WACA: Politique de répartition de charge des services web dans une architecture de type Cloud

Sylvain Lefebvre*, Sathya Prabhu Kumar *, Raja Chiky *

*LISITE - ISEP 28, rue notre des Champs, 75006, PARIS
prénom.nom@isep.fr,
http://www.isep.fr

Résumé. De plus en plus d'applications font appel à des services web déployés sur des architectures de type cloud. Ces services peuvent traiter des données massives et des fichiers en grandes quantités pour répondre à un nombre croissant de demandes (requêtes) des utilisateurs. Ces données peuvent être de différents types allant des données multimédia aux données numériques provenant de capteurs. En plus de la volumétrie des données, les applications web doivent également gérer la satisfaction des utilisateurs ce qui impose une garantie de la qualité du service fourni.

Pour ces raisons, la répartition des traitements sur une grappe de machines devient nécessaire pour un grand nombre d'applications. L'efficacité de ces applications dépendra particulièrement de la politique de répartition de charge utilisée.

Cet article présente une nouvelle politique de répartition de charge appelée WACA (Workload And Cache Aware load balancing) qui prend en compte la charge des machines et le contenu de leur cache. Cette politique permet de réduire le temps d'exécution des requêtes des utilisateur en se basant sur des résumés de cache construits sous forme de filtre de Bloom. WACA est un algorithme de répartition de charge centralisé qui affecte les requêtes vers la machine susceptible de contenir les données visées dans son cache en évitant de la surcharger. La politique proposée a été développée et évaluée en utilisant un framework de répartition des applications web appelé CLOUDIZER et la stratégie a été comparée à des algorithmes de répartition de charge traditionnels.

1 Introduction

La répartition de charge consiste à affecter des tâches à différents noeuds d'exécution afin d'éviter la surcharge d'une ou plusieurs machines, dans le but d'utiliser plus efficacement les ressources disponibles. Plus généralement, il s'agit d'un problème d'optimisation de ces ressources en fonction des besoins du calcul à effectuer. Proposer une technique de répartition de charge adaptée à une architecture de type cloud est un domaine de recherche très actif. Toutefois et tel que montré par Liu et Wee (2009), les services de répartition de charge proposés par certains fournisseurs des plateformes cloud ne satisfont pas complètement les attentes des utilisateurs. En effet ces outils sont parfois très génériques et ne tirent pas partie des particularités de certains types d'applications déployées sur ces plate-formes.

Par conséquent il est nécessaire de fournir aux utilisateurs de ces plateformes les outils leur permettant de mettre en place des stratégies de répartition de charge efficaces et adpatées à leurs besoins. Pour pouvoir comparer différents algorithmes de répartition de charge, nous avons développé une nouvelle plateforme appelée CLOUDIZER. L'objectif de cette plateforme est de permettre la distribution, la réplication et la surveillance d'applications web respectant le modèle REST "REpresentational State Transfer" Fielding (2000). Cette plateforme permet de déployer simplement et facilement des applications dans un cloud en assurant la répartition de charge dans le but de tirer profit des avantages de l'élasticité offerte par ce type d'architecture.

Les applications sont vues comme des "boîtes noires" par la plate-forme, c'est-à-dire que le code source et la structure de l'application ne sont pas affectés par le déploiement dans CLOUDIZER. La plupart des applications web utilisent la mise en cache des pages les plus affichées afin de limiter le nombre de requêtes sur leurs bases de données, ce qui permet généralement une forte amélioration des temps de réponse. Cependant notre plateforme n'ayant pas connaissance du fonctionnement de l'application, nous avons conçu une politique de répartition , appelée WACA (Workload And Cache Aware load balancing), qui détecte rapidement les similitudes entre les requêtes, et de façon transparente, transmet ensuite les requêtes à la machine susceptible de contenir les données dans le cache, tout en maintenant une charge équilibrée et maîtrisée en cas de forte demande pour une machine particulière. Le but de cet article est de montrer comment se comporte notre nouvelle politique de répartition de charge par rapport à d'autres algorithmes utilisés pour les applications web. Pour cela nous allons utiliser notre plateforme Cloudizer afin de déployer une application de recherche d'images dont la performance sera évaluée face à différents scénarios d'utilisation.

Cet article est organisé comme suit : la section 2 décrit la plate-forme de répartition CLOUDIZER, la section 3 rappelle le fonctionnement de la structure de donnée appelée "filtres de Bloom" puis décrit l'algorithme WACA et sa version améliorée. La section 4 présente l'état de l'art autour de la répartition de charge des services web. Enfin la section 5 présente les résultat de l'évaluation de notre algorithme et nous concluons à la section 6 et donnons nos perspectives de recherche.

