digest 5 (MD5, [Riv91]) et le Secure Hash Algorithm

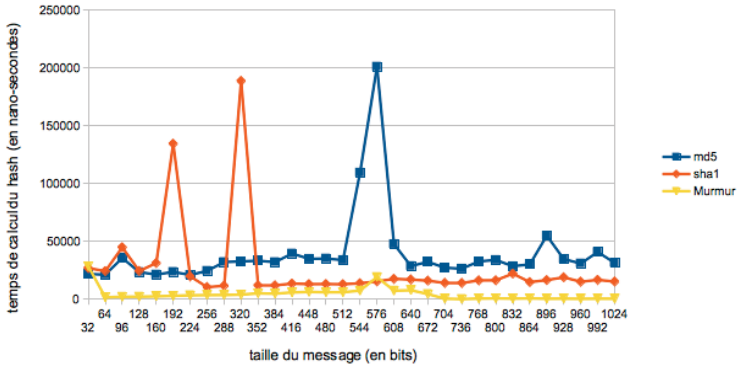


FIGURE 6.4 – Comparaison des fonctions MD5, SHA-1 et MumurHash

(SHA-1, [EJ01]). Ces fonctions sont conçues de manière à fournir un hachage de qualité cryptographique. Les fonctions de hachage cryptographique minimisent le nombre de collisions possibles, c'est à dire qu'elle garantissent l'impossibilité de retourner le même résultat à partir de deux éléments d'entrée différents et que la moindre modification dans la donnée en entrée produit un résultat complètement différent du précédent. De plus, la fonction doit garantir l'irréversibilité du résultat obtenu, c'est à dire qu'il doit être impossible de retrouver, à partir du résultat, la donnée fournie en entrée à la fonction, . Il existe donc aujourd'hui une grande variété de fonctions de hachage qui sont conçues pour s'exécuter plus rapidement, en fournissant parfois moins de sureté d'exécution mais qui, utilisées dans le bon environnement ne créent pas de risque en matière de sécurité. La fonction MurmurHash [App08] est un exemple récent d'une telle fonction de hachage. A titre de comparaison, le graphique 6.4 montre le temps calcul moyen des trois fonctions de hachage citées dans ce paragraphe pour des chaînes de caractères de longueur croissante. Il apparait que la fonction MurmurHash fournit en moyenne un temps de calcul inférieur aux deux autres fonctions, sur l'ensemble des tailles. C'est donc cette fonction de hachage qui est utilisée par l'algorithme WACA, sauf mention contraire. Afin d'obtenir des comparaisons équitables lorsque WACA est comparé à d'autres algorithmes utilisant le hachage, les autres algorithmes sont implantés avec la même fonction. Lorsque, comme dans le cas des filtres de Bloom, plusieurs fonctions de hachage sont nécessaires, il faut éviter les collisions non souhaitées. Une technique fréquemment utilisée pour cela consiste à ajouter un "sel", c'est à dire une courte chaine de caractère permettant de distinguer les chaînes données en en-

trée. Dans le cas spécifique de WACA, chaque fonction de hachage est associé à son propre sel, qui est le numéro de la fonction. Ceci permet d'obtenir, par les propriétés des fonctions de hachage, des résultats suffisamment différents ne créant pas de collisions inattendues.

6.4 Algorithme sans historique (WACA 1)

Le pseudo-code 1 présente une première version du déroulement de l'algorithme. En l'état, cet algorithme a une complexité de $O(n)$ où n est le nombre de noeuds présents dans le système. Cette fonction prend en entrée une liste de machines (nodesList), appelées "nœuds", et une requête (request). Elle s'appuie sur trois fonctions utilitaires :

La fonction *getLeastLoaded* : Cette fonction prend comme paramètres deux machines et renvoie la moins chargée des deux selon un critère défini par l'utilisateur.

La fonction *queryBloomFilter* : Cette fonction prend en entrée une machine et une requête et renvoie VRAI si la requête a déjà été insérée dans le filtre de Bloom associé à cette machine, FAUX sinon. Dans l'implantation de WACA par défaut, la charge est évaluée par le nombre de requêtes en cours d'exécution sur la machine.

La fonction *insertInBloomFilter* : Cette fonction prend elle aussi une machine et une requête comme paramètres. Elle insère la requête dans le filtre de Bloom correspondant à la machine indiquée.

L'algorithme itère sur la liste des nœuds disponibles pour le service demandé *nodesList*. Pour chaque nœud n, l'appel à la fonction *queryBloomFilter* vérifie si la machine n a déjà reçu une requête similaire (ligne 5). A la ligne 9, l'algorithme vérifie si le nœud sélectionné est bien le moins chargé parmi ceux qui ont la requête dans leur filtre.

Si le nœud courant n'a pas d'entrée dans son filtre pour la requête en cours (ligne 11), l'algorithme sélectionne comme cible le nœud avec le nombre de requêtes en cours le moins élevé (lignes 13 et 15).

À la fin de la boucle, l'algorithme retourne le nœud le moins chargé qui a la donnée dans son filtre. Si aucun nœud n'a la donnée dans son filtre, l'algorithme retourne le nœud le moins chargé parmi toutes les machines et ajoute la requête à son filtre (lignes 22,23). Afin d'évaluer l'efficacité de ce nouvel algorithme, nous avons mis en place une infrastructure

de tests minimale à l'ISEP, de manière à pouvoir comparer WACA à d'autres politiques de répartition.

Algorithm 1 WACA version 1

```
1: function wacaLoadBalance(nodesList, request)
2:     bloomNode ← null
3:     llNode ← null
4:     for all n ∈ nodesList do
5:         if queryBloomFilter(n, request) then
6:             if bloomNode = null then
7:                 bloomNode ← n
8:             else
9:                 bloomNode ← getLeastLoaded(bloomNode, n)
10:            end if
11:        else
12:            if llNode = null then
13:                llNode ← n
14:            else
15:                llNode ← getLeastLoaded(llNode, n)
16:            end if
17:        end if
18:    end for
19:    if bloomNode ≠ null then
20:        return bloomNode
21:    else
22:        insertInBloomFilter(llNode, request)
23:        return llNode
24:    end if
25: end function
```

6.4.1 Tests de WACA 1

Application et programme de test : L'application de test utilisée pour évaluer la performance des politiques de répartition dans ces premiers tests est une application web REST [Fie00] de recherche d'images. Les recherches les plus populaires sont placées en cache. Le programme de test lance une série de 10 requêtes consécutives sur l'ensemble des catégories disponibles (une dizaine dans le cas de cette application). Les séries sont lancées par un nouveau thread (fil d'exécution) toutes les 50 millisecondes et se terminent lorsqu'une réponse correcte (sans erreurs) est reçue pour chacune des dix requêtes. Les requêtes de chaque série sont lancées l'une après l'autre. La politique WACA est comparée à une stratégie de répartition en Round Robin, parce que cette politique de répartition sert d'algorithme par défaut dans de nombreux systèmes. WACA est aussi comparé ici à une stratégie de répartition de type Join Shortest Queue sur le modèle de [Bon90]. Cette politique a été choisie car WACA s'appuie sur une politique similaire pour estimer la charge des machines. Ce choix permet donc de bien distinguer l'effet du à la localisation des données permise par l'utilisation des filtres de Bloom. La performance des politiques est mesurée à 6 niveaux de concurrence différents, correspondant au lancement de 10, 30,

50 , 80, 100 et 120 séries de requêtes à 50 milisecondes d'intervalle. La taille des séries ne change pas et reste constante au cours des tests. Au delà de 120 séries concurrentes, l'application de recherche d'images sature rendant impossible une évaluation à plus forte charge. Pour chaque niveau de concurrence et chaque stratégie de répartition de charge, le test est lancé cinq fois et le temps de réponse de chaque requête est enregistré.

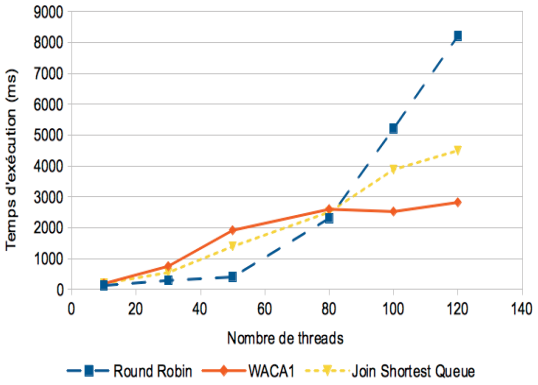
Matériel Les différentes stratégies sont testées avec un noeud *maître*, qui assure la répartition sur la base de la stratégie sélectionnée, et 5 noeuds *esclaves* qui assurent l'exécution des requêtes. Une machine différente est dédiée à l'envoi des requêtes de test, ainsi qu'à l'enregistrement des temps d'exécution. Ces machines sont configurées comme suit :

- 1 noeud Maître (Ubuntu 9) : 1 Intel Pentium D 3Ghz, 1Go de RAM, 30Go d'espace disque,
- 4 noeuds esclave (Ubuntu 9) : 1 Intel Pentium D 3Ghz, 1Go de RAM and 30 Go d'espace disque,
- 1 noeud esclave (Ubuntu 9) : 1 Intel core duo 2.5ghz, 4Go de RAM, 120 Go d'espace disque,
- Noeud de test (Windows 7) : Intel i7 1.73Ghz, 4 Go of RAM, 320 Go d'espace disque.

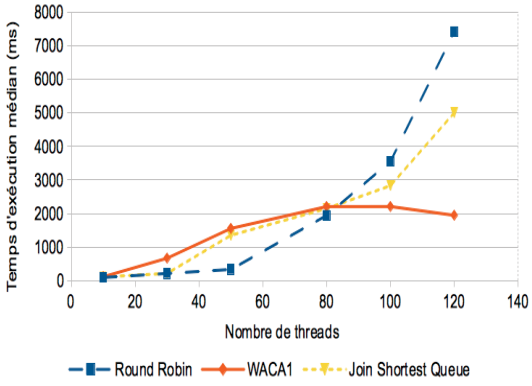
Nous avons choisi d'utiliser une configuration matérielle différente pour au moins un des nœuds esclaves afin de tester le comportement de WACA vis à vis de l'hétérogéneité des machines. Les nœuds sont reliés sur un réseau local ethernet à 100Mb/s.

Paramètrage du répartiteur La politique WACA version 1 a été implantée et testée à l'aide du framework Cloudizer (décrit en chapitre 3). Pour cette application, la taille du filtre avec compteurs a été fixée à 100 cases, et la taille de la file à 25, ce qui permet d'avoir une probabilité de faux positifs égale à 0.155 d'après les formules de la section 6.2. Cette capacité a été choisie car elle correspond à la plus petite capacité des nœuds du système. Les fonctions de hachage choisies lors de ces tests sont les fonctions SHA1 et MD5.

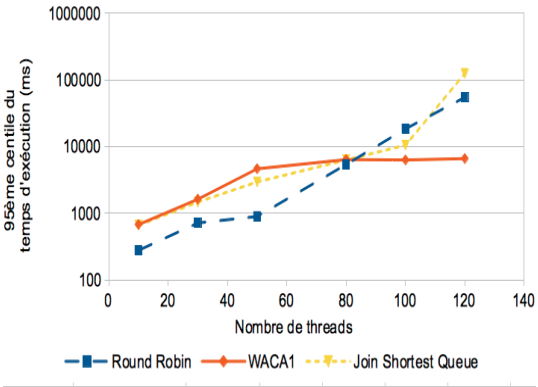
Résultats À partir des données enregistrées lors des tests, nous présentons en figure 6.5 différentes métriques sur les temps d'exécution des requêtes de recherche sur la base



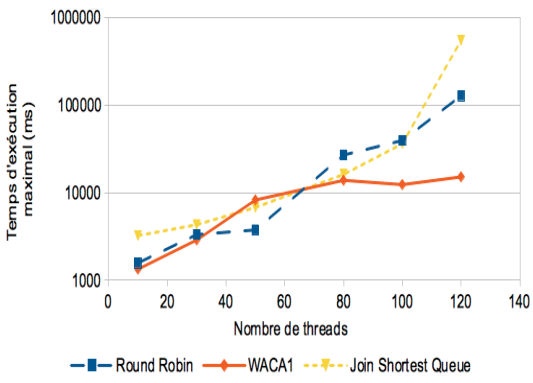
(a) Temps d'exécution moyen des requêtes



(b) Valeur médiane des temps d'exécution



(c) 95ème centile des temps d'exécution



(d) Temps d'exécution maximal

FIGURE 6.5 – Mesures du temps d'exécution des requêtes, WACA version 1

d'images décrite en plus haut. Étant donné que les applications déployées dans le framework CLOUDIZER sont vues comme des "boites noires" il n'est pas possible de mesurer le nombre de requêtes exécutées à partir des données en cache. Il est cependant possible de montrer que, passé un certain niveau de concurrence, et malgré un temps de décision supérieur, la stratégie de répartition WACA fournit de meilleures performances que les deux autres stratégies examinées.

La figure 6.5a montre le temps d'exécution moyen par requête, mesuré en millisecondes. Ce graphique montre qu'à faible charge (10 à 30 threads), la stratégie Round Robin permet d'atteindre un temps de réponse moyen meilleur que les stratégies WACA et JSQ. Cette meilleure performance semble être dûe au faible temps de décision de cette stratégie ($O(1)$).

Cette tendance est confirmée par l'examen des graphiques 6.5b, 6.5c et 6.5d montrant respectivement les valeurs médianes, les 95ème centile et la valeur maximale du temps d'exécution des requêtes sur 5 essais. Les mesures telles que la médiane, le 95ème centile et le temps d'exécution maximal permettent de relativiser la valeur moyenne mesurée en donnant une idée de la dispersion des temps de réponse. Dans les graphiques obtenus on observe des tendances similaires sur ces quatre mesures.

On observe également une inflexion autour de 80 threads où les trois politiques de répartition affichent pratiquement la même performance. Au delà de ce point, la stratégie Round Robin montre une augmentation brutale dans ses temps d'exécution, et la stratégie WACA affiche un temps d'exécution moyen pratiquement divisé par deux par rapport au JSQ et trois par rapport au Round Robin. De plus il a été constaté lors des tests que la stabilité de l'application était mise à mal lors de l'utilisation de la stratégie Round Robin à ces niveaux de concurrence.

Ainsi il est possible d'affirmer qu'en dépit d'un temps de décision avant transmission de la requête plus élevé, la stratégie WACA permet de fournir une meilleure stabilité au système, ainsi qu'un temps d'exécution des requêtes plus prévisible comme on peut le déduire des grahiques de la médiane(figure 6.5b) et du 95ème centile(figure 6.5c). La principale limite de cette version de l'agorithme WACA est donc sa performance lorsque la charge sur le serveur est faible ou moyenne. Il apparait au travers de l'examen des journaux des machines, qu'une seule machine attirait à elle la plupart des requêtes en raison de sa capacité à les résoudre plus rapidement grâce à une puissance de calcul accrue. Nous avons donc mis au point un mécanisme permettant de limiter cet effet : les compteurs d'historique.

6.5 Algorithme avec historique

L'objectif de cette nouvelle version de l'algorithme WACA est de produire une meilleure répartition des données utilisées dans les caches des différentes machines. L'ajout, pour chaque machine d'un compteur d'historique, incrémenté à chaque fois qu'une requête auparavant absente est ajoutée au filtre, permet d'évaluer le nombre de requêtes différentes auxquelles un serveur peut répondre. En comparant les compteurs des différentes machines,

il est possible de choisir d'affecter une nouvelle requête à la machine qui a reçu le moins de requêtes différentes. Lorsque la diversité des requêtes augmente, ce mécanisme permet d'éviter qu'un nœud ait un plus grand nombre de requêtes dans son cache que les autres machines du système. Cela résulte en une charge de travail mieux répartie sur l'ensemble des machines disponibles.

Algorithm 2 WACA version 2

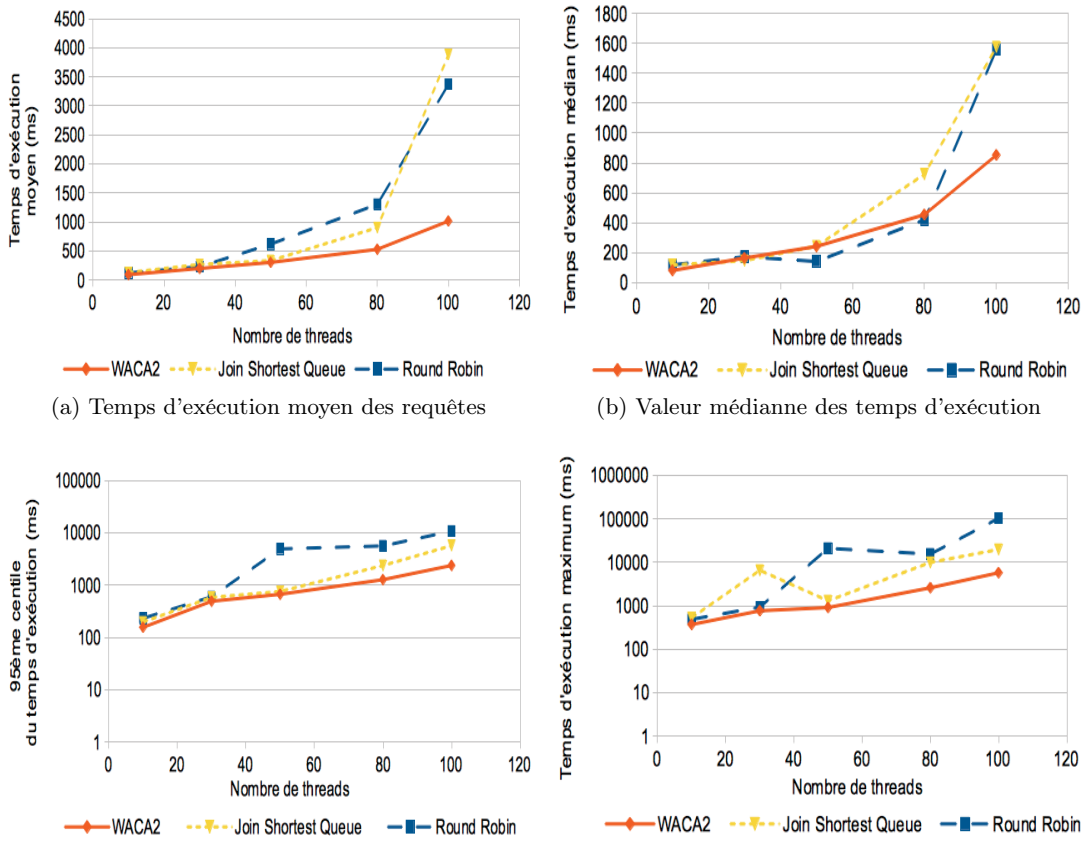
```
 1: function WACA2LOADBALANCE(nodesList, request)
 2:     bloomNode ← null
 3:     llNode ← null
 4:     target ← null
 5:     llTmp ← null
 6:     for all n ∈ nodesList do
 7:         if queryBloomFilter(n, request) then
 8:             if bloomNode = null then
 9:                 bloomNode ← n
10:             else
11:                 bloomNode ← getLeastLoaded(bloomNode, n)
12:             end if
13:         else
14:             if llTmp = null then
15:                 llTmp ← n
16:             else if llTmp.history() > n.history() then
17:                 llTmp ← n
18:             else
19:                 llTmp = getLeastLoaded(llTmp, n)
20:             end if
21:         end if
22:     end for
23:     target ← llTmp
24:     if bloomNode ≠ null then
25:         target ← bloomNode
26:     end if
27:     insertInBloomFilter(target, request)
28:     return target
29: end function
```

Cette modification a entrainé l'écriture de l'algorithme WACA version 2 (cf. algorithme 2). Le changement principal se situe aux lignes 16 et 17, où le noeud le moins chargé est sélectionné par rapport à son historique plutôt que par rapport à son compteur de requêtes.

6.5.1 Compléxité de l'algorithme et gain apporté

Les deux versions de l'algorithme présentées ici s'exécutent en temps proportionnel au nombre de machines disponibles $O(n)$ et en espace $O(nkb)$. Cette complexité est à relativiser étant donné la rapidité des opérations effectuées pour chaque nœud (consultation / insertion dans le filtre de Bloom). De plus, ce ralentissement est compensé par le gain obtenu en temps d'exécution par l'utilisation du cache. L'efficacité de l'algorithme dépend donc de la différence entre le temps d'exécution d'une requête avec les données en cache et son temps d'exécution avec les données en dehors du cache. Si ce gain n'est pas signi-

ficativement important, alors le recours à une politique de répartition type "cache-aware" n'est peut être pas approprié pour l'application. Les temps de réponses typiques d'un service web doivent se situer autour de 500 millisecondes au niveau des serveurs pour rester acceptables ([Nie93]). Cette nouvelle version de l'algorithme de répartition a été évaluée avec le même programme de test que WACA version 1.



(a) Temps d'exécution moyen des requêtes

(b) Valeur médianne des temps d'exécution

(c) 95ème centile des temps d'exécution

(d) Temps d'exécution maximal

FIGURE 6.6 – Mesures du temps d'exécution des requêtes, WACA version 2

6.5.2 Experimentation de WACA avec historique

Les courbes présentées en figure 6.6 montrent que les temps d'exécution des requêtes en utilisant la stratégie WACA2 sont plus proches des temps d'exécution de la stratégie JSQ. De plus, on constate que l'écart de performance en faveur de la stratégie WACA

se maintient pour un nombre élevé de requêtes concurrentes. Le choix d'implanter un historique de requêtes a donc permis de réduire l'écart de performance lorsque la charge est faible ou moyenne. Là encore la tendance reste visible à travers les 4 métriques choisies. Cette amélioration de la performance s'est faite au détriment de la stabilité du système car au delà de 100 threads, il n'était plus possible de réaliser l'expérience en raison d'un grand nombre d'erreurs sur une des machines esclaves.

Une amélioration générale des performances est visible sur les graphiques présentés en figure 6.6. Ceci est dû à l'utilisation d'une nouvelle version de la plateforme CLOUDIZER, dont le répartiteur a été réécrit en JAVA, en lieu et place du langage GROOVY utilisé dans l'expérimentation précédente. Malgré la différence entre les deux dispositifs expérimentaux,

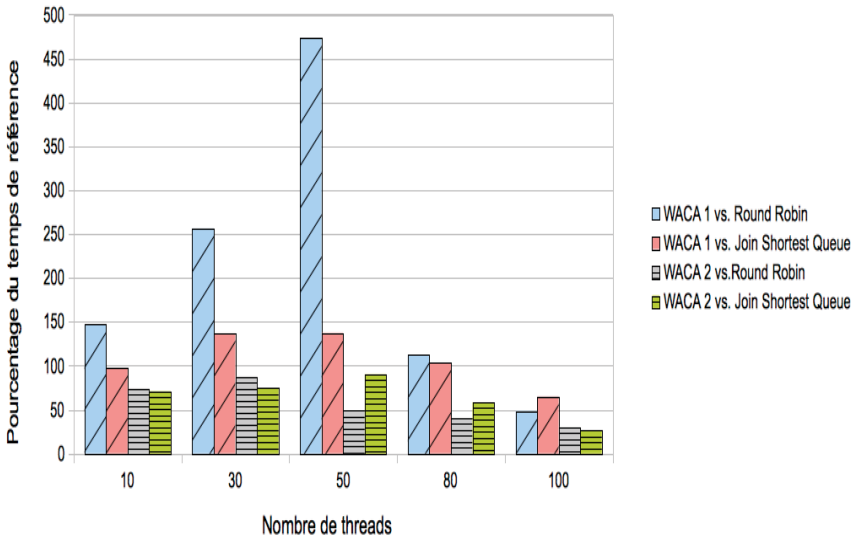


FIGURE 6.7 – Comparaisons de WACA 1 et 2

il est possible de comparer les deux stratégies en étudiant les gains relatifs obtenus dans chaque expérience par rapport au politiques Round Robin et JSQ. La figure 6.7 montre une évaluation du gain en temps d'exécution obtenu pour chaque version de WACA par rapport aux algorithmes Round Robin et Join the Shortest Queue. Un score inférieur à 100 exprime un temps d'exécution moyen inférieur à celui de la politique de référence notée dans la légende. Nous constatons donc que WACA avec historique permet d'obtenir des temps d'exécutions inférieurs à ceux obtenus avec les politiques Round Robin et JSQ quel

que soit le niveau de concurrence de l'application, ce qui n'est pas le cas avec WACA 1.

Les politiques de répartition fondées sur les résumés de cache comme la stratégie WACA décrite ici sont cependant limitées lorsque le nombre de machines augmente, leur temps d'exécution dépendant du nombre de nœuds présents dans le système. Or, le contexte d'application de cette stratégie nécessite un grand nombre de machines. Une nouvelle version de l'algorithme a donc été mise au point avec à l'esprit la réduction de la complexité de l'algorithme au minimum.

6.6 Block Based WACA (BB-WACA)

L'algorithme Block Based Waca (BB-WACA) tire avantage de la division en blocs des Filtres de Bloom en Blocs (FBB) pour indexer le contenu des filtres. Cette indexation permet de réduire le temps de sélection d'un nœud de manière significative, tout en conservant le taux de faux positifs souhaité par l'utilisateur. Dans l'algorithme WACA, la sélection d'une machine se fait en fonction de deux critères : la présence de la requête dans le filtre de la machine, et la charge de cette machine. Le premier critère nécessite de requêter tous les filtres de Bloom des machines candidates et le second nécessite la comparaison de la charge de toutes les machines entre elles.

Cet algorithme utilise un tableau d'index, dont le fonctionnement est décrit en section suivante, pour filtrer les machines pouvant potentiellement répondre à une requête donnée, et réduire ainsi le nombre de filtres à examiner. La figure 6.8 montre le flot d'exécution de cette nouvelle version de l'algorithme.

6.6.1 Description de l'algorithme

De même que précédemment, un filtre de Bloom en Bloc (FBB) est créé pour chaque machine du système. Chaque filtre est dimensionné d'après les formules décrites en section 6.4. Parmi les filtres créés lors de l'initialisation, deux paramètres du plus grand des filtres sont conservés : la taille de ses blocs B_{max} et le nombre de fonctions de hachages utilisées par ce filtre, appelé K_{max}. En plus de ces filtres, l'algorithme BB-WACA crée une structure appelée l'*index T*. L'index est un tableau de listes chainées dont la taille est égale à B_{max}.

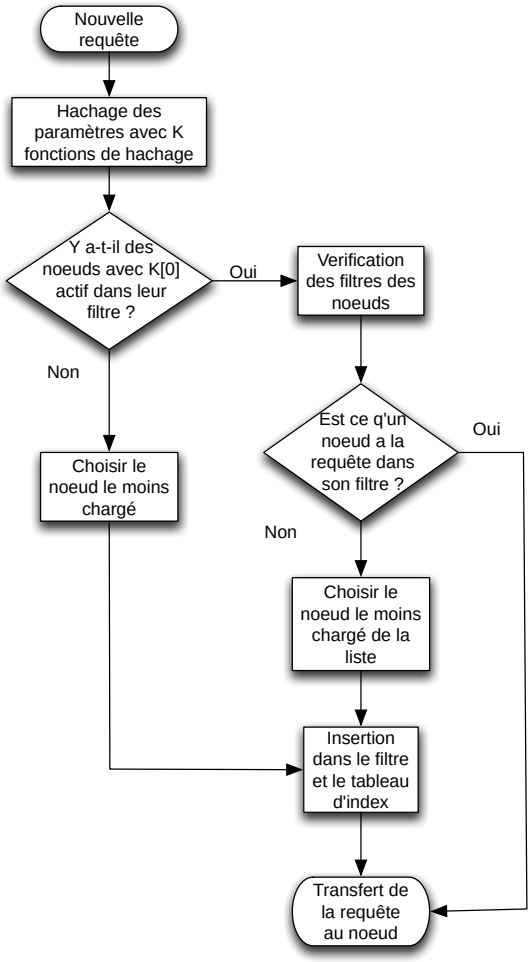


FIGURE 6.8 – Flot d'exécution de l'algorithme BB WACA

Lorsqu'une requête est reçue, elle est hachée K_{max} fois. Le premier hash h_0 indique une cellule dans l'index T. Si la cellule de l'index n'est pas vide, l'algorithme itère sur la liste des machines de la cellule pour tester l'appartenance de la requête à un des filtres disponibles. Si la requête n'est trouvée dans aucun des filtres, alors c'est la machine la moins chargée du système qui est sélectionnée pour exécuter la requête et l'insérer dans son filtre. Lors de cette insertion, le premier des K_{max} hash, appelé h_0, est utilisé pour sélectionner une position dans l'*index T*, et la machine cible est ajoutée à la liste contenue dans cette case du tableau (si elle n' y est pas déjà présente). La structure finale de l'*index T* peut être représentée par le schéma 6.9.

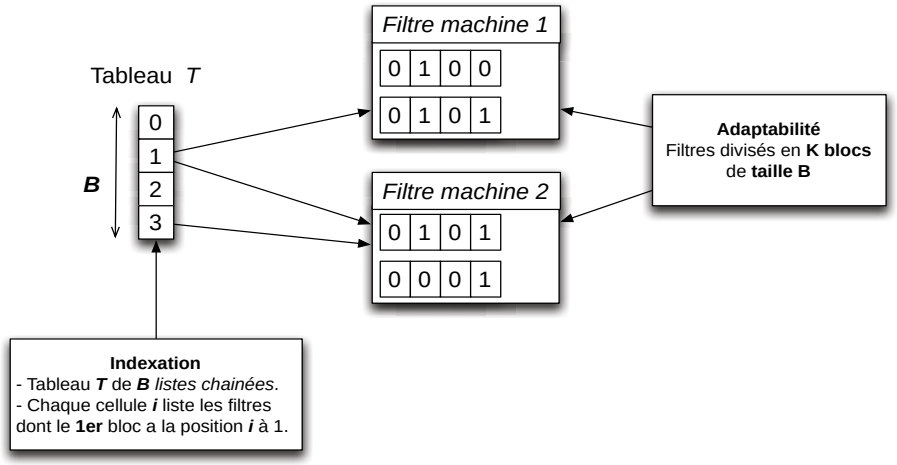


FIGURE 6.9 – Schéma du tableau d'index

L'ajout de ce tableau permet lors de la réception d'une requête de rechercher une machine candidat en priorité dans la liste contenue dans la cellule du tableau, ce qui réduit le nombre d'itérations à effectuer avant de trouver une machine ayant effectivement la requête dans son cache.

Dans le cas où une requête n'est trouvée dans aucun des filtres, l'algorithme sélectionne la machine la moins chargée de toutes. Le système maintient donc une liste ordonnée par le nombre de requêtes en cours sur les machines, dont le premier élément est le nœud le moins chargé. Cette liste est implantée par un tas de Fibonacci ([FT87]) qui est un tas binomial particulier, dont la racine est toujours l'élément dont la clé est la plus petite des clés de l'arbre, de sorte que cette structure est fréquemment utilisée pour l'implantation de listes de priorités. Cette structure a aussi l'avantage de fournir une complexité faible pour les opérations telles que la suppression (temps constant), trouver le minimum (temps constant), et l'insertion (temps logarithmique), qui sont les trois principales opérations utilisées par notre algorithme. Chaque FBB est associé à une machine et peut donc être associé au niveau de charge de cette dernière. Le niveau de charge de chaque machine est utilisé comme clé dans le tas de Fibonacci, et la valeur associée à cette clé est le FBB du nœud, de sorte que les FBB sont classés par ordre de charge dans le tas. La position centrale du répartiteur est mise à profit pour mettre à jour le tas à chaque fois qu'une réponse est envoyée à un client : la clé de la machine correspondante est décrémentée. Le

désavantage de cette approche est qu'elle nécessite de maintenir un pointeur sur chaque nœud du tas de Fibonacci.

Chaque machine est associée à un compteur de requêtes, et il est possible de connaître le nombre de processeurs disponibles sur chaque machine. Lorsque le nombre de requêtes en cours dépasse le nombre de cœurs disponibles sur une machine, elle est considérée comme surchargée par l'algorithme, et ne pourra recevoir une nouvelle requête que si elle est la moins chargée de toutes les machines du système.

L'algorithme final est récapitulé dans le listing 3. De même que dans les versions précédentes, cet algorithme est appelé à chaque arrivée d'une requête dans le système. Les paramètres de la requête sont hachés K_{max} fois. Le premier hachage h_0 est utilisé pour déterminer la présence d'une liste dans la cellule correspondante de l'index T (lignes 5 et 6). Si la cellule h_0 contient une liste de machines, alors l'algorithme itère sur les membres de cette liste (ligne 8). Pour chaque filtre dans la liste, l'algorithme détermine si la requête courante y a déjà été insérée et si le noeud associé à ce filtre n'est pas surchargé (lignes 9 et 10). Si le filtre contient la requête et que la charge du noeud est acceptable, alors le noeud est retourné par l'algorithme. Si aucune des machines de la liste ne contient la requête, elle est alors transférée à la machine la moins chargée de la liste. Si toutes les machines de la liste sont surchargées, ou qu'aucune liste n'existe pour la requête courante, on sélectionne alors la machine la moins chargée de tout le système via le tas de Fibonacci (ligne 19).

6.6.2 Analyse de l'algorithme

Empreinte mémoire L'algorithme utilise un FBB par machine pour représenter les résumés. La taille des filtres est déterminée par les équations décrites dans la section 6.2. Chaque filtre dépend donc de la capacité mémoire de chaque machine en terme de nombre d'objets et du taux de faux positifs souhaité par l'utilisateur. Dès lors, l'empreinte mémoire de l'algorithme est donnée par la somme de la taille des filtres et de l'index T. Cette dernière dépend, comme noté dans la section précédente, de la taille du plus grand bloc, parmi tous les filtres. Pour illustrer cela, la figure 6.10a montre la progression de l'utilisation mémoire de l'algorithme par rapport à l'augmentation du nombre de machines et avec un taux de faux positifs souhaité de $0,1\%$, et pour stocker 1000, 10.000 puis 100.000 objets. L'axe

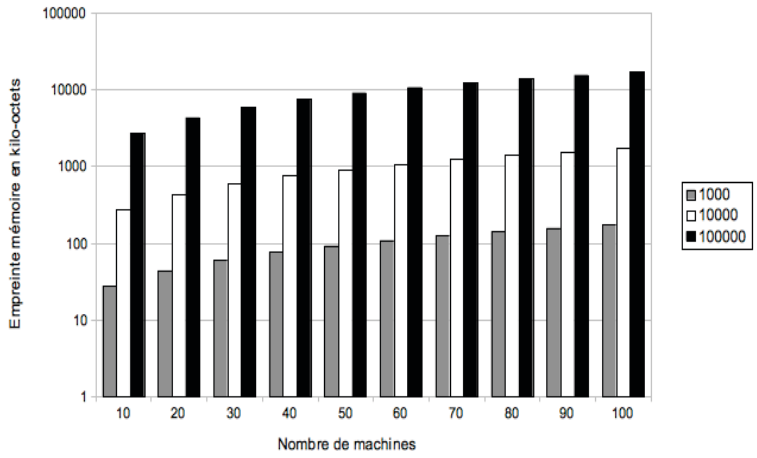
Algorithm 3 Block Based WACA

```
1: function bbWacaLoadBalance(nodesList, request)
2:     target ← Nil
3:     bf ← Nil
4:     H ← hash(request)
5:     i ← H[0]
6:     if T[i] not empty then
7:         bf ← T[i].head
8:         while bfnot Nil do
9:             load ← getNodeLoad(bf)
10:            if bf.lookup(H) AND load = true then
11:                return bf.node
12:            else if load = true then
13:                target ← getLeastLoaded(target, bf.node)
14:            end if
15:            bf ← bf.next
16:        end while
17:    end if
18:    if target = Nil then
19:        target ← getLeastLoaded(nodesList)
20:        bf ← target.getFilter()
21:    end if
22:    if not bf.lookup(H) then
23:        bf.insert(H)
24:    end if
25:    return target
26: end function
```

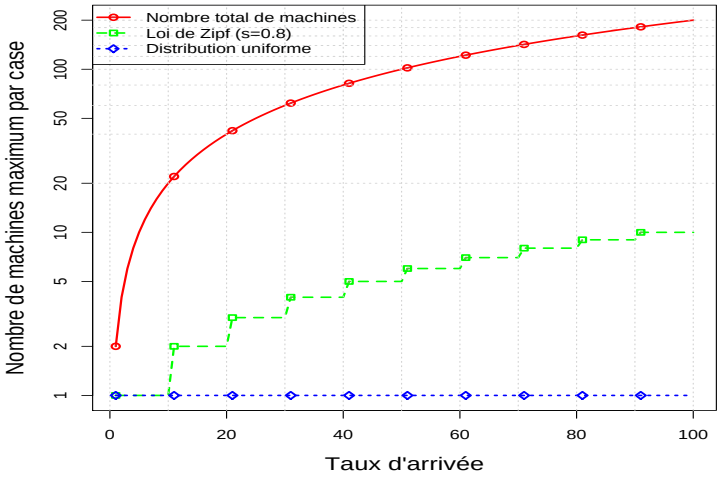
des ordonnées est en échelle logarithmique pour améliorer la lisibilité de la figure. Dans le pire des cas, il apparait que pour 100 machines stockant chacune 100.000 objets, la mémoire utilisée par l'algorithme est équivalente à 17Mo. Cette taille bien qu'importante reste supportable pour la plupart des machines modernes. En pratique, les listes de filtres de l'index T sont instanciées à la volée, ce qui permet de limiter la consommation totale du système, étant donné que le taux de remplissage de l'index T est limité à 50% d'après les formules de la section 6.2.