2 Framework CLOUDIZER

2.1 Description du système

Cloudizer est une plateforme logicielle ouverte développée par l'ISEP 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 de nouvelles stratégies de répartition de charge adaptées aux applications cible, en fournissant une interface de programmation suffisamment expressive et claire pour le programmeur. Deuxièmement la plate-forme 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.

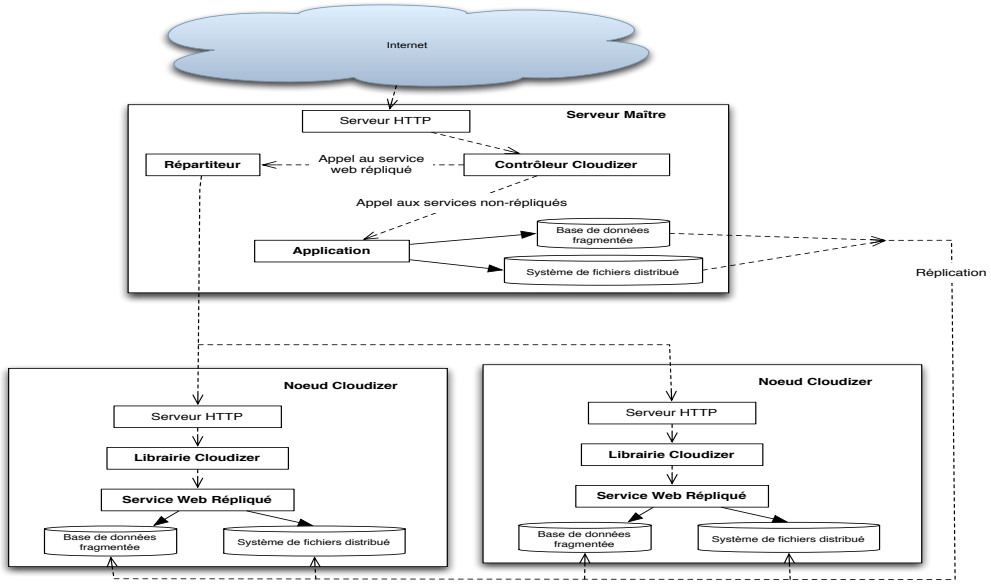


FIG. 1: Cloudizer Framework Architecture

CLOUDIZER utilise l'architecture 3-tiers de la plupart des applications Web pour permettre la répartition de charge. Comme le montre la figure 1, le code du service à distribuer est répliqué sur toutes les machines du système. Les trois composants principaux de cette plateforme sont faiblement couplés et 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 machine du système, appelées "noeuds". 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 *noeuds Cloudizer* selon la stratégie de répartition sélectionnée par l'utilisateur et appliquée par le *répartiteur*. 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 noeuds,

mais les fichiers ou les bases de données sont répartis via l'utilisation d'un système de fichiers distribué type "HDFS" (Shvachko et al. (2010)) ou d'une base de donnée utilisant la fragmentation horizontale. Quand un noeud esclave (*noeud Cloudizer*) s'enregistre dans le système il envoie à une fréquence fixée des "battements de coeur" au *contrôleur* afin que celui-ci puisse connaître les noeuds actifs. Les noeuds enregistrent également la liste des services web actifs sur la machine.

2.2 Les noeuds

Les noeuds de la plateforme Cloudizer sont des machines contenant simplement la librairie Java utilisée par tous les éléments de la plateforme. Cette librairie est exécutable et un simple fichier de configuration permet de spécifier l'adresse du *contrôleur* auprès du quel la machine doit s'enregistrer. Chaque noeud est responsable de la supervision des services de la machine sur laquelle il est déployé et reporte régulièrement un statut au *contrôleur*. Ce statut contient les informations décrivant la machine tel que le nombre de coeurs, sa capacité de stockage, la taille de la mémoire et le taux d'utilisation processeur. Le délai d'envoi des données peut être configuré par l'utilisateur.

Les noeuds dialoguent avec le *contrôleur* via un protocole au format REST. Ils peuvent communiquer entre eux via un protocole d'appel de méthodes à distance et peuvent donc servir de mandataires pour appeler des services distribués.

2.3 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 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, à ce stade du développement chargées lors de l'initialisation du répartiteur, en fonction de la configuration définie par l'utilisateur. A l'avenir, cette configuration pourra se faire à chaud, via une interface web.