Limites au nombre d'itérations Le principe de l'algorithme BB-WACA décrit en section 6.6 précise que le nombre d'itérations effectuées lors de la sélection d'une machine dépend au final du nombre de filtres de Bloom présents dans la cellule de l'index T qui est examinée. Or, un filtre est ajouté à la liste d'une cellule s'il n'y est pas déjà et qu'un des éléments du premier bloc du filtre correspond au numéro de la cellule. Ces conditions d'ajout peuvent se traduire en une probabilité d'ajout d'un filtre à une cellule. De cette probabilité, il est possible de déduire le nombre maximum potentiel d'ajouts dans une cellule de l'index et donc le nombre maximum et moyen d'itérations de l'algorithme.

Soit n_b le nombre maximum de serveurs dans une cellule b de l'index T. Soit λ le nombre moyen de requêtes reçues par le système dans l'intervalle de temps t. Il est généralement



(a) Taille des filtres vs. nombre de machines



(b) Estimation du nombre maximum de filtres par cellules de l'index

FIGURE 6.10 – Complexité

admis que les arrivées de requêtes dans un système informatique suivent une loi de Poisson avec comme paramètre l'espérance λ. L'intervalle de temps séparant les arrivées est appelé Δ [Fei02]. En supposant que la fonction de hachage choisie pour traiter les requêtes dans l'algorithme BB-WACA soit efficace et ne produit pas trop de collisions, il est possible de déduire que les affectations de requêtes à chaque bloc (les "arrivées") dans chaque cellule b de l'index T, suivent elles aussi une loi de Poisson de paramètres λ_b et t, où b est une cellule

de l'index T. En fonction de la distribution des paramètres des requêtes, le taux λ_b peut être différent pour chaque cellule. Par exemple, si cette distribution n'est pas uniforme, alors un certain type de paramètres sera plus fréquent que les autres, ce qui provoquera un plus grand nombre de requêtes pour la cellule b. Pour répondre à cette fréquence plus importante, l'algorithme BB WACA est conçu pour ajouter une nouvelle machine dans une cellule si aucune des machines déjà présente n'est en mesure de répondre à la nouvelle requête. Cette action fait donc augmenter le nombre de machines dans la cellule.

Par cette propriété de l'algorithme, il est possible de déduire que si le système est correctement dimensionné, c'est à dire que le nombre de machines est suffisant pour répondre au taux d'arrivées λ des requêtes, alors le nombre de filtres à examiner c dans la cellule b de l'index T peut être évalué par la formule d'équilibre des files d'attentes 6.8, où μ est le taux de service et ρ est le taux d'utilisation des machines :

$$c = \frac{\lambda_b/\mu}{\rho} \tag{6.8}$$

Ici, λ_b correspond au taux d'arrivée de requêtes dans le bloc b. On suppose que le temps de de service μ est fixe. La figure 6.10b montre l'évolution du nombre maximum de machines par bloc d'après la formule 6.8, pour un intervalle $t = 30$ et un temps de service moyen $\mu = 60$. L'axe des ordonnées est en échelle logarithmique afin d'améliorer la lisibilité des courbes. L'axe des abscisses exprime le taux taux d'arrivée λ. La courbe pleine représente le nombre total de machines dans le système nécessaire pour répondre au débit λ en abscisse. Les trois courbes en pointillés expriment l'évolution du nombre maximum de filtres par cellule de l'index T en fonction du biais de la distribution suivie par les paramètres des requêtes. Chaque courbe représente le nombre maximum possible d'éléments dans une cellule quelconque du tableau d'index, en fonction de la distribution des requêtes. Il apparait que l'algorithme est assez sensible au biais, comme en témoigne l'évolution de la courbe en correspondant à la distribution Zipfienne. Cependant ce nombre reste éloigné du nombre total de machines dans le système (représenté par la courbe continue). La solution la plus simple pour limiter cette croissance résiderait dans la limitation du nombre de machines pouvant répondre à une requête donnée par l'utilisation d'un nombre fixé de répliques des données utilisées. Cependant, cette technique risque à terme de nuire au temps de réponse

en produisant plus d'attente sur les machines contenant la réplique la plus demandée.

6.7 Evaluation de la politique BB WACA

Nous avons pu tester la politique BB-WACA avec différents paramètres. Trois points étaient à évaluer : premièrement comment se comporte la politique BB-WACA par rapport à d'autres politiques de répartition ? Deuxièmement, quelle est l'influence de la distribution des requetes sur la performance de la politique BB-WACA ? Enfin, comment se comporte l'algorithme lorsque le nombre de machines à interroger augmente ?

6.7.1 Évaluation par simulation

L'évaluation de BB-WACA et des versions précédentes des politiques de répartition s'est faite avec l'outil Simizer développé au cours de ces travaux de thèse et décrit au chapitre 4. Lors de ces tests les politiques de répartition ont été évaluées par rapport à trois facteurs principaux :

- hétérogénéité des machines / passage à l'échelle : L'algorithme exploite-t-il au mieux les ressources disponibles fournies par les machines du système ? Est-ce que l'ajout de nouvelles ressources dégrade les performances de l'algorithme ?
- hétérogénéité des requêtes : Différentes requêtes de différents utilisateurs ne requièrent pas toujours la même quantité de ressources pour être exécutées. Est-ce que la diversité dans les besoins en ressources des différentes requêtes affecte le comportement de la stratégie ?
- distribution des paramètres : Il a été montré dans divers travaux d'études sur les charges de travail que les appels typiques à un service web peuvent suivre différentes lois de répartitions (Gaussienne, Zipfienne, Uniforme). Il est donc nécessaire de tester le comportement de la politique face à ces différentes lois [Fei02, GJTP12].

Stratégies utilisées pour la comparaison Les trois politiques WACA ont été comparées au stratégies suivantes :

- Round Robin : Il s'agit d'un algorithme statique de répartition circulaire, en "tourniquet",

TABLE 6.1 – Configurations des tests de la stratégie WACA

Type de distribution	Paramètre	Nombre de requêtes distinctes
Gaussienne	200.0	1000
Zipf	0.8	1000
Uniforme	1/1000	1000

– Hachage Cohérent ou Consistent Hashing : Décrit par [KR04] et largement utilisé dans les tables de hachage distribuées, ainsi que dans divers systèmes de stockages tels que Cassandra [LM09] ou Memcached [Fit04].

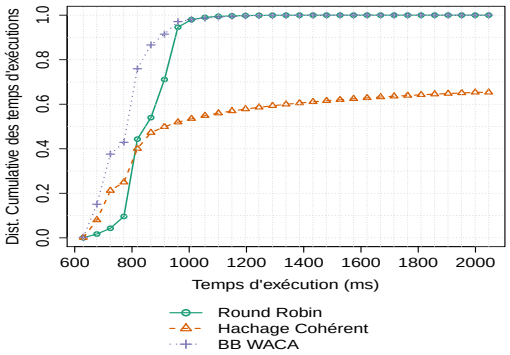
Le tableau 6.1 liste les différentes configurations d'envoi de requêtes testées lors des simulations. Pour chaque configuration, une trentaine de simulations sont effectuées avec chaque politique de répartition. Le travail simulé consiste en un ensemble de 1000 requêtes accédant des ressources différentes dont la la taille correspond à 8 mégaoctets. Chaque requête nécessite l'exécution de 2250 Millions d'instructions (MI). Chaque distribution a été testée sur 20, 50 puis 100 machines. Le nombre total de requêtes envoyées varie à chaque simulation, mais s'établit moyenne de 67 requêtes par seconde. À chaque test, les machines sont divisées en deux catégories : la moitié des machines possèdent 2 cœurs et 512 mégaoctets de mémoire, et l'autre moitié possède 4 cœurs et 1 gigaoctet de mémoire. Les cœurs de chaque machine sont configurés pour exécuter 4000 millions d'instructions par secondes (MIPS).

Résultats Les graphiques en figures 6.11, 6.12 et 6.13, montrent les temps d'exécution simulés respectivement pour une distrbution des requêtes Uniforme, Gaussienne et Zipfienne. La politique BB-WACA (courbes pointillée) est comparée à une politique Round Robin (courbe pleine) et une politique de Hachage Cohérent (tirets).

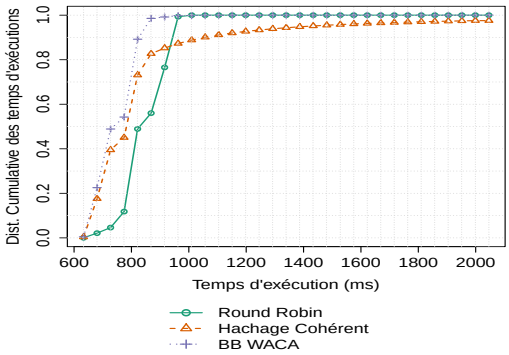
Les résultats obtenus montrent un comportement efficace de la politique BB-WACA face aux différentes distributions de paramètres utilisée. La distribution cumulative des temps de réponse montre bien qu'il est possible de garantir un meilleur temps de réponse avec la politique BB-WACA qu'avec les politiques de répartition en Round Robin et Hachage Cohérent. La différence de performance entre les politiques de répartition étudiées et l'algorithme BB-WACA s'explique aussi par l'utilisation d'un compteur de requête dans

l'algorithme BB-WACA, qui évite l'envoi de requêtes à des machines surchargées.

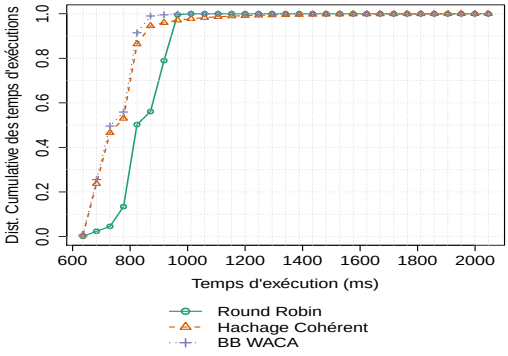
Il est important de noter que lorsque la quantité de ressources dans le système est suffisante, comme ici lors des tests avec 100 machines (graphiques 6.11c et 6.12c), la différence de performance entre BB-WACA et la politique de Hachage Cohérent est peu significative, en dépit du fait que la politique WACA repose sur une structure de données pouvant générer des faux positifs. L'utilisation de filtres de Bloom pour évaluer la localisation des données ne nuit donc pas à la performance de la répartition. Cependant, dans le cas où la distribution des requêtes est biaisée (distribution Zipfienne, figure 6.13c), la capacité d'adaptation des filtres de Bloom indexés utilisés par la stratégie BB-WACA permet une amélioration significative par rapport à la stratégie du Hachage Cohérent.



(a) 20 machines

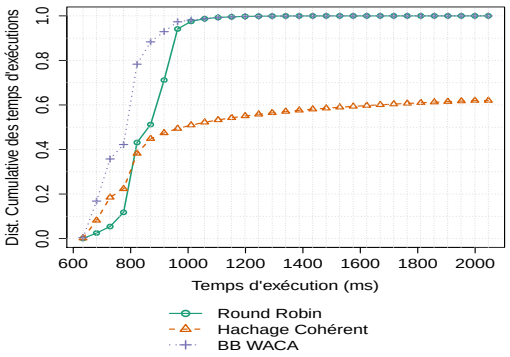


(b) 50 machines

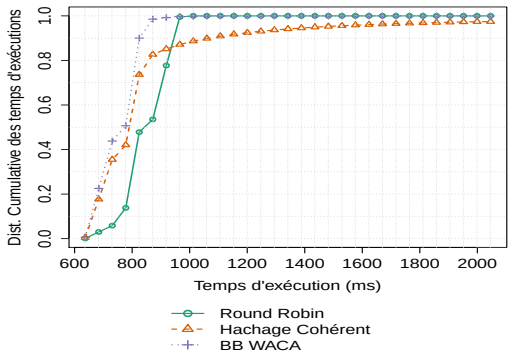


(c) 100 machines

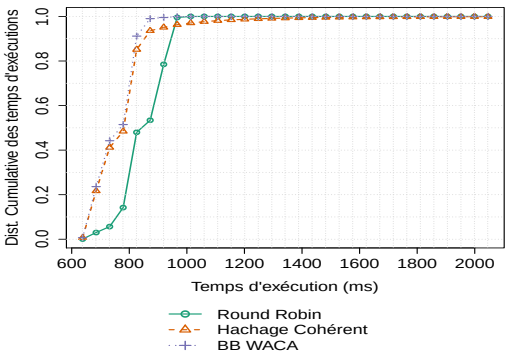
FIGURE 6.11 – Distribution des temps de réponses simulés, 1000 requêtes, distribution Uniforme



(a) 20 machines



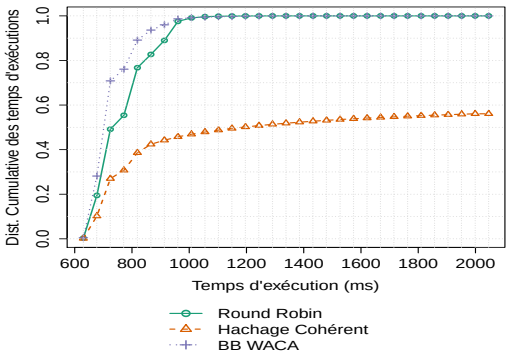
(b) 50 machines



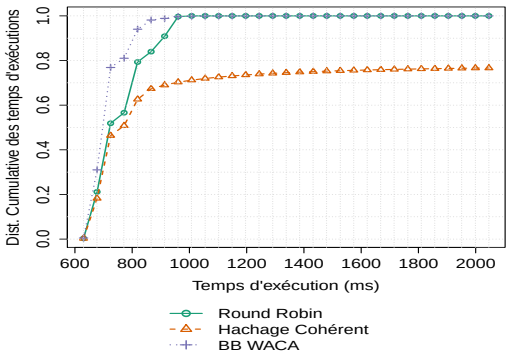
(c) 100 machines

FIGURE 6.12 – Distribution des temps de réponses simulés, 1000 requêtes, distribution Gaussienne

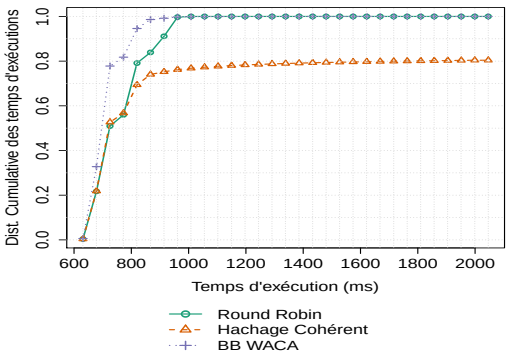
Le diagramme en barres 6.14 confirme cette affirmation. Ce graphique montre la répartition du nombre de requêtes reçues par chaque machine du système simulé lors d'un test sur 20 machines, dans lequel les machines 0 à 9 sont équipées de 2 cœurs et 512 mégaoctets de mémoire, et les machines 10 à 19 sont équipées de quatre cœurs et 1 gigaoctet de mémoire. Ce graphique montre bien que l'algorithme Round Robin (barres de gauche) traite toutes les machines à égalité en leur envoyant à chacune le même nombre de requêtes de traitement. En revanche, ce n'est pas le cas pour la stratégie de Hachage Cohérent (barres du milieu) pour lequel les requêtes se répartissent aléatoirement sur l'ensemble des machines. La stratégie BB-WACA (barres de droites) affecte en revanche plus de requêtes aux machines 11 à 20, plus puissantes, qu'aux machines 1 à 10. Cette répartition est effectuée



(a) 20 machines



(b) 50 machines



(c) 100 machines

FIGURE 6.13 – Distribution des temps de réponses simulés, 1000 requêtes, distribution Zipfienne

naturellement grâce au mécanisme de répartition de charge par comptage des requêtes mis en place dans la stratégie WACA, et permet de prendre en compte les capacités hétérogènes des différentes machines d'un système.

L'évaluation de BB-WACA passe aussi par l'évaluation du temps de sélection d'une machine, qui permet d'évaluer la capacité de traitement de l'algorithme. La comparaison effectuée dans les graphiques de la figure 6.15, montre les différences de temps de sélection de machines entre les trois versions de la stratégie WACA. Ces résultats sont présentés pour une loi de distribution des requêtes Gaussienne (figure 6.15a) et Zipfienne (figure 6.15b). On y observe une nette amélioration du temps de sélection par rapport aux deux version précédentes de l'algorithme, due à l'utilisation des filtres de Bloom indexés. Les

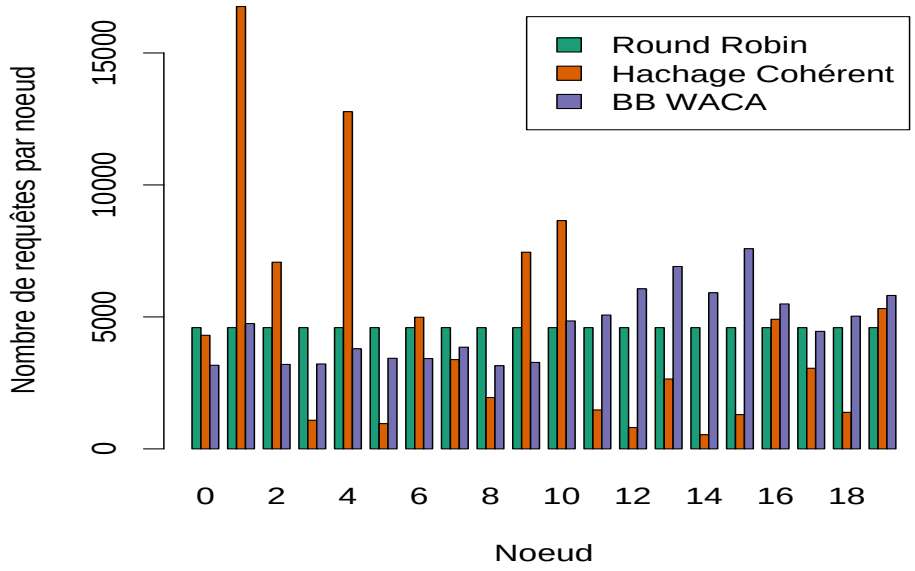


FIGURE 6.14 – Répartition des requêtes sur 20 machines, (Zipf)

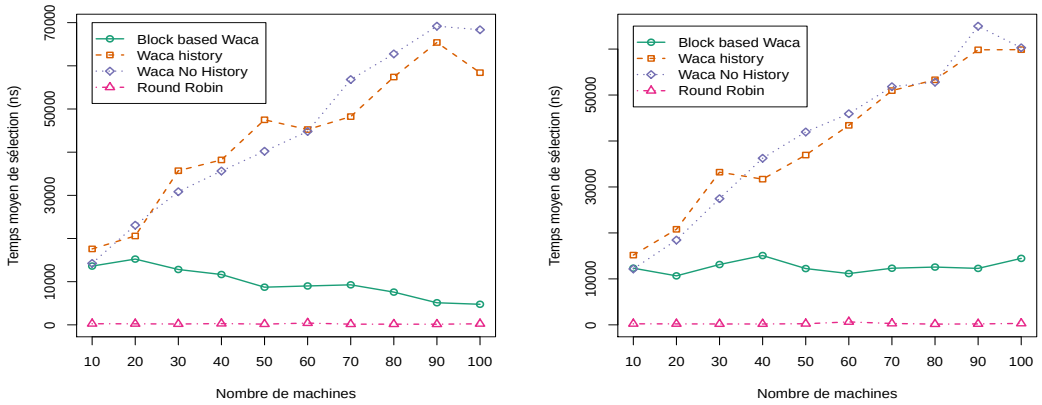
résultats obtenus à travers les différentes simulations sont encourageant et permettent donc de supposer que le comportement de l'algorithme WACA s'avérera performant dans un environnement de déploiement réel.

6.7.2 Évaluation pratique sur l'Elastic Compute Cloud

Système de test Le service de traitement de données météorologiques décrit en section 6.1.1 a été utilisé pour évaluer en conditions réelles la performance de la politique de répartition WACA. Ces tests ont été effectués sur un service d'infrastructure à la demande : le service Elastic Compute Cloud d'Amazon [1].

Ce service permet à ses utilisateurs de louer des machines virtuelles moyennant un prix horaire. Ce prix dépend de la puissance de la machine virtuelle sélectionnée, qui varie selon plusieurs critères comme le nombre de cœurs et leurs vitesses, la quantité de mémoire vive ou encore bande passante disque ou réseau. En plus de ce prix horaire, plusieurs autres

1. `aws.amazon.com`, consulté le 5 octobre 2013



(a) Temps moyen de sélection : loi Gaussienne

(b) Temps moyen de sélection : loi de Zipf

FIGURE 6.15 – Résultats de simulation : temps de sélection des machines

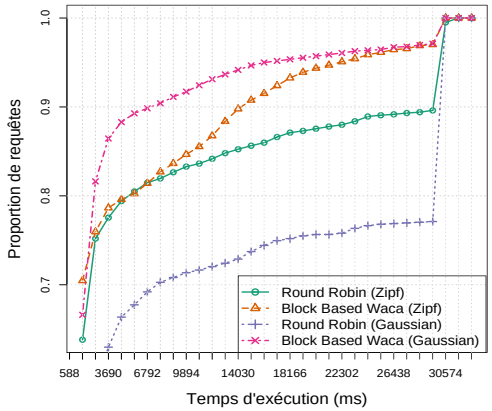
postes de facturation existent, comme la quantité d'entrées sorties disques, la quantité de données transmises et reçues sur le réseau. Ce service a été choisi pour sa flexibilité et la rapidité avec laquelle il est possible de déployer un nombre important de machines. Cependant, le nombre total de machines utilisables par un seul utilisateur est de 20, un déploiement plus important doit faire l'objet d'une demande motivée auprès du fournisseur. L'application est répartie sur les machines à l'aide du service Cloudizer décrit au chapitre 3 de ce document.

La politique BB-WACA a été évaluée sur plusieurs combinaisons de machines différentes. Le système de test se compose de 7 à 20 machines. Deux machines de type Large sont utilisées comme client et comme Répartiteur. Le client utilise un processus génératif pour lancer les requêtes, ce qui lui permet de s'adapter dynamiquement à la capacité du système. Le reste des machines constitue les nœuds du système, exécutant le service d'interpolation des données. Les requêtes sont un échantillon de 1000 requêtes différentes accédant l'ensemble des données stockées sur les machines. Lorsque les données ne sont pas en mémoire, le temps de réponse d'un serveur de type t1.micro varie entre 12 et 16 secondes. Lorsque les données sont en cache, le temps de réponse oscille entre 550 et 750 millisecondes. Ces intervalles sont aussi influencés par le nombre de requêtes en cours au même instant sur chaque machine. Les arrivées des utilisateurs suivent une loi de Poisson

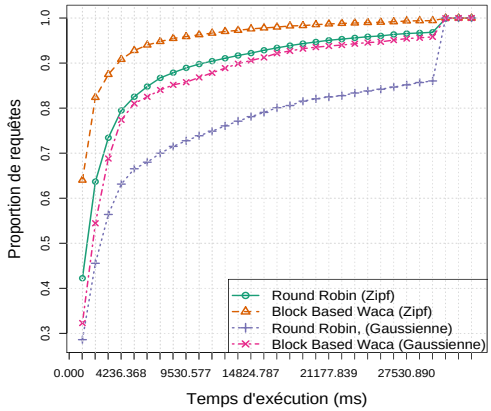
de taux d'arrivée $\lambda = 5$ et d'intervalle $t = 1$ seconde. Chaque utilisateur est simulé par un processus léger émettant consécutivement des requêtes choisies aléatoirement parmi les 1000 requêtes types. Ce choix peut se faire via une loi de Zipf ou une loi Gaussienne, utilisant les mêmes générateurs que ceux utilisés dans le simulteur Simizer (cf. chapitre 4. Chaque utilisateur émet un nombre variable de requêtes, déterminé par son temps total d'activité sur le site et un temps d'attente entre chaque requête. Ce temps d'attente suit une loi exponentielle. L'activité d'un utilisateur cesse une fois qu'un certains temps total est écoulé, lui aussi déterminé par une loi exponentielle. Cette simulation du comportement des utilisateurs est fondée sur l'ouvrage de Feitelson[Fei02] disponible en ligne [2] concernant la modélisation des charges de travail dans les systèmes distribués. Des tests ont été menés avec 5, 10, 15 et 18 machines. Le test mené avec 5 machines est destiné à montrer la capacité de la stratégie WACA à utiliser les ressources disponibles sur des machines hétérogènes, car ce système de test était constitué de deux machines de type "small" et trois machines de type "large".

Résultats Les résultats des tests décrits ci-dessus sont présentés dans les graphiques en figure 6.16. Sur chaque graphique, quatre courbes sont visibles. Les courbes ayant des figures géométriques comme symbole (rond et triangle) correspondent aux tests menés avec une loi de Zipf. Les courbes présentant des croix correspondent aux tests menés avec une loi Gaussienne. Ces courbes présentent la distribution cumulative des temps de réponses observés sur trois à cinq tests. Les résultats montrent que la stratégie BB-WACA permet d'avoir régulièrement une proportion plus importante de requêtes rapides comparée à la politique Round Robin. L'écart devient plus significatif avec l'augmentation du nombre de machines, ou avec l'utilisation de machines de différentes capacités (graphique 6.16a). Il est à noter que plus le nombre de machines est important, plus la proportion de données en cache est importante, car il y a plus de mémoire disponible. Une fois que la taille cumulée des caches des machines atteint la taille des données à traiter, les performances commencent à atteindre un plateau et ne varient plus beaucoup. Une observation intéressante des résultats montre que l'utilisation du Round Robin lorsque les requêtes sont distribuées selon une loi de Zipf permet d'obtenir de meilleurs temps de réponse que lorsque les requêtes suivent

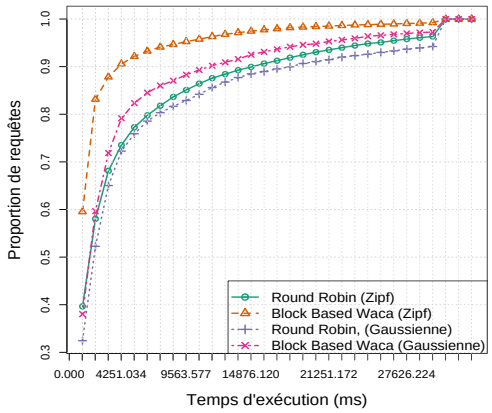
2. `http://www.cs.huji.ac.il/~feit/wlmod/`, consulté le 5 octobre 2013



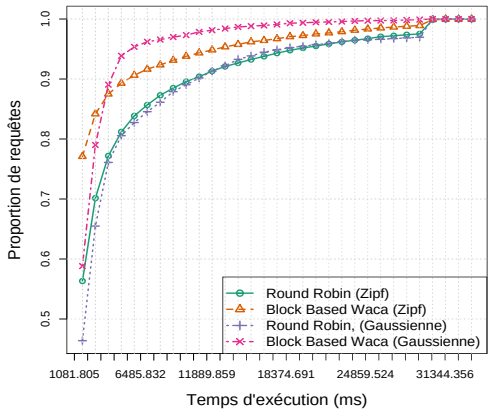
(a) Distribution des temps d'exécution, 5 machines



(b) Distribution des temps d'exécution, 10 machines



(c) Distribution des temps d'exécution, 15 machines



(d) Distribution des temps d'exécution, 18 machines

FIGURE 6.16 – Résultats d'exécutions sur le cloud d'Amazon

une loi Gaussienne. En effet, selon une loi de Zipf, la partie des requêtes demandées le plus fréquemment est plus présente dans les caches des machines. Par conséquent, le Round Robin, qui répartit les requêtes de manière plus juste comparée à l'algorithme BB- WACA, est plus performant que lorsque la distribution des requêtes est Zipfienne. Cependant, dans les deux cas, la stratégie BB-WACA permet d'obtenir un meilleur niveau de service, en fournissant une fraction de requêtes plus importante dans des temps de réponses acceptables

(inférieurs à 1 seconde).

6.8 Conclusion

Les résultats de l'évaluation de la performance des stratégies WACA dans le Cloud permettent de confirmer les résultats obtenus par simulation et ont permis de fournir les moyens d'évaluer de manière précise la performance d'une application dans le cloud. Les stratégies WACA ont été conçues de manière à fournir un algorithme de répartition prenant en compte à la fois la charge de travail des machines et la localisation des données. Les résultats de simulation, ainsi que les évaluations en pratique sur le Cloud d'Amazon montrent que ces objectifs sont atteints dans la dernière version WACA. Il ressort de la comparaison entre les simulations effectuées avec le logiciel Simizer et les tests réels, que les résultats de simulations ne permettent que d'indiquer une tendance du comportement des politiques, qui se traduit par des écarts moins significatifs dans les résultats observés en pratique. Ces variations d'écarts peuvent être attribuées aux variations de performance observable dans les environnements Cloud et à la difficulté de modéliser fidèlement le comportement de l'application étudiée.

Les éléments distinctifs de l'algorithme WACA sont, par rapport à des algorithmes utilisant des structures de données semblables, sa vitesse d'exécution et sa généricité, c'est à dire la possibilité d'être déployé sur de nombreuses infrastructures allant du traitement parallèle des données aux services web (montrés dans ce chapitre). L'algorithme WACA s'adapte à différentes distributions de requêtes en permettant une adaptation de la charge à la capacité mémoire des machines. Il est possible d'ajouter d'autres optimisations à cet algorithme dont les différentes versions ont fait l'objet de plusieurs publications ([SL12, LKC13, Lef13]). Les futures évolutions de cet algorithme consisteront à concevoir un nouveau protocole d'échange des filtres de Bloom entre les nœuds, afin de permettre une exécution distribuée de cet algorithme.

Chapitre 7

CAWA : Un algorithme de répartition basé sur les les coûts

7.1 Introduction

Nous avons montré dans le chapitre précédent que les algorithmes fondés sur la charge et la localité fournissent de bonnes performances dans les environnements de type infrastructure à la demande. Ces gains sont obtenus sur deux fronts :

– La latence est réduite, ce qui implique une réduction des temps de réponse,

– Le nombre d'accès disque est réduit, ce qui libère des ressources sur les machines.

Le fait d'exécuter les requêtes plus rapidement et en consommant moins de ressources améliore le rapport coût/ efficacité des machines. Cet avantage est particulièrement intéressant dans les plateformes de Cloud Computing de type "Infrastructure à la Demande" où les machines peuvent être louées à l'heure et les opérations sur les disques sont facturées.

Une politique de répartition de charge adéquate permettrait donc de réduire le coût d'exécution des tâches dans le système. Les nouvelles infrastructures de type Cloud utilisent des tarifs biens définis et disponibles en ligne. Par exemple, le service d'infrastructure à la demande Elastic Compute Cloud (EC2), utilisé pour exécuter les tests du chapitre précédent, a une politique de prix qui dépend de la puissance des machines : de quelques centimes de l'heure pour une instance de type "Micro" équipée de 700 mégaoctets de RAM et d'un seul processeur, jusqu'à 1 dollar l'heure pour une machine virtuelle équipée de

processeurs graphiques [1].

Par conséquent, il est possible de se poser la question de l'utilisation de cette information de prix pour en déduire l'efficacité d'un système en terme de coût. Le coût (au sens financier) d'exécution d'une requête ne représente-t-il pas un indicateur de performance compact dénotant un traitement rapide et efficient des requêtes ? Si cet indicateur est pertinent, mettre au point une stratégie visant la diminution des coûts ou leur optimisation permettra aussi d'obtenir une plus grande efficacité du système. Pour cela, il est nécessaire de calculer le coût d'exécution des différentes requêtes en fonction de la machine sur laquelle elles s'exécutent. Cette information obtenue, il reste à déterminer une affectation optimale des tâches aux différentes machines, permettant de minimiser le coût total d'exécution de l'ensemble des tâches.

Une approche similaire est utilisée par Rogers et al. [RPC10] : les auteurs de ces travaux analysent a priori un échantillon de requêtes et déterminent la composition optimale du système à utiliser en fonction des caractéristiques de l'échantillon observé. Cependant, la stratégie développée par les auteurs détermine a priori les différentes ressources à utiliser. La robustesse aux changements dans les caractéristiques de la charge n'est donc pas garantie par cette approche. En effet, la distribution des requêtes, ou même la performance et le nombre des machines disponibles peuvent varier au cours d'une même journée, ce qui nécessite d'utiliser un algorithme pouvant s'adapter à ces changements. D'autant plus que les infrastructures de type Cloud permettent l'ajout ou la suppression de ressources à la demande. Les autres approches fondées sur le coût[AaC09, MH11] reposent sur les notions de budget et de date limite d'exécution dans le temps pour optimiser le coût d'exécution des travaux.

Ce chapitre étudie une nouvelle approche pour la mise au point d'un algorithme utilisant l'information de coût comme indicateur de performance. Après la description générale de l'algorithme en section 7.2, les différentes phases de cette stratégie sont décrites dans les sections 7.3 et 7.4. La section 7.5 montre les résultats obtenus par simulation de l'approche, et la section 7.6 pose les bases d'un algorithme mêlant les approches WACA et CAWA. La section 7.7 résume les conclusions de ce chapitre et les limites de cette approche du

1. `http://aws.amazon.com/pricing/ec2/`, consulté le 5 octobre 2013

problème de répartition.

7.2 Approche proposée : Cost AWare Algorithm

L'objectif d'un algorithme fondé sur le coût est d'affecter les requêtes aux différentes ressources du système, de sorte que leur exécution soit la plus efficace et rapide possible en fonction de l'historique du système, tout en minimisant le coût total d'exécution. Ce problème peut être modélisé par un problème d'affectation des tâches, qui peut être résolu en temps polynomial par l'algorithme dit "Hongrois" [Kuh55, Mun57]. L'exécution d'un tel algorithme peut être lente, particulièrement lorsque le nombre de tâches à répartir est trop élevé. Afin de réduire cette complexité, un algorithme de classification est utilisé pour regrouper les tâches en catégories homogènes. L'optimisation est ensuite exécutée sur ces groupes de tâches plutôt que sur l'ensemble des différentes tâches identifiées. Cette approche est nommée "Cost-AWare Algorithm" (CAWA). Cet algorithme se décompose en plusieurs étapes :

L'initialisation : Durant cette phase, le système répartit les tâches selon une stratégie définie par l'utilisateur, et enregistre les temps d'exécution de chaque tâche.