2.4 Description et création de services

Une particularité importante de CLOUDIZER est sa capacité à assurer une répartition de charge pour n'importe quelle application de façon transparente et sans toucher à la structure du code de celle-ci. Pour cela nous avons modélisé les applications pouvant être déployées sur CLOUDIZER à l'aide du modèle de description de la figure 3, ce modèle est implanté en XML et permettra de générer un service web REST.

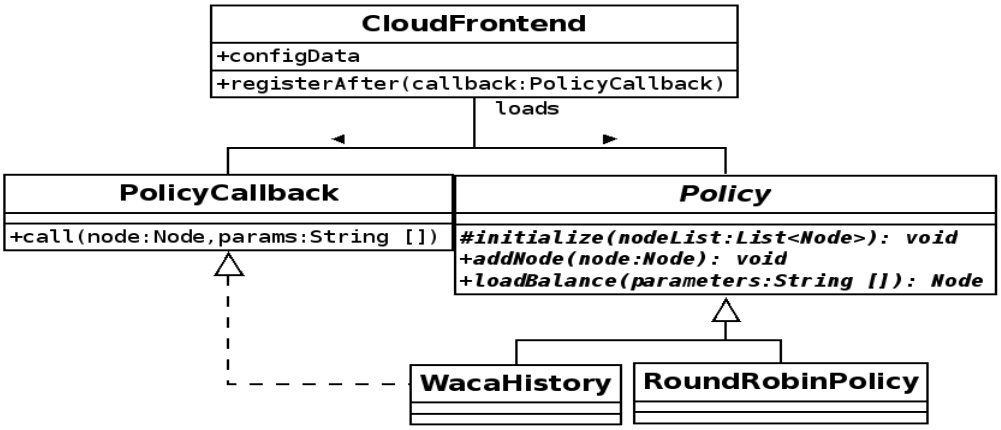


FIG. 2: Modèle des stratégies de répartition de charge

La classe "*Algorithm*" permet de spécifier les fichiers exécutables et de configurations nécessaires à l'exécution du programme. La liste des paramètres (la classe "*Parameter*") décrit les paramètres à fournir à l'exécutable ainsi que leurs positions et préfixes le cas échéant. La classe "*OutputFormat*" permet de décrire le type de donnée en sortie de l'exécution du programme. Par exemple, le format "TextOuput"définit les fichiers textes de type CSV et permet donc de décrire un séparateur de colonnes, ainsi que la liste de ces colonnes. A partir de cette description, un utilitaire de la plate-forme CLOUDIZER permet de générer un service Web REST. Cet utilitaire utilise la librairie Java JETTY (JETTY (2012)) afin de générer un service web Java qui se chargera d'exécuter le binaire fournit avec les paramètres appropriés. Une fois cette "enveloppe" créée pour le programme, il devient possible de la déployer de manière autonome mais aussi sur la plateforme Cloudizer.

3 WACA

Nous avons décrit en introduction que les applications web mettaient fréquemment en cache certaines données nécessaires à leur éxécution. Dans le but de concevoir une stratégie de répartition qui tire partie au maximum de cet effet, nous avons mis trois contraintes en évidence :

- Résumer les caches des noeuds des systèmes,
- Maintenir une charge équilibrée sur les machines,
- Prendre une décision pour l'affectation des requêtes rapidement.

Afin de maintenir un résumé cohérent des requêtes reçues par chaque machine nous allons utiliser les filtres de Bloom avec compteurs, habituellement utilisés dans les systèmes de caches répartis tels que ceux de Fan et al. (2000) et Dominguez-Sal et al. (2012).

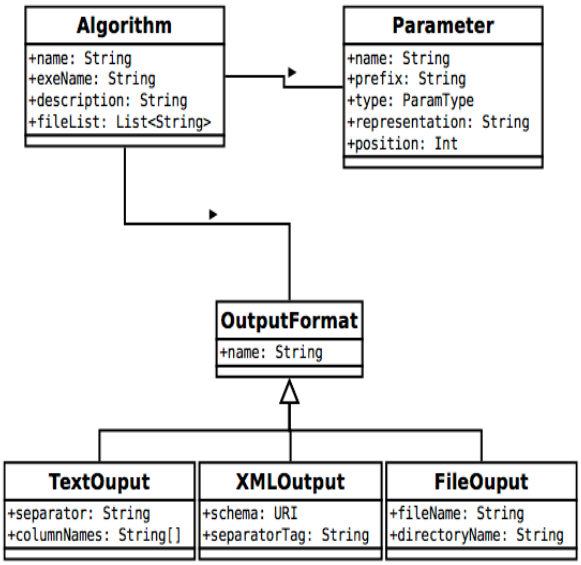


FIG. 3: Modèle de description d'algorithme

3.1 Rappels sur les Filtres de Bloom

Filtres standard Un filtre de Bloom, décrit par B.H Bloom dans Bloom (1970), 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. Le filtre de Bloom consiste a deux composantes principales : un ensemble k de fonctions de hachage et un vecteur de bits d'une taille m fixée. Toutes les fonctions de hachage sont configurées de telle sorte que leurs intervalles correspondent à la taille du vecteur. Par exemple, si le vecteur de bits est de taille 200, alors toutes les fonctions de hachage doivent retourner une adresse entre 1 et 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 calcule les valeurs des fonctions de hachage et on active (on met à 1) les bits du vecteur correspondants.