La classification/ optimisation Décrite schématiquement en figure 7.1, cette phase est déclenchée régulièrement, après un certain intervalle de temps ou un nombre de requêtes fixé par l'utilisateur. Elle est exécutée en tâche de fond. C'est durant cette phase que les tâches préalablement enregistrées sont regroupées en classes (classification) et que l'algorithme Hongrois est utilisé (optimisation).

La répartition : Cette phase commence à la fin de l'exécution de la phase de classification. Durant cette phase, chaque tâche est affectée à la machine qui est désignée comme la plus efficace selon le résultat obtenu lors de l'optimisation. Les différentes requêtes sont à nouveau enregistrées comme lors de l'étape d'initialisation, pour pouvoir être utilisées lors de la prochaine classification.

Le cœur de l'algorithme est la phase de classification optimisation, dont le problème d'optimisation est modélisé dans la section suivante.

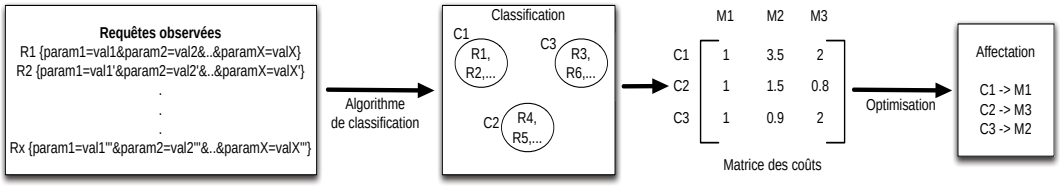


FIGURE 7.1 – Description de la phase de classification / optimisation

7.2.1 Modélisation du problème

L'algorithme CAWA cherche à optimiser les coûts d'exécution dans un environnement Cloud en analysant les requêtes passées. Au fur et à mesure que le répartiteur reçoit des requêtes, le système enregistre les temps d'exécution des requêtes. Ces requêtes sont groupées par un algorithme de classification tel que l'algorithme des k-moyennes [M+67] en M classes correspondant au nombre de machines du système. Cette classification est réalisée en considérant les requêtes comme les individus à classifier et les arguments des requêtes (voire des caractéristiques d'exécution des tâches du passé telles que le taux d'utilisation processeur) comme les paramètres régissant la classification. Cette classification peut être réactualisée après un certain temps ou après un nombre prédéfini de requêtes. Le but est d'obtenir une matrice $M \times M$ nommée C, répertoriant pour chaque classe de tâche i son coût d'exécution $C_{i,j}$, avec j le numéro de machine. Cette matrice est appelée matrice des coûts.

$$C = \begin{Bmatrix} C_{0,0} & C_{0,1} & \cdots & C_{0,j} \\ C_{1,0} & C_{1,1} & \cdots & C_{1,j} \\ \vdots & \vdots & \ddots & \vdots \\ C_{i,0} & C_{i,2} & \cdots & C_{i,j} \end{Bmatrix} \tag{7.1}$$

Une fois cette matrice générée, on l'utilise pour optimiser le programme suivant avec C_j le coût horaire de la machine j :

$$\begin{array}{l} \min \sum_{i=1}^{M} \sum_{j=1}^{M} x_{ij} \frac{C_j}{T_{ij}} \\ \text{st.} \\ x_{i,j} = \{0,1\} \\ \sum_{i=1}^{M} x_i = 1 \\ \sum_{j=1}^{M} x_j = 1 \end{array} \tag{7.2}$$

L'objectif de ce système est de minimiser le coût financier d'exécution des requêtes. Le vecteur des coûts C pour les M machines est fourni par l'utilisateur. La variable de décision

x désigne pour chaque classe de requêtes i la machine j pour laquelle le coût d'exécution est le moins élevé. La matrice X obtenue indiquera vers quelle machine diriger les requêtes en fonction de leur classe. Les contraintes sur la matrice X permettent de s'assurer qu'une machine ne s'occupe que des requêtes d'une classe. Cette solution peut être obtenue à l'aide de l'algorithme dit "Hongrois" [Kuh55, Mun57] qui est un algorithme d'optimisation conçu pour résoudre ce type de problème à partir de la matrice des coûts. Cet algorithme permet de trouver une affectation optimale des tâches permettant de minimiser le coût total d'exécution, et ceci en temps polynomial $O(n^3)$, où n est la dimension de la matrice C, ce qui en fait l'algorithme générique le plus efficace pour résoudre ce genre de problème.

7.2.2 Avantages de l'approche

Il est possible d'arguer que la réduction de la complexité par l'utilisation d'un algorithme de classification en amont ne soit pas efficace, en raison de la complexité de l'algorithme de classification lui même. Cependant, il est nécessaire de noter que la classification permet d'obtenir des prototypes de tâches. Ces prototypes permettront de prévoir à quelle machine affecter une tâche qui n'a pas été observée précédemment, ce qui est impossible à faire si l'affectation est faite tâche par tâche.

7.3 Phase de classification

L'analyse par classification des charges de travail observées sur un système permet de séparer les différentes catégories d'utilisation d'un système donné [MHCD10]. Cette approche peut être utilisée pour optimiser les coûts d'exécution comme dans les travaux de Rogers et al. [RPC10], car elle permet d'associer les catégories identifiées avec un coût d'exécution. Le contexte de l'algorithme CAWA est celui des services Web, par conséquent, les "tâches" à répartir sont des requêtes HTTP adressées à un certain service. L'objectif de la phase de classification est d'obtenir, à partir d'un ensemble de requêtes de service observées, une partition de l'ensemble des requêtes sur les machines disponibles dans le système. Pour parvenir à une classification pertinente, il est nécessaire de choisir l'espace de représentation des requêtes puis de sélectionner la méthode de classification la plus appropriée à cette représentation.

7.3.1 Représentation et pré-traitement des requêtes

Structure d'une requête web Les données étudiées sont des requêtes de pages web envoyées par des clients à un serveur. La durée de service est donc liée aux paramètres utilisés dans la requête. La RFC numéro 3986 ([EJ01]) définit la notion d' Identifiant de Ressource Uniforme (URI) comme suit :

"Un identifiant de ressource uniforme (URI, Uniform Ressource Identifier) fournit un moyen simple et extensible d'identification d'une ressource." page 4, section 1.

La notion de *ressource* englobe ici tous les éléments pouvant être identifiés par une URI : il peut s'agir d'un fichier image, d'une vidéo, d'une page web, mais aussi de services web. Par exemple, l'URI *"https ://www.google.com/search ?q=RFC3986"* désigne les résultats d'une recherche Google sur le texte "RFC3986". La plupart des services web étant aujourd'hui dynamiques, cette requête déclenche probablement un processus de recherche de mots-clés dans les bases de données du moteur de recherche. Le résultat de cette recherche est représenté par l'URI, mais la requête a eu comme effet de bord de déclencher ce processus de recherche. Les URI identifient donc le résultat obtenu, mais l'envoi de la requête au serveur déclenche un service produisant le résultat souhaité. Une URI est composée de plusieurs parties distinctes [EJ01], dans l'ordre d'apparition :

Le schéma : Le schéma désigne la partie comprise entre le début de la chaîne et le caractère " :". Cet élément détermine comment interpréter la suite de l'URI (page 11, section 3.1).

L'autorité : Le composant Autorité permet de désigner l'hôte de la ressource désignée (page 12, section 3.2).

Le chemin : Le chemin est un ensemble hiérarchisé précisant la ressource accédée sur l'hôte.

La requête : Cette partie précise la ressource accédée en donnant des paramètres au service invoqué (page 15, section 3.3).

Le dernière partie, la *requête* peut se décomposer en plusieurs *paramètres* chacun accompagné de *valeurs*. Les valeurs des paramètres fournis au service déterminent le plus souvent le

flot final d'exécution du code, ce qui conditionne *in fine* le temps d'exécution des requêtes. Par conséquent ces paramètres sont des variables pertinentes pour classifier les requêtes reçues en différentes catégories. De plus, ces paramètres conditionnent le temps d'exécution des requêtes, ce qui permet de déterminer le coût de la requête.

Représentation des requêtes Pour appliquer un algorithme de classification à ces données, il est nécessaire de normaliser les données reçues et d'associer la performance observée avec les paramètres. Le service web étudié étant considéré comme une "boite noire", aucune information ne peut être obtenue a priori sur la distribution des requêtes. Chaque requête est donc représentée par un vecteur numérique. Les éléments tels que les dates, qui peuvent être reconnus et convertis dans ce type de représentation, le sont. Les chaînes de caractères alphanumériques sont converties en valeurs entières via une fonction de hachage. Le temps d'exécution de la requête ainsi que le numéro de machine sur lequel elle s'est exécutée sont ajoutés au vecteur. L'ensemble des paramètres peut être considéré comme des variables explicatives du temps d'exécution. Le temps d'exécution des requêtes est normalisé par rapport au nombre de requêtes observées sur chaque machine et à l'intervalle de temps écoulé depuis le début de l'observation. Soit $t_{r,m}$ le temps d'exécution de la requête r sur la machine m, et T_m la somme des temps d'exécution des requêtes observées sur la machine m pendant la durée δ_t, tel que :

$$T_m = \sum t_{r,m} \tag{7.3}$$

Il est alors possible de calculer le temps effectivement passé à exécuter la requête r proportionnellement à δ_t avec la formule suivante :

$$p_r = \frac{t_{r,m}}{T_m} \times \delta_t \tag{7.4}$$

Il est à noter cependant que le temps passé à traiter une requête peut varier en fonction du nombre de requêtes actuellement en cours d'exécution sur la machine. Plus le nombre de requêtes augmente, moins la proportion de temps consacré à chaque requête est importante. L'estimation du coût de chaque requête est mécaniquement affectée par ce problème. Cette donnée du temps d'exécution est considérée comme une méta-donnée, c'est à dire une information additionnelle, qui peut influencer la classification des requêtes. Cette information peut être utilisée de deux manières. Premièrement, lors de la classification en

l'ajoutant au vecteur décrivant chaque requête. Autrement cette information est utilisée pour calculer le coût d'exécution des requêtes sur les machines, car ce coût est relatif au temps d'exécution.

7.3.2 Algorithmes de classification

Les algorithmes de classification sélectionnés doivent permettre de choisir le nombre de classes résultantes de sorte que l'on puisse le définir comme égal au nombre de machines présentes dans le système. Parmi les techniques possibles nous pouvons citer :

L'algorithme des k-moyennes L'algorithme des k-moyennes [M+67] est un algorithme de partitionnement de données qui permet de répartir un ensemble d'individus donné en un nombre de classes déterminé à l'avance, si les individus sont représentés dans un espace Euclidien. Chaque classe possède un *centre* qui peut être déterminé aléatoirement lors de l'initialisation de l'algorithme. A chaque itération, l'algorithme affecte chaque individu à la classe dont le centre est le plus proche. La détermination de la proximité entre classes se fait à l'aide d'une formule de calcul de distance. Ce peut être la distance Euclidienne, la distance de Manhattan, ou toute autre formule permettant d'estimer la similarité entre deux individus. Ensuite, un nouveau centre est déterminé pour chaque classe en faisant la moyenne des individus qui y sont affectés. L'itération est répétée à nouveau et l'algorithme s'arrête lorsqu'il n'y a plus de changement dans les affectations des individus ou lorsqu'un nombre prédéterminé d'itérations est atteint.

Le choix du nombre de classes à atteindre ainsi que la détermination des centres initiaux ont une grande influence sur l'affectation finale. En outre, l'espace de représentation et de normalisation des paramètres des individus doit être choisi avec soin de manière à refléter correctement les associations au sein des classes. Les choix de représentation des requêtes pour l'algorithme CAWA sont discutés en section 7.3.1.

La classification hiérarchique ascendante La classification hiérarchique ascendante utilise la notion de dissimilarité pour produire une classification en rassemblant les individus les plus proches en différentes classes. Ce rassemblement s'effectue en fusionnant les classes

les plus proches les unes des autres. La détermination de la dissimilarité entre classes peut se faire de plusieurs manières :

- *Saut minimum* : Calcule la distance séparant les individus les plus proches entre les deux classes.
- *Saut maximum* : Calcule la distance séparant les individus les plus éloignés entre deux classes.
- *Lien moyen* : Calcule la distance moyenne entre les individus des classes comparées.
- *Distance de Ward* : Calcule la distance entre les centres de gravité des classes comparées.

Au départ de l'algorithme, chaque individu ou observation est affecté à une classe et il existe autant de classes que d'individus. A chaque itération, les deux classes les plus proches sont fusionnées, et le score de dissimilarité est recalculé entre la nouvelle classe et les autres classes existantes. Cette opération est répétée jusqu'à ce que le nombre de classes souhaité par l'utilisateur soit atteint. L'avantage de cette méthode est qu'elle demande moins de calculs que l'algorithme des k-moyennes, et ne dépend pas d'une sélection aléatoire des centres, ce qui permet d'obtenir des classes parfois plus cohérentes qu'avec les k-moyennes.

De plus, la classification hiérarchique repose sur la distance entre les individus. Cette approche simplifie la représentation des individus, car il n'est pas nécessaire de normaliser les individus dans un repère absolu. Ceci facilite la mise en relation des différentes requêtes, tant que le nombre de paramètres reste restreint.

7.4 Phase d'optimisation

7.4.1 Matrice des coûts

La matrice des coûts associe le coût moyen d'exécution des éléments appartenant à une classe à chaque machine du système. Dans un premier temps, les coûts sont normalisés par machine : Soit $c_{r,m}$ le coût de la requête r exécutée sur la machine m. Ce coût dépend du temps machine affecté à la résolution de cette requête. Étant donné que les machines peuvent répondre à plusieurs requêtes en parallèle dans un laps de temps donné, le coût d'une seule requête doit être estimé en proportion de l'intervalle de temps observé, du coût

de la machine et du nombre de requêtes actuellement en cours d'exécution sur la machine. Le coût unitaire de la requête se calcule donc par rapport à la durée de l'intervalle de temps observé obtenue par l'équation 7.4 :

$$c_{r,m} = p_r \times \delta_t \times C_m \tag{7.5}$$

Toutes les requêtes d'une classe donnée ne peuvent cependant pas avoir été exécutées sur toutes les machines du système durant la période d'observation. Ainsi, lorsque le coût d'exécution d'une requête est inconnu sur une machine, la moyenne des coûts observés pour toutes les requêtes reçues sur la machine est utilisée. Cette approche permet d'obtenir une estimation raisonnable du coût d'exécution potentiel d'une requête. La formation de cette matrice est nécessaire pour pouvoir identifier les ensembles de tâches réalisés le plus efficacement par certaines machines. La performance de chaque machine est donc évaluée dans l'algorithme CAWA par le coût moyen d'exécution des requêtes. Cependant, comme le notent Rogers et al. dans [RPC10], l'estimation du coût d'une même requête consommant les mêmes ressources peut varier en fonction du nombre total de requêtes. Mécaniquement, plus une machine exécute de requêtes dans un intervalle de temps t, moins chaque requête coûte cher. Or, plus le nombre de requêtes exécutées sur une machine est important, plus la performance se dégrade, à cause des accès multiples aux mêmes ressources matérielles. Ces deux effets antagonistes rendent difficile l'estimation précise des coûts.

7.4.2 Résolution par la méthode Hongroise

Le problème d'affectation des tâches obtenu à la suite de la classification décrite en section précédente est facilement solvable par l'algorithme Hongrois décrit par [Kuh55] et analysé dans [Mun57]. Cette méthode permet de résoudre un problème d'affectation de tâches en utilisant une représentation matricielle des coûts et des travailleurs. L'algorithme hongrois est composé d'une boucle qui répète les étapes suivantes, jusqu'à ce que l'optimum soit atteint :

1. Soustraction des lignes : La valeur minimale de chaque ligne est soustraite à toutes les valeurs de la ligne. Après cette étape, chaque ligne de la matrice contient au minimum un zéro.

2. Soustraction des colonnes : La valeur minimale de chaque colonne est soustraite à toutes les valeurs de la colonne.

3. Sélection des zéros indépendants : Cette étape consiste à découvrir une sous matrice de la matrice des coûts qui contient des tâches interdépendantes. La valeur minimale de la sous matrice est ajoutée aux valeurs des lignes ne faisant pas partie de la sous-matrice, et retirée des colonnes de la sous-matrice.

A chaque étape, l'algorithme vérifie qu'il est possible d'affecter toutes les tâches à toutes les machines sans créer de conflit. Lorsque ce n'est pas possible, l'étape suivante est exécutée, sinon l'algorithme est arrêté et retourne l'affectation trouvée.

7.4.3 Répartition vers les machines

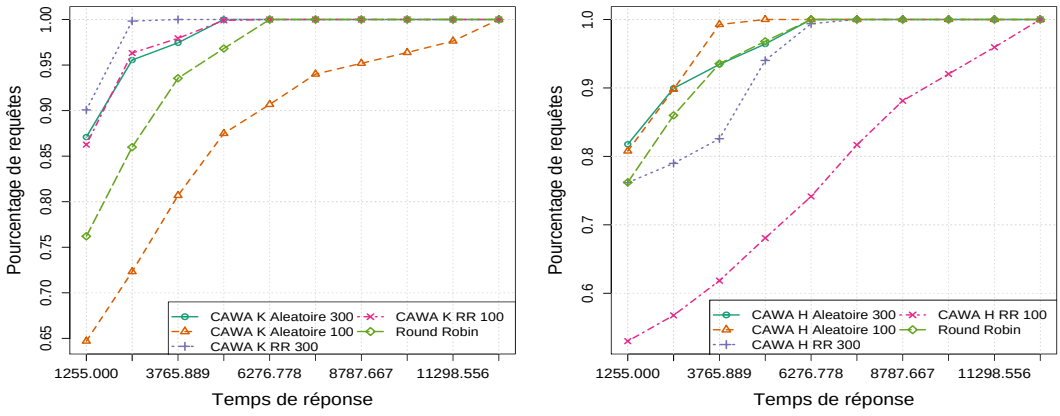
À l'issue de la phase d'optimisation, un modèle d'affectation est retourné à l'algorithme de répartition. Ce modèle est constitué d'une liste d'associations entre un centre d'une classe et une machine du système. Chaque nouvelle requête est comparée aux centres de classes obtenus à l'étape de classification. Lorsque le centre le plus proche est trouvé, la requête est transférée à la machine associée à ce centre. Il s'agit donc de la machine pour laquelle l'exécution de cette requête semble la plus efficace. L'algorithme utilisé pour trouver le centre le plus proche est actuellement un algorithme de recherche naïf calculant la requête la plus proche sur l'ensemble des centres.

7.5 Évaluation par simulation

L'implantation de l'algorithme CAWA a été faite dans le simulateur Simizer décrit en chapitre 4. CAWA est comparé à l'algorithme Round Robin. Nous rappelons ici que le but de l'algorithme CAWA est d'utiliser le coût comme indicateur de performance. Par conséquent, il ne s'agit pas d'une optimisation des coûts à proprement parler mais d'une optimisation des ressources dont le coût est connu à l'avance.

7.5.1 Preuve de concept : exemple avec deux machines

Le modèle de système utilisé est simulé avec le logiciel Simizer. Il consiste en un ensemble de machines clientes, un répartiteur central et un ensemble de serveurs d'applications. Le but de ces tests est d'établir si l'estimation du coût comme indicateur de performance ainsi que l'affectation des requêtes par optimisation utilisée dans l'algorithme CAWA, permettent d'obtenir des résultats performants. La charge de travail se compose d'un ensemble de requêtes divisé en deux catégories principales : les requêtes "simples" pour lesquelles la quantité d'instructions est égale à 1500 Millions d'instructions (MI) et les requêtes "complexes" dont le nombre d'instructions est significativement plus élevé, soit 2500 MI. Le processus d'envoi des requêtes est similaire à celui utilisé lors des tests menés pour l'algorithme WACA au chapitre 6. Les requêtes se répartissent en un ensemble de 300 paramètres différents. L'initialisation de l'algorithme CAWA nécessite de choisir un type particulier de répartition initiale. Deux types de répartition ont été testés à ce niveau : une répartition initiale en Round Robin et une répartition initiale aléatoire, pour chaque type de partitionnement utilisé. Ces méthodes sont aussi comparées à une répartition en Round Robin poursuivie sur l'ensemble des requêtes du test. Les graphiques de la figure 7.2 montrent la



(a) CAWA K-moyennes

(b) CAWA avec classification hiérarchique

FIGURE 7.2 – Distribution cumulative des temps de réponses, 2 machines

distribution des temps de réponses simulés sur un système à deux machines avec un taux

d'arrivée de requêtes de 10 requêtes par seconde. Chaque courbe correspond à un algorithme de répartition et chaque point de la courbe désigne le pourcentage de requêtes dont le temps de réponse est inférieur à la valeur figurant en abscisse. Il est aussi possible de paramétrer le nombre de requêtes à observer avant de déclencher la première classification de CAWA, par conséquent, la légende indique pour chaque courbe les paramètres utilisés. Ces résultats montrent que certaines configurations de l'algorithme CAWA ne permettent pas d'améliorer les temps de réponses par rapport à un algorithme comme le Round Robin. Sur ces tests, la stratégie de répartition CAWA utilisant une classification des requêtes avec l'algorithme des K-moyennes obtient des temps d'exécution significativement meilleurs que la stratégie utilisant une classification hiérarchique ascendante. En outre, il ressort de ces tests que les quantités de requêtes les plus pertinentes pour procéder à la classification sont de 300 requêtes pour l'algorithme des k-moyennes et de 100 requêtes pour la classification hiérarchique. Les deux configurations permettent d'obtenir un meilleur niveau de service par rapport à une stratégie en Round Robin, bien que l'écart soit plus significatif pour la configuration utilisant l'algorithme des k-moyennes. Ces premiers résultats encourageants ne concernent cependant qu'une simulation avec deux machines. Dès lors, se pose la question du maintien de la performance de l'algorithme lorsque le nombre de machines ou que le nombre de requêtes différentes augmentent.

7.5.2 Robustesse de l'algorithme

Cette section étudie le comportement de l'algorithme CAWA face à un plus grand nombre de machines. Alors que l'algorithme semble fonctionner correctement pour deux machines, cette approche échoue à prendre en compte de nouvelles ressources dans sa déclinaison fondée sur une répaprtition initiale aléatoire.

Influence du nombre de machines Le graphique en figure 7.3 montre la distribution des temps de réponse obtenus par simulation sur une charge de travail similaire à celle utilisée pour les deux machines lors du test précédent. Le gain en performance obtenu par l'ajout de machines est notable, car les temps moyens sont bien inférieurs à ceux affichés lors du premier tests (graphiques 7.2b et 7.2a). Il apparaît cependant que la version de

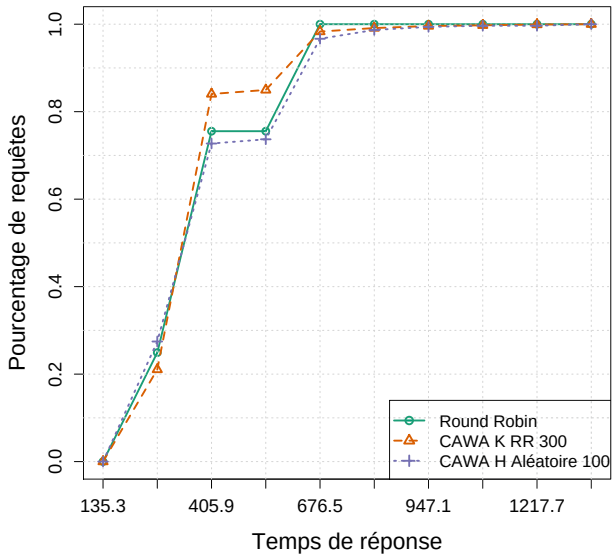


FIGURE 7.3 – CAWA : Distribution cumulative des temps de réponse, 10 machines

l'algorithme CAWA utilisant une stratégie de répartition initiale aléatoire, signifiée par la courbe avec des marqueurs en croix, ne semble pas gagner en efficacité, car les temps de réponse obtenus avec cette politique restent légèrement inférieurs à ceux obtenus avec une politique de type Round Robin tout au long du test. En revanche, l'algorithme CAWA utilisant une stratégie de répartition initiale en Round Robin (marqueurs en triangle) semble obtenir un écart de performance significatif par rapport à l'algorithme du Round Robin pur (marqueurs ronds).

Discussion et limites Ces résultats préliminaires sont encourageants, et dénotent que l'approche développée dans ces travaux peut fournir une amélioration significative des résultats de temps de réponses pour un petit nombre de machines. Lorsque le nombre de machines augmente, cependant, les écarts entre le Round Robin et CAWA se réduisent. Deux explications sont possibles pour ces ralentissements : la première et la plus simple consiste à étudier l'influence de la répartition initiale choisie par l'utilisateur. Dans les simulations décrites dans ce chapitre, il semble que le Round Robin soit une bonne stratégie initiale. Il convient donc d'établir des critères de choix en fonction du type de charge étudié

permettant de sélectionner la stratégie initiale la plus appropriée. La deuxième hypothèse, déjà évoquée au début de ce chapitre, est que le modèle de coût utilisé est biaisé. Le modèle actuel, bien que permettant de refléter correctement l'évolution du coût unitaire des requêtes, ne permet pas de pénaliser suffisamment la surcharge d'une machine : le coût baisse mécaniquement avec le nombre de requêtes affectées à chaque machine. Cet effet nuit à l'optimisation, car la matrice des coûts est faussée. Il conviendra donc de rechercher des modèles de couts ou des mécanismes de pénalisation appropriés pour établir une matrice des coûts plus précise.

La troisième limite de cette approche uniquement fondée sur le coût, est que l'information sur la localisation des données n'est pas préservée. A l'issue de la phase d'optimisation, il se peut que la nouvelle catégorie de requêtes affectée à une machine soit très éloignée de la catégorie précédente. Conséquemment, si des données avaient été mises en mémoire principale lors de l'exécution des requêtes précédant l'optimisation, le bénéfice de cette mise en cache sera perdu lors de l'exécution des requêtes suivantes. Il est nécessaire de définir des mécanismes permettant de conserver l'information des données consultées par chaque requête afin de réduire l'impact potentiel d'un changement de catégorie.

7.6 Vers une approche hybride : Localisation et optimisation des coûts

La mise au point d'une stratégie hybride mêlant les filtres de Bloom pour résumer le contenu et l'optimisation des coûts d'exécution est envisageable. Le développement de cette approche nécessite de définir une fonction permettant d'extraire l'information de localisation des données dans l'ensemble des filtres. Cette information peut ensuite être utilisée pour modifier la matrice des coûts afin de pénaliser les changements de catégories éventuellement induits par l'optimisation.

Une approche possible consisterait à utiliser l'algorithme BB-WACA lors de la répartition initiale des requêtes, afin de construire les filtres de Bloom nécessaires à la connaissance de la localisation des données.

Pour ce faire, une étape supplémentaire est créée dans l'algorithme. Cette nouvelle

étape, appelée la phase de *localisation*, précède l'exécution de l'algorithme Hongrois. Lors de cette nouvelle phase, la liste des requêtes utilisée pour établir la classification est parcourue. Pour chaque requête dans cette liste, l'index de filtres de Bloom décrit en section 6.6, est utilisé pour obtenir les identifiants des machines qui ont reçu la requête au cours de la phase de répartition précédente. Avec cette liste de machines, ainsi qu'avec l'index de la catégorie à laquelle appartient la requête, il est possible de réduire le coût associé à cette catégorie dans la matrice de coûts. Cette réduction de coût doit être proportionnelle au coût supplémentaire engendré par la lecture des données depuis le disque. Une fois que toutes les requêtes observées ont été examinées, la matrice des coûts mise à jour est passée à l'algorithme hongrois pour terminer la phase d'optimisation. A l'issue de cette phase, les coût sont donc modifiés en fonction de la localisation des données.

L'avantage de cette approche est qu'elle n'introduit pas de contraintes supplémentaires dans la résolution du problème d'optimisation, ce qui permet de conserver le bénéfice de la faible complexité de l'algorithme Hongrois. Cependant cette approche est encore spéculative, et il est nécessaire pour pouvoir l'utiliser de connaitre la quantité de données lue par chaque requête, une information qui n'est pas toujours disponible. Une implémentation de test ainsi que des expérimentations supplémentaires seront nécessaires pour établir la pertinence d'une telle approche, notamment son impact sur la complexité de l'algorithme.

7.7 Conclusion et travaux futurs

Les travaux développés dans ce chapitre et les résultats de simulations obtenus, montrent que l'approche par calcul de coûts pour évaluer la performance permet une certaine amélioration des temps de réponse obtenus par simulation. Cependant l'approche manque encore de maturité car le modèle de coût utilisé n'est pour le moment pas assez précis. De plus, l'approche utilisée pour mettre en œuvre cet algorithme n'est pas étudiée pour prendre en compte une grand nombre de machines et de classes. La classification est actuellement gourmande en terme de calculs et s'exécute après un certains nombres de requêtes. L'utilisation d'un algorithme de classification itératif permettrait de répartir la classification sur un plus grand nombre d'étapes, et d'initier des changements moins abrupts dans l'affectation des requêtes aux différentes machines. Une deuxième amélioration sera de ne plus avoir

à affecter les requêtes similaires à une seule machine mais à plusieurs, selon une certaine probabilité.

Enfin il est possible d'améliorer la phase d'optimisation en réduisant la complexité de l'algorithme d'affectation des tâches, en offrant une matrice de répartition finale moins rigide, fondée sur des probabilités. Le relâchement des contraintes d'affectation permettra ainsi d'obtenir plusieurs solutions probables permettant d'assurer une répartition plus équitable des requêtes lorsque la charge est importante ou variable. Les autres améliorations possibles de cet algorithme concernent la modélisation du coût des requêtes qui doit intégrer les différents postes de coûts possibles, comme l'espace disque utilisé ou la bande passante réseau, et non se fonder uniquement sur l'utilisation de la machine.

Chapitre 8

Conclusion

Les travaux de cette thèse s'inscrivent dans un projet de plateforme de collecte et de traitement de données multimédia, appelée MCube, et destinée à fournir des services de surveillance de cultures aux agriculteurs. Cette plateforme a été développée à partir de l'étude des logiciels existants et des contraintes propres de ce système. Les techniques modernes de collecte et de traitement des données massives permettent de fournir des services de plus en plus efficaces et pertinents dans la gestions des ressources, notamment dans le domaine de l'agriculture. La plateforme MCube fait donc partie intégrante des technologies émergentes en matière de services fondés sur l'analyse de données.

La masse de données impliquée dans ce projet impose de distribuer les traitements, de manière parallèle ou au fil de l'eau. le développement d'une plateforme dédiée appelée Cloudizer a été réalisée durant cette thèse. Cette plateforme assure la répartition de charge de plusieurs services Web en permettant à l'utilisateur de sélectionner la stratégie de répartition qu'il souhaite en fonction de ses besoins. Ce service a été élaboré pour l'étude de différentes stratégies de répartition de charge et a été utilisé pour les expérimentations présentées dans ce document.

Les recherches menées durant cette thèse ont porté sur la mise au point de deux nouvelles stratégies de répartition de charge répondant aux besoins des systèmes déployés sur des infrastructures d'informatique à la demande (les clouds). L'analyse de l'état de l'art des stratégies de répartition existantes a mis en évidence que deux types de stratégies sont adaptées à ces environnements : les stratégies fondées sur la localisation des données et les

stratégies fondées sur le coût.

Deux stratégies ont été développées : la stratégie WACA "Workload And Cache Aware Algorithm" (Algorithme Renseigné sur la Charge et le Cache), qui utilise des filtres de Bloom pour décrire la liste des données présentes sur les machines du système, est une stratégie tirant partie de la localisation des données, particulièrement efficace pour les services Web nécessitant des accès disques fréquents. Cette stratégie fournit de meilleurs temps de réponses que certaines stratégies de distribution de requêtes existantes comme le Round Robin ou le Hachage Cohérent, grâce à sa capacité d'adaptation. La stratégie WACA innove par l'utilisation d'un index permettant de réduire substantiellement les temps de recherche d'une donnée particulière dans les résumés. Les expérimentations menées sur cette stratégie ont démontré l'efficacité de cette nouvelle structure par rapport aux versions précédentes de l'algorithme.

La seconde approche développée est une stratégie fondée sur le coût financier d'exécution de requêtes de services. Cette stratégie, baptisée CAWA pour "Cost AWare Algorithm" (Algorithme Renseigné sur le Coût) utilise l'analyse de données pour établir des profils de requêtes et évaluer les coûts d'exécution respectifs de ces profils sur les différentes machines du système. La répartition utilise un algorithme d'optimisation afin de sélectionner la machine la plus appropriée pour chaque profil. Les études effectuées par simulation de cet algorithme mettent en évidence son efficacité à petite échelle, mais aussi les limites de cette approche lorsque le nombre de machines augmente. Ces limites, font apparaitre le besoin de développer une stratégie hybride mélangeant la localisation des données et l'optimisation des coûts.

Le développement de ces stratégies a été facilité par la mise au point d'un simulateur de plateforme d'infrastructure à la demande appelé Simizer. Ce système a émergé de deux besoins : premièrement faciliter la mise au point et le développement de nouvelles stratégies ou protocoles pour les applications distribuées, et deuxièmement l'absence de simulateur existant permettant de rendre ces services. Ce logiciel est d'ores et déjà utilisé dans un autre contexte : celui de l'étude de la cohérence de données dans un système de stockage distribué [SRSE13].

Ces travaux montrent qu'à l'heure du développement de nombreux services d'analyses

de données, les techniques de répartition de charge utilisées dans les principaux logiciels servant à ces traitements ne sont pas toujours suffisantes. La localisation des données joue un rôle clef dans une répartition efficace, mais la structure de donnée fournissant cette information doit être adaptative, afin de permettre une meilleure capacité de traitement des messages.

Perspectives de recherche et développements

Intégration des composants MCube

Les travaux sur le projet MCube devront faire l'objet de tests en situation réelle, à partir des données collectées durant ces trois dernières années. L'application d'analyse de données multimédia, nécessaire à un premier cas d'étude sera disponible à la fin du mois de décembre 2013. À ce jour, le dispositif de collecte des données à partir des passerelles multimédia souffre encore de défauts empêchant son utilisation en production.

L'intégration des deux plateformes MCube et Cloudizer nécessite des développements supplémentaires. En particulier, le système doit fournir aux utilisateurs l'accès aux données sur les traitements en cours d'exécution. Ces développements devront être achevés avant la fin du projet, qui aura lieu en mai 2014.

Développement des projets Simizer et Cloudizer

Les projets Cloudizer et Simizer peuvent encore faire l'objet d'améliorations. La publication en tant que projets libres de droits sera nécessaire afin d'en faciliter l'adoption et pourra faciliter la validation du projet Simizer par rapport à d'autres simulateurs existants tels que les projets SimGrid ou CloudSim. Le projet Cloudizer fera l'objet de plusieurs améliorations. L'objectif à court terme est l'ajout d'un mode de fonctionnement distribué du répartiteur, qui permettra d'assurer un meilleur passage à l'échelle du système, et de traiter un plus grand nombre de services. Le second développement du projet Cloudizer sera son intégration avec les plateformes de services d'infrastructures à la demande existantes, afin de permettre l'ajout dynamique de ressources de calcul ou de stockage.

Le simulateur Simizer fera lui aussi l'objet de plusieurs développements, notamment

au niveau de l'interface de programmation, ce qui autorisera la simulation d'un plus grand nombre d'applications. La modélisation de cas d'applications précis constitue aussi un effort de recherche nécessaire, notamment dans le cadre de l'étude des problèmes de cohérence de données pouvant survenir lors du passage à l'échelle de certains systèmes de stockage.