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 un certaine probabilité. Cette probabilité de *faux positif* est dûe à la différence de taille entre le nombre d'éléments possibles et la taille du filtre car m (la taille du filtre) est inférieure à $|S|$. De plus ce taux évolue avec le nombre d'insertions n dans le filtre, d'après la relation suivante :

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

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 nombre d'éléments insérés en utilisant la

formule :

$$k = ln(2) \times (m/n)$$

Filtres avec compteurs Le filtre de Bloom avec compteur (CBF pour Counting Bloom Filter) est une extension du filtre de Bloom standard qui fournit la possibilité de supprimer des éléments dans le vecteur, il a été introduit par Fan et al. (2000). Le vecteur de bits y est remplacé par un tableau d'entiers, chaque entier ayant le rôle d'un compteur. L'insertion d'un élément est réalisée en incrémenant de 1 les entiers aux positions renvoyées par les k 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.

Le principal avantage des filtres de bloom avec compteur comme technique de résumé est sa facilité d'implémentation et sa complexité $O(1)$ pour la recherche d'élément. Par ailleurs, comme notre objectif est de répartir les applications web de façon transparente sans toucher au code, il suffit de hacher les URLs des appels aux services web pour l'insertion dans le filtre. Ceci ne nécessite aucune connaissance de la structure interne des applications web déployées. Nous avons donc choisi cette technique pour générer les résumés des requêtes reçues par les machines dans notre algorithme.

3.2 Principe Général de l'algorithme

La figure 4 montre une vue abstraite de la stratégie proposée. Cet algorithme est déclenché chaque fois que le *serveur maître* reçoit une requête destinée à l'application cible. Le *répartiteur* maintient une liste des machines disponibles pour cette application et trois structures de données sont attachées à chaque machine : Un filtre de Bloom avec compteur, tel que décrit en section 3.1, 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 noeud 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 noeud le moins chargé de toutes les machines qui sera sélectionné comme cible.

Le noeud le *moins chargé* est celui dont le compteur de requête est le plus bas au moment de l'examen des noeuds par l'algorithme. Le compteur est incrémenté à chaque fois qu'une requête est envoyée au noeud, et décrémenté dès qu'une réponse est reçue de ce noeud. Cependant l'algorithme est conçu de manière à pouvoir fonctionner avec une autre définition de la "charge" des noeuds.

Hétérogénéité des machines Comme la taille du cache est limitée, l'algorithme doit maintenir à jour les filtres des noeuds en ne gardant que les entrées les plus récentes. Cette suppression doit permettre de conserver une faible probabilité de faux positif. 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 1, en divisant

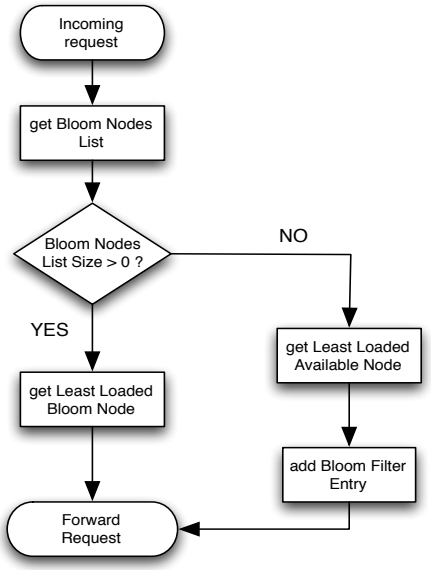


FIG. 4: Description de l'algorithme

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.

$$C(m) \approx \frac{M_m}{S_r} \tag{1}$$

$$Q(m) = C(m) \times k \tag{2}$$

La *Capacité* est la valeur qui permet de déterminer, en fonction du taux de faux positif fourni par l'utilisateur et des formules citées en section 3.1, le nombre k de fonction de hachage optimal à utiliser pour dimensionner le filtre d'une machine donnée. Cependant, le taux de faux positif possible augmente avec le nombre d'éléments présents dans filtre. Il est donc nécessaire de maintenir le taux de faux positifs à la valeu configurée par l'utilisateur. Pour cela, une File d'attente, dont la taille est égale à la *Capacité* multipliée par le nombre k de fonctions de hachage utilisées dans le filtre (équation 2), est associée à chaque filtre. 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 éqauivaut à une suppression. Ce mécanisme permet de maintenir la probabilité de faux positif du Filtre de Bloom avec Compteur constante.

Choix de la fonction de hachage La fonction de hachage sélectionnée est la fonction Message Digest 5 de Rivest (1991). Cette fonction est utilisée dans des travaux similaires comme ceux de Fan et al. (2000) et fournit un taux de collisions très bas de sorte que les valeurs obtenues sont distribuées de manière équitable sur l'ensemble des valeurs possibles, pour un temps

d'éxécution acceptable. Il existe cependant dans la littérature des fonctions de hachage plus appropriées à ce type d'utilisation, telles que la fonction MurmurHash Appleby (2008), qui est conçue pour s'exécuter rapidement, au détriment des propriétés cryptographiques fournies par les fonctions comme MD5.

3.3 Algorithme sans historique

Le pseudo-code 1 présente une première version du déroulement de l'algorithme. En l'état, cet algorithme a une compléxité de $O(n)$ où n est le nombre de noeuds présents dans le système. Cet algorithme est déclenché à chaque nouvelle requête reçue sur le système.

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
```

L'algorithme itère sur la liste des noeuds disponibles pour le service demandé *nodesList*. Pour chaque noeud n, l'appel à la fonction *queryBloomFilter* vérifie si la machine n a déjà reçu une requête similaire (ligne 5). Cette fonction retourne VRAI si la requête est présente dans le filtre du noeud, et FAUX dans le cas contraire. A la ligne 9, l'algorithme vérifie si le noeud sélectionné est bien le moins chargé parmis ceux qui ont la requête dans leur filtre.

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

A la fin de la boucle, l'algorithme retourne le noeud le moins chargé qui a la donnée dans son filtre. Si aucun noeud n'a la donnée dans son filtre, l'algorithme retourne le noeud le moins chargé parmis toutes les machines et ajoute la requête à son filtre (lignes 22,23).

3.4 Algorithme avec historique

Nous avons détecté un effet de bord important concernant les filtres de Bloom, permettant à une machine qui serait plus rapide que les autres d'attirer un plus grand nombre de

requêtes différentes et donc d'enregistrer dans son filtre la majorité des requêtes envoyées. La conséquence est que malgré une puissance de calcul plus élevée, cette machine se retrouve rapidement surchargée car elle va attirer à elle la majorité des requêtes. Pour réduire ce risque, nous avons introduit dans notre algorithme de nouveaux compteurs, associés à chaque noeud : les compteurs d'historique. Le but de ces compteurs est de permettre une répartition plus juste des requêtes entre les noeuds afin d'éviter la surcharge d'une machine. Un compteur d'historique est associé à chaque noeud du système. Le compteur est incrémenté lorsqu'une requête est transférée à un noeud qui n'a pas reçu la même requête auparavant. Nous utilisons ensuite ce compteur dans le cas où aucun noeud n'a reçu de requête similaire.

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 2. La principale modification 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.

3.5 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 compléxité est à relativiser étant donné la rapidité des opérations effectuées pour chaque noeud (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 significativement important, alors le recours à une politique de répartition type "cache-aware" n'est peut être pas approprié. Les

temps de réponses typiques d'un service web doivent se situer autour de 500 mili-secondes au niveau des serveurs pour rester acceptables d'après Nielsen (1993). Nous montrerons en partie 5.6 que le temps de sélection d'un noeud par l'algorithme est largement en dessous de cette limite.

4 Etat de l'art

De nombreuses politiques de répartition de charge ont été développées pour les services Web. Ces stratégies sont divisées en deux catégories principales : celles qui affectent les requêtes sans prendre en considération le contenu sémantique de ces messages (appelées en anglais *Content-blind Policies*), et les politiques qui se basent sur le contenu des requêtes pour l'affectation (appelées en anglais *Content-aware Policies*).

Des exemples de la première catégorie sont décris dans Bonomi. (1990) et Lu et al. (2011). Dans Bonomi. (1990), les auteurs présentent une approche pouvant être appliquée lorsque l'information de charge des sites est connue. La politique d'allocation au site le moins chargé (celui qui exécute le plus petit nombre de requêtes) est appelée "Join the Shortest Queue" (JSQ).Lu et al. (2011) proposent une amélioration du JSQ qui permet d'avoir de multiples répartiteurs de requêtes vers