Liens avec la cohérence des données

Les systèmes de stockage de données sont aujourd'hui au cœur des problématiques de passage à l'échelle pour un grand nombre d'applications. L'optimisation des protocoles de réplication et de distribution des requêtes vont donc de pair avec les stratégies permettant de connaître rapidement la localisation des données dans un système distribué. La stratégie WACA est donc susceptible d'être utilisée dans ce contexte afin de permettre un accès rapide à la localisation des répliquas dans une base de données distribuée.

Le second type de système sensible à cette approche sont les services de cache distribués. Ces services reposent sur un ensemble de logiciels dont il serait intéressant d'étudier l'efficacité avec la stratégie WACA. La cohérence des caches est aussi très exigeante en matière de contraintes dans la mesure où les données ainsi stockées font l'objet de mises à jour fréquentes, et d'un grand nombre d'accès concurrents. De plus, les système de traitements de données massives en mémoire commencent à émerger, se posant en alternative aux systèmes optimisant les accès disques.

Répartition de charge dynamique et distribuée

La problématique principale des systèmes déployés sur les plateformes d'infrastructures informatique à la demande est le support de l'élasticité, c'est à dire la capacité d'un système à changer de taille à la demande et dynamiquement. Cette capacité est aussi liée à l'optimisation des coûts, car un système de répartition efficace permet de consolider plusieurs instances de services sur une machine.

Il est pour cela nécessaire de mettre en place les mécanismes appropriés au niveau du système d'exploitation pour supporter la réplication et le transfert de processus en cours d'exécution. La stratégie d'optimisation des coûts CAWA peut dans ce contexte s'avérer utile pour détecter les machines les plus efficaces pour traiter certaines requêtes.

De même l'adaptation de l'algorithme WACA à un environnement dynamique pourrait s'avérer intéressante du point de vue théorique, car dans ce contexte la localisation des données peut s'avérer critique pour réduire les temps de transferts nécessaires lors de la migration d'un processus.

La plateforme Cloudizer fournit le cadre logiciel nécessaire pour développer des stratégies de répartition plus dynamiques et non pas uniquement fondées sur la distribution de requêtes. Le but est de parvenir à développer un service de déploiement permettant de d'assurer la répartition du plus grand nombre d'applications possibles. Une répartition de charge dynamique et élastique nécessite le développement d'algorithmes de répartition distribués permettant le passage à l'échelle du système. Utiliser les filtres de Bloom et l'optimisation des coûts dans ce contexte peut s'avérer difficile. Il conviendra donc de développer un ensemble de protocoles et d'algorithmes permettant la diffusion efficace de l'information de localisation parmi l'ensemble des nœuds du système. La recherche sur les systèmes pairs à pairs pourra servir de base à ces développements.

Publications

Conférences Internationales

[1] Sathiya Prabhu Kumar, Raja Chiky, Sylvain Lefebvre, and Eric Gressier Soudan. Libre : A consistency protocol for modern storage systems. In *Proceedings of the 6th ACM India Computing Convention*, Compute '13, pages 8 :1–8 :9, New York, NY, USA, 2013. ACM.

[2] S. Lefebvre, R. Chiky, and K.S. Prabhu. Waca : Workload and cache aware load balancing policy for web services. In *Systems and Computer Science (ICSCS), 2012 1st International Conference on*, pages 1–6, 2012.

[3] Sylvain Lefebvre. Indexed bloom filters for web caches summaries. In Costin Badica, Ngoc Thanh Nguyen, and Marius Brezovan, editors, *Computational Collective Intelligence. Technologies and Applications*, volume 8083 of *Lecture Notes in Computer Science*, pages 507–516. Springer Berlin Heidelberg, 2013. 111

Revue Nationale

[4] Sylvain Lefebvre, Sathya Prabhu Kumar, and Raja Chiky. Waca : Politique de répartition de charge des services web dans une architecture de type cloud. *Revue des Nouvelles Technologies de l'Information*, Ingénierie des protocoles et Nouvelles technologies de la répartition : sélection dárticles de CFIP/NOTERE 2012, RNTI-SM-2 :79–98, 2013. 6, 111

Conférences Nationales

[5] Sylvain Lefebvre, Raja Chiky, and Renaud Pawlak. Cloudizer : framework de distribution des services dans une architecture de type cloud (démonstration). *12e Conférence Internationale Francophone sur l'Extraction et la Gestion des Connaissances*, 2012.

[6] R. Pawlak, S. Lefebvre, Z. Kazi-Aoul, and R. Chiky. Cloud elasticity for implementing an agricultural weather service. In *New Technologies of Distributed Systems (NOTERE), 2011 11th Annual International Conference on*, pages 1–8, 2011.

Annexe A

Cloud Computing : Services et offres

A.1 Définition

Le Cloud Computing ou "l'informatique en nuage" est défini par le National Institute of Standards and Technologies (NIST), comme un modèle de service mutualisé, accessible par le réseau, qui permet aux organisations de déployer rapidement et à la demande des ressources informatiques [MG11].

Le modèle économique des plateformes de Cloud Computing repose sur le partage des ressources entre un grand nombre d'utilisateurs. Ce modèle est appelé multi-tenant. Le fournisseur de service parvient à assurer des économies d'échelle en partageant ses ressources propres entre un nombre important d'utilisateurs. Un ensemble de technologies permettent de parvenir à une utilisation optimale des ressources. La virtualisation des serveurs, par exemple, permet aux fournisseurs de partager leurs machines physiques entre plusieurs utilisateurs opérant des machines virtuelles. Le système de virtualisation permet d'assurer l'isolation et le partage équitable des ressources disponibles entre les différents hôtes. Des technologies comme les réseaux définis par logiciel (Software Defined Networking) permettent d'assurer l'isolation des trafics réseaux des différents utilisateurs.

A.2 Modèles de services

La définition du NIST retient trois modèles de services de Cloud Computing, définissant le types de ressources et de flexibilité que l'utilisateur possède.

A.2.1 L'Infrastructure à la Demande (Infrastructure as a Service)

Dans ce modèle, un fournisseur met à disposition de ses utilisateurs des machines virtuelles, louées pour une certaine période de temps renouvelable tacitement. Les machines peuvent être demandées via une interface ou un service web, afin d'automatiser le processus d'allocation.

L'utilisateur a donc la maitrise de l'ensemble de la configuration logicielle des machines virtuelles qu'il loue. Il peut installer les programmes et services qu'il souhaite, sans restrictions, sur ces machines. La plupart des plateformes fournissent aussi un annuaire d'images systèmes prêtes à déployer et permettent aux utilisateurs de créer leurs propres images afin de faciliter le déploiement de nouvelles machines.

Les plateformes d'infrastructures à la demande les plus utilisées actuellement sont les services "Elastic Compute Cloud" d'Amazon, et le service de Rackspace, son principal concurrent dans ce domaine [1]

A.2.2 La Plateforme à la demande (Platform as a Service)

Les services de plateformes à la demande mettent à disposition de leurs utilisateurs un canevas logiciel ou un service précis, via une interface programmation définie ou un ensemble de services Web. Le fournisseur assure la gestion des ressources sous-jacentes en fonction de la demande de l'utilisateur, et facture son service en fonction de l'utilisation qui en a été faite.

Par exemple, le service Cloud Foundry [2] propose à ses utilisateurs de déployer automatiquement leurs applications web sur un ensemble de machines. Le service assure ensuite la répartition de charge automatiquement entre les différentes machines en fonctions des accès à l'application, et facture un prix pour la location des machines.

D'autres types de services existent dans ce modèle. Il est possible de trouver des services de bases de données réparties, par exemple le service AppEgine de Google [3], qui fournit

1. `http://www.slideshare.net/FrostandSullivan/cloud-infrastructure-as-a-service-market-update`, consulté le 5 octobre 2013
2. `http://docs.cloudfoundry.com/`, consulté le 5 octobre 2013
3. `https://developers.google.com/appengine/?csw=1`, consulté le 5 octobre 2013

une API permettant de stocker et requêter des données dans son service.

Ces services permettent à leurs utilisateurs d'utiliser des canevas logiciels, intergiciels, ou autres composants en s'abstrayant des aspects de gestion de la configuration et des ressources inhérents à ces types de déploiement.

A.2.3 Le logiciel à la demande (Software As a Service)

Les services de logiciels à la demande ont émergé avec le développement des applications "web". Ces applications fournissent l'expérience d'un client lourd à travers une page web. Elles sont généralement hébergées chez le fournisseur de service lui même, ce qui permet à l'utilisateur de ne pas avoir à gérer les mises à jour du logiciel ou le stockage des données utilisateurs. Les exemples les plus connus de ces services sont le service de courriel GMail [4], de Google, ou le service de gestion de relation client "salesforce.com" [5].

A.3 Amazon Web Services

TABLE A.1 – Caractéristiques des instances EC2

Type	**Quantité de Mémoire (Go)**	**Unités de calcul EC2 (CPU)**	**Espace Disque (Go)**	**Plateforme**	**Performance E/S**
m1.small	1.7	1	160	32 bit	Modérée
m1.large	7.5	4	850	64 bit	Elevée
m1.xlarge	15	8	1 690	64 bit	Elevée
t1.micro	613 Mo	2	10	32 ou 64 bits	Basse

L'offre de services en nuage d'Amazon est actuellement une des seules couvrant tout le spectre des différentes offres de services Cloud, et la plus mature des plateformes d'infrastructure à la demande. Ce service, nommé EC2 (Elastic Compute Cloud) permet de louer des machines virtuelles facturées à l'heure, pour des prix dépendants de la capacité de calcul (fréquence et nombre des processeurs) de la quantité de mémoire vive et de la rapidité des accès disque. Le tableauA.1 suivant résume quelques exemples de configurations types disponibles sur ce service.

4. `https://gmail.com`, consulté le 5 octobre 2013
5. `http://www.salesforce.com/fr/?ir=1`, consulté le 5 octobre 2013

L'offre de produits Cloud d'Amazon est complétée par un ensemble de services de niveau plateforme. Par exemple, il est possible d'accéder à un service de base de données relationnelle et un service stockage clef-valeur, un service de mise en cache et un service d'analyse de données fondé sur le canevas Hadoop. Un ensemble de service de gestion de flux de travail et de stockage est aussi fourni aux utilisateurs, et facturé en fonction du nombre d'accès ou de la quantité de données stockées.

L'avantage de l'écosystème d'Amazon est que tous ces systèmes sont accessibles via des services web, ce qui permet de déployer rapidement des systèmes complexes en combinant ces différents services.

Annexe B

Code des politiques de répartition

Dans cette annexe, nous incluons des extraits de code décrivant l'implémentation en langage JAVA des politiques de répartition BB-WACA et CAWA décrites respectivement dans les sections 6.6 et 7.2.

B.1 Implémentation de WACA

L'ensemble de la logique de l'algorithme BB-WACA se concentre dans la méthode *loadBalance*, décrite ci-dessous.

Listing B.1 – Implémentation de la stratégie WACA

```
public Node loadBalance(Request r) {

        BlockBloomFilter<ServerNode> targetBf= null;
        ServerNode target = null;

        //1. Hachage de la requete
        int[] hash = hash(r.getParameters(),kMax);
        int firstB = hash[0] % mMax;

        //2. Verifie si une liste existe avec cette requete
        if(firstBlocks[firstB] == null)
            firstBlocks[firstB] =
                new LinkedList<BlockBloomFilter<ServerNode>>();

        int groupSz = firstBlocks[firstB].size();

        if(groupSz > 0) {
        //2.1 Si une liste est trouvee, on itere sur la liste
```

```
        Iterator<BlockBloomFilter<ServerNode>> it =
                    firstBlocks[firstB].iterator();

        while(it.hasNext()) {

            BlockBloomFilter<ServerNode> bf = it.next();

            //2.1.1 Verification du filtre
            //et de la charge de la machine

            boolean lookup = bf.lookup(hash,1)
                    , loadChk = checkLoad(bf.getData());

            if(lookup && loadChk) {
                target = bf.getData();
                updateHeap(target,target.getRequestCount() +1);

                return target;

            } else if(loadChk) {

                target = getLeastLoaded(target, bf.getData());
            }
        }
    }

    // 2.1.2 target == null si aucun filtre
    // ne contient la requete

    if(target!=null) {
        nodeTab.get(target.getId()).getData().insert(hash);
        updateHeap(target,target.getRequestCount() +1.0);
        return target;
    }

    //2.3 Selection de la machine la moins chargee,
    // et insertion dans le filtre correspondant,
    // ainsi que dans l'index.
    FNode<BlockBloomFilter<ServerNode>> tgtFnode = heap.min();
    targetBf =tgtFnode.getData();

    if(!firstBlocks[firstB].contains(targetBf))
        firstBlocks[firstB].addFirst(targetBf);
```

```
        targetBf.insert(hash);
        target = targetBf.getData();
        updateHeap(target, target.getRequestCount()+1.0);

        return target;
    }
```

B.2 Implémentation de l'algorithme CAWA

L'extrait de code suivant contient les deux fonctions principales nécessaires à l'algorithme CAWA. La fonction *clusterNode* recherche le centre le plus proche de la requête en cours et renvoie la machine associée à ce centre. Si la classification n'a pas encore été exécutée par l'algorithme, alors la répartition se fait soit selon un algorithme en Round Robin, soit selon une sélection aléatoire.

La fonction *loadBalance* doit renvoyer la machine sélectionnée par l'algorithme, elle appelle donc la fonction *clusterNode* pour récupérer la machine cible. Elle incrémente ensuite le compteur de requêtes associé à cette machine afin de permettre le calcul des coûts nécessaire à l'optimisation.

Listing B.2 – Implémentation de la stratégie CAWA

```
    private Node clusterNode(Request r){
        Node targetNode;

        double dist = Double.MAX_VALUE;

        // Si il n'ya pas encore eu de classification
        if(!CLUSTERED) {

                if(RANDOM) {
            targetNode =
                nodeList.get(random.nextInt(nodeList.size()));
            } else {
            targetNode =
              nodeList.get(counter++ % nodeList.size());
            }

        } else {

        // recherche du centre le plus proche de la requete
```

```
            Vector req = r.requestToVectorH();
            Vector k = null;

             for(Vector v: protoToNode.keySet()) {

                  double tmpDist = v.distanceTo(req);
                  if(tmpDist < dist) {
                       k =v;
                       dist = tmpDist;
                  }
             }
             targetNode = protoToNode.get(k);
        }
        return targetNode;
    }

    @Override
    public Node loadBalance(Request r) {

        Node targetNode = clusterNode(r);

        if(counter % REQUEST_THRESOLD == 0)
             Arrays.fill(nodeCount, 0);

                  // Compte le nombre de requetes par machine
                  // pour calculer le cout moyen des requetes
        nodeCount[targetNode.getId()]++;

        return targetNode;
    }
```

Annexe C

CV

Etudes et diplômes

Depuis Octobre 2010 : Thèse de doctorat, Conservatoire National des Arts et Métiers, Paris,

Sujet : *"Services de répartition de charge pour le Cloud : application au traitement de données multimédias"*

Direction : Eric GRESSIER-SOUDAN (CEDRIC-CNAM) et Raja CHIKY (LISITE-ISEP)

Laboratoire : LISITE-ISEP

2010 : Master 2 MIAGE parcours Ingéniérie des Systèmes Informatiques Distribués, en apprentissage ESIAG, "École Supérieure d' Informatique Appliquée à la Gestion (ESIAG)", Université de PARIS XII Créteil.

2009 : Master 1 Méthodes Informatiques Appliquées à la Gestion des Entreprises, en apprentissage, ESIAG.

2008 : Licence 3 Economie et Gestion, Mention MIAGE, en apprentissage. ESIAG, avec mention.

2007 : BTS Informatique de Gestion, option développeur d'application : Lycée Polyvalent Le Rebours, 75013 PARIS.

2005 : Baccalauréat Economique et Social, Lycée Saint-Michel de Picpus, 75012 PARIS.

Expérience Professionnelle

2007-2010 : Apprentissage, Essilor International, Vincennes, France : Infrastructure & Operations Services : support de niveau 3 : Supervision Windows server, configuration management.

2006-2007 : Stages développement WEB ASP / PHP / JavaScript, IC Entreprise (94), SYKIO (93)

Enseignement

ISEP : Tutorat : Développement Web, gestion de projet (188h/an eq. TD).

CNAM : 3 cours et un TP sur les algorithmes et techniques de répartition de charge, niveau Master 1 et 2. (16h eq. TD)

Compétences Techniques

Administration Système : Linux et Windows, virtualisation.

Cloud Computing : Connaissance d'Amazon Web Services (EC2 et ELB), Cloud Foundry.

Développement : UML, gestion de projet.

Langages de programmation : Maîtrise de Java, connaissances en PHP, C, C++, powershell, R.

Frameworks : Grails, Hadoop.

Bibliographie

[AaC09] Marcos Dias De Assunção and Alexandre Costanzo. Evaluating the Cost-Benefit of Using Cloud Computing to Extend the Capacity of Clusters Categories and Subject Descriptors. *Distributed Computing*, 2009. xix, 71, 72, 114, 19

[Aba12] D. Abadi. Consistency tradeoffs in modern distributed database system design : Cap is only part of the story. *Computer*, 45(2) :37–42, 2012. 60

[ABPH07] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6) :255–261, March 2007. 82

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds : A berkeley view of cloud computing. Technical report, 2009. 5

[AGP+08] Nicolas ALLEZARD, Jean-Pierre GUIGNARD, Olivier PIETQUIN, Florence SEDES, Jean-François SEIGNOLE, and Jean-François SULZER. Architecture générique de stockage multimédia réparti avec recherche et indexation distribuées (projet itea2 lindo). 2008. 27

[Ama12] Amazon. Amazon elastic compute cloud (amazon ec2). http ://aws.amazon.com/ec2/, 2012. 60, 71

[App08] A Appleby. Murmurhash, 2008, 2008. 86

[BB12] Anton Beloglazov and Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic

consolidation of virtual machines in cloud data centers. *Concurr. Comput. : Pract. Exper.*, 24(13) :1397–1420, September 2012. 42

[BCC+99] Lee Breslau, Pei Cue, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions : Evidence and implications. In *In INFOCOM*, pages 126–134, 1999. 44

[BCMS11] Michaela Brut, Dana Codreanu, Ana-Maria Manzat, and Florence Sedes. Adapting indexation to content, context and queries characteristics in distributed multimedia systems. In *Seventh International Conference on Signal Image Technology and internet-Based Systems*, 2011. 17, 21, 27

[BGPS05] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Gossip algorithms : Design, analysis and applications. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 1653–1664. IEEE, 2005. 64

[BKB98] Ladjel Bellatreche, Kamalakar Karlapalem, and Gopal K. Basak. Horizontal class partitioning for queries in object-oriented databases, 1998. 68

[BKB07] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 119–128. IEEE, 2007. 62

[BLM+12] Laurent Bobelin, Arnaud Legrand, David Alejandro Gonzàlez Màrquez, Pierre Navarro, Martin Quinson, Frédéric Suter, and Christophe Thiery. Scalable multi-purpose network representation for large scale distributed system simulation. In *Proceedings of the 12th IEEE International Symposium on Cluster Computing and the Grid*, CCGrid'12. IEEE Computer Society Press, 2012. 42

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7) :422–426, July 1970. 6, 69, 79, 80

[BMHS13] Martin Becker, Juergen Mueller, Andreas Hotho, and Gerd Stumme. A generic platform for ubiquitous and subjective data. In *1st International*

Workshop on Pervasive Urban Crowdsensing Architecture and Applications, PUCAA 2013, Zurich, Switzerland – September 9, 2013. Proceedings, pages New York, NY, USA. ACM, 2013. Accepted for publication. 1

[Bon90] F. Bonomi. On job assignment for a parallel system of processor sharing queues. *IEEE Transactions on Computers*, 1990. 65, 88

[Bra00] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. 11

[BS10] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, MMSys '10, pages 35–46, New York, NY, USA, 2010. ACM. 4, 57, 80

[Can11] Rankloud : A scalable ranked query processing framework on hadoop. In *EDBT*, 2011. 14, 18, 19

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable : a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association. 13, 16

[CK88] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions on*, 14(2) :141–154, 1988. 60

[CLQ08] Henri Casanova, Arnaud Legrand, and Martin Quinson. Simgrid : a generic framework for large-scale distributed experiments. In *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, UKSIM '08, pages 126–131, Washington, DC, USA, 2008. IEEE Computer Society. 5, 41, 42

[Cor07] Kyle Cordes. Youtube scalability, 2007. 13, 16

[CSC09] Kasturi Chatterjee, S. Masoud Sadjadi, and Shu-Ching Chen. A distributed multimedia data management over the grid. *Multimedia Services in Intelligent Environments - Integrated Systems*, 2009. 12, 15, 23

[DCGV02] Ronald P Doyle, Jeffrey S Chase, Syam Gadde, and Amin M Vahdat. The trickle-down effect : Web caching and server request distribution. *Computer Communications*, 25(4) :345–356, 2002. 45

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. pages 1–13, 2004. 6, 60, 67

[Döl04] Mario Döller. *THE MPEG-7 MULTIMEDIA DATABASE SYSTEM (MPEG-7 MMDB)*. PhD thesis, 2004. 14

[DSASSLP12] David Dominguez-Sal, Josep Aguilar-Saborit, Mihai Surdeanu, and Josep Lluis Larriba-Pey. Using Evolutive Summary Counters for Efficient Cooperative Caching in Search Engines. *IEEE Transactions on Parallel and Distributed Systems*, 23(4) :776–784, April 2012. 60, 74

[DSSASLP08] David Dominguez-Sal, Mihai Surdeanu, Josep Aguilar-Saborit, and Josep Lluis Larriba-Pey. Cache-aware load balancing for question answering. In *Proceedings of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 1271–1280, New York, NY, USA, 2008. ACM. 6, 70

[EJ01] D. Eastlake, 3rd and P. Jones. Us secure hash algorithm 1 (sha1), 2001. 86, 118

[FCAB98] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache : A scalable wide-area web cache sharing protocol. Technical report, 1998. 6, 69, 81, 82

[Fei02] DrorG. Feitelson. Workload modeling for performance evaluation. In MariaCarla Calzarossa and Salvatore Tucci, editors, *Performance Evaluation of Complex Systems : Techniques and Tools*, volume 2459 of *Lecture Notes in Computer Science*, pages 114–141. Springer Berlin Heidelberg, 2002. 44, 45, 51, 100, 102, 109, 99

[FFH12] Florian Fittkau, Sören Frey, and Wilhelm Hasselbring. Cdosim : Simulating cloud deployment options for software migration support. In *Proceedings of the 6th IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA 2012)*, pages 37–46. IEEE Computer Society, September 2012. doi : 10.1109/MESOCA.2012.6392599. 42

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Phd thesis, University of California, 2000. 10, 31, 79, 88

[Fit04] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124) :5–, August 2004. 103, 102

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3) :596–615, July 1987. 97

[GBB+08] Patrick Giroux, Stephan Brunessaux, Sylvie Brunessaux, Jérémie Doucy, Gérard Dupont, Bruno Grilheres, Yann Mombrun, and Arnaud Saval. Weblab : An integration infrastructure to ease the development of multimedia processing applications. In *International Conference "Software & Systems Engineering and their Applications"*, 2008. 17, 18, 27

[GBKYKS03] Phillip B. Gibbons, Suman Nath Brad Karp Yan Ke, and Srinivasan Seshan. Irisnet : An architecture for a worldwide sensor web. *Pervasive Computing*, 2003. 15

[GJTP12] Katja Gilly, Carlos Juiz, Nigel Thomas, and Ramon Puigjaner. Adaptive admission control algorithm in a qos-aware web system. *Inf. Sci.*, 199 :58–77, September 2012. xix, 60, 63, 64, 102, 19

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2) :51–59, 2002. 16

[GPS11] Sandip Kumar Goyal, RB Patel, and Manpreet Singh. Adaptive and dynamic load balancing methodologies for distributed environment : a review.

International Journal of Engineering Science and Technology (IJEST), 3(3) :1835–1840, 2011. xix, 60, 61, 63, 64, 19

[GTC+07] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. Does internet media traffic really follow zipf-like distribution ? In *SIGMETRICS*, volume 7, pages 359–360, 2007. 45

[HCK+07] Felix Hupfeld, Toni Cortes, Bjoern Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. Xtreemfs : a case for object-based storage in grid data management. In *3rd VLDB Workshop on Data Management in Grids, co-located with VLDB*, volume 2007, 2007. 14

[Hil86] W D Hillis. *The connection machine*. MIT Press, Cambridge, MA, USA, 1986. 62, 65

[HSD11] Jonathon S. Hare, Sina Samangooei, and David P. Dupplaw. Openimaj and imageterrier : Java libraries and tools for scalable multimedia analysis and indexing of images. In *Proceedings of the 19th ACM international conference on Multimedia*, MM '11, pages 691–694, New York, NY, USA, 2011. ACM. 11, 12, 14, 19, 20

[IOY+11] A. Iosup, S. Ostermann, M.N. Yigitbasi, R. Prodan, T. Fahringer, and D. H J Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6) :931–945, 2011. 4, 57, 80

[KBAK10] D. Kliazovich, P. Bouvry, Y. Audzevich, and S.U. Khan. Greencloud : A packet-level simulator of energy-aware cloud computing data centers. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–5, 2010. 42

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees : distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM. 6, 68, 74

[KMGS09] Aaron Kaplan, Jonathan Mamou, Francesco Gallo, and Benjamin Sznajder. Multimedia feature extraction in the sapir pro ject. In *GSCL*, 2009. 18

[Kos05] Multimedia database systems : Where are we now ? In *IASTED International Conference on DATABASES AND APPLICATIONS*, 2005. 13, 14

[KR04] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures - SPAA '04*, page 36, 2004. 6, 61, 74, 103, 102

[Kuh55] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2) :83–97, 1955. 115, 117, 122

[KW00] Poul Henning Kamp and Robert N. M. Watson. Jails : Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000. 37

[Lan01] Douglas Laney. 3D data management : Controlling data volume, velocity, and variety. Technical report, META Group, February 2001. 2

[Lef13] Sylvain Lefebvre. Indexed bloom filters for web caches summaries. In Costin Badica, Ngoc Thanh Nguyen, and Marius Brezovan, editors, *Computational Collective Intelligence. Technologies and Applications*, volume 8083 of *Lecture Notes in Computer Science*, pages 507–516. Springer Berlin Heidelberg, 2013. 111

[LKC13] Sylvain Lefebvre, Sathya Prabhu Kumar, and Raja Chiky. Waca : Politique de répartition de charge des services web dans une architecture de type cloud. *Revue des Nouvelles Technologies de l'Information*, Ingénierie des protocoles et Nouvelles technologies de la répartition : sélection dárticles de CFIP/NOTERE 2012, RNTI-SM-2 :79–98, 2013. 6, 111

[LM09] Avinash Lakshman and Prashant Malik. Cassandra : structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 5–5, New York, NY, USA, 2009. ACM. 69, 103, 102

[LMHJ10] Kevin Lee, David Murray, Danny Hughes, and Wouter Joosen. Extending sensor networks into the cloud using amazon web services. In *Networked*

Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on, pages 1–7. IEEE, 2010. 1

[LW09] Huan Liu and Sewook Wee. Web server farm in the cloud : Performance evaluation and dynamic architecture. In MartinGilje Jaatun, Gansen Zhao, and Chunming Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 369–380. Springer Berlin Heidelberg, 2009. 3

[LXK+11] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James Larus, and Albert Greenberg. Join-idle-queue : A novel load balancing algorithm for dynamically scalable web services. *The 29th International Symposium on Computer Performance, Modeling, Measurements and Evaluation*, 2011. 65

[M+67] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, page 14. California, USA, 1967. 116, 120

[MB02] Manzur Murshed and Rajkumar Buyya. Using the gridsim toolkit for enabling grid computing education. In *Proc. of the Int. Conf. on Communication Networks and Distributed Systems Modeling and Simulation*, pages 18–24, 2002. 41

[MFF] S. Mccanne, S. Floyd, and K. Fall. ns2 (network simulator 2). http ://www-nrg.ee.lbl.gov/ns/. 42

[MG11] Peter Mell and Timothy Grance. The nist definition of cloud computing (draft). *NIST special publication*, 800(145) :7, 2011. 1, 5, 139, 149

[MH11] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011. 72, 114

[MHCD10] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. Towards characterizing cloud backend workloads : insights from google compute clusters. *SIGMETRICS Perform. Eval. Rev.*, 37(4) :34–41, March 2010. 117

[Mol07] Ingo Molnár. Cfs scheduler, 2007. 48

[Mpe12a] Mpeg-21 standard specification, http ://mpeg.chiariglione.org/standards/mpeg-21, 2012. 14

[MPE12b] Mpeg-7 standard specification, http ://mpeg.chiariglione.org/standards/mpeg-7, 2012. 14, 16

[Mun57] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial & Applied Mathematics*, 5(1) :32–38, 1957. 115, 117, 122

[Nak04] S. Nakrani. On Honey Bees and Dynamic Server Allocation in Internet Hosting Centers. *Adaptive Behavior*, 12(3-4) :223–240, December 2004. 62

[Nie93] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. 93

[NVPC+12] Alberto Núñez, JoseL. Vázquez-Poletti, AgustinC. Caminero, GabrielG. Castañé, Jesus Carretero, and IgnacioM. Llorente. icancloud : A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10(1) :185–209, 2012. 42

[OA03] A Osman and H Ammar. Dynamic load balancing strategies for parallel computers. 2003. 60, 62, 63

[PAB+98] Vivek S Pail, Mohit Aront, Gaurav Bangat, Michael Svendsent, Peter Druschelt, Willy Zwaenepoelt, and Erich Nahumq. Locality-Aware Request Distribution in Cluster-based Network Servers. *Distribution*, pages 205–216, 1998. 67, 68

[PLCKA11] Renaud Pawlak, Sylvain Lefebvre, Raja Chiky, and Zakia Kazi-Aoul. L'elasticité du cloud computing au service de la météo agricole de précision. to appear in NOTERE 2011, 2011. 6, 78

[R C13] R Core Team. *R : A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. 53

[RGO12] Arnon Rotem-Gal-Oz. Fallacies of Distributed Computing Explained. Technical report, 2012. 60

[Riv91] Ronald L. Rivest. The MD5 Message-Digest algorithm (RFC 1321). 1991. 85

[RLTB10] Martin Randles, David Lamb, and a. Taleb-Bendiab. A Comparative Study into Distributed Load Balancing Algorithms for Cloud Computing. *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, pages 551–556, 2010. 62

[RNCB11] Anton Beloglazov Cesar A. F. De Rose Rodrigo N. Calheiros, Rajiv Ranjan and Rajkumar Buyya. Cloudsim : A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software : Practice and Experience (SPE)*, January 2011. 5, 41, 42

[RPC10] Jennie Rogers, Olga Papaemmanouil, and Ugur Ctintemel. A generic auto-provisioning framework for cloud databases. *Data Engineering Workshops (ICEDEW), 2010 IEEE 26th International Conference on*, 2010. 73, 114, 117, 122

[SCR+00] S Shepler, B Callaghan, D Robinson, R Thurlow, C Beame, M Eisler, Zambeel Inc, and D Noveck. Nfs version 4 protocol, 2000. 14

[SDK+11] F. Stiegmaier, M. Doeller, H. Kosch, A. Hutter, and T. Riegel. Air : Architecture for interoperable retrieval on distributed and heterogeneous multimedia repositories. *Analysis, Retrieval and Delivery of Multimedia Contents*, 2011. 16

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, 0 :1–10, 2010. 13, 14, 25, 26, 33, 63, 67, 78, 79

[SL12] Sathiya Prabhu Kumar Sylvain Lefebvre, Raja Chiky. Waca : Workload and cache aware load balancing policy for web services. 2012. 6, 111

[SL13] Georgia Sakellari and George Loukas. A survey of mathematical models, simulation approaches and testbeds used for research in cloud computing. *Simulation Modelling Practice and Theory*, 2013. 5, 42

[SRSE13] Prabhu K Sathiya, Chiky Raja, Lefebvre Sylvain, and Gressier Soudan Eric. Libre : A consistency protocol for modern storage systems (to appear). In *ACM Compute'13*. ACM, 2013. 132

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive : a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2) :1626–1629, August 2009. 25

[UIM12] http ://uima.apache.org/. 2012. 18

[Vaj09] Peter Vajgel. Needle in a haystack : efficient storage of billions of photos. 2009. 16

[Whi09] Tom White. *Hadoop : The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. 14, 17, 19, 22, 36, 67, 78

[Xiv13] Xively. Xively is the public cloud specifically built for the internet of things. `https://xively.com/whats_xively/`, 2013. 1

[YL11] Zhen Ye and Shanping Li. A request skew aware heterogeneous distributed storage system based on cassandra. In *Computer and Management (CAMAN), 2011 International Conference on*, pages 1–5, 2011. 69

[YS07] B. Yagoubi and Y. Slimani. Task Load Balancing Strategy for Grid Computing. *Journal of Computer Science*, 3(3) :186–194, March 2007. 60, 63, 65

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing : state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1) :7–18, 2010. 5

[ZCF+10] Matei Zaharia, N. M. Mosharaf Chowdhury, Michael Franklin, Scott Shenker, and Ion Stoica. Spark : Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley, May 2010. 77, 78

[ZRS+05] Qi Zhang, Alma Riska, Wei Sun, Evgenia Smirni, and Gianfranco Ciardo. Workload-aware load balancing for clustered web servers. *Parallel and Distributed Systems, IEEE Transactions on*, 16(3) :219–233, 2005. 66

Glossaire

- $BB\text{-}WACA$: Block Based Workload And Cache Aware Algorithm
- $CAWA$: Cost AWare Algorithm
- $CNAM$: Conservatoire National des Arts et Métiers
- $EC2$: Elastice Compute Cloud
- FB : Filtre de Bloom
- FBC : Filtre de Bloom avec Compteur
- FBB : Filtre de Bloom en Blocs
- $REST$: Representational State Transfert
- $WACA$: Workload And Cache Aware algorithm

le cnam

Sylvain LEFEBVRE

Services de répartition de charge pour le Cloud : Application au traitement de données multimédia

le cnam

Résumé :
Le travail de recherche mené dans cette thèse consiste à développer de nouveaux algorithmes de répartition de charge pour les systèmes de traitement de données massives. Le premier algorithme mis au point, nommé "WACA" (Workload and Cache Aware Algorithm) améliore le temps d'exécution des traitements en se basant sur des résumés de contenus. Le second algorithme, appelé "CAWA" (Cost Aware Algorithm) tire partie de l'information de coût disponible dans les plateformes de type "Cloud Computing" en étudiant l'historique d'exécution des services.
L'évaluation de ces algorithmes a nécessité le développement d'un simulateur d'infrastructures de "Cloud" nommé Simizer, afin de permettre leur test avant le déploiement en conditions réelles. Ce déploiement peut se faire de manière transparente grâce au système de distribution et de surveillance de service web nommé "Cloudizer", développé aussi dans le cadre de cette thèse. Ces travaux s'inscrivent dans le cadre du projet de plateforme de traitement de données Multimédia for Machine to Machine (MCUBE), dans le lequel le canevas Cloudizer est mis en œuvre.

Mots clés :
Répartition de charge, Cloud, Données Massives

Abstract :
The research work carried out in this thesis consists in the development of new load balancing algorithms aimed at big data computing. The first algorithm, called « WACA » (Workload and Cache Aware Algorithm), enhances response times by locating data efficiently through content summaries. The second algorithm, called CAWA (Cost AWare Algorithm) takes advantage of the cost information available on Cloud Computing platforms by studying the workload history.
Evaluation of these algorithms required the development of a cloud infrastructure simulator named Simizer, to enable testing of these policies prior to their deployment. This deployment can be transparently done thanks to the Cloudizer web service distribution and monitoring system, also developed during this thesis. These works are included in the Multimedia for Machine to Machine (MCUBE) project, where the Cloudizer Framework is deployed.

Keywords :
Load balancing, Cloud Computing, Big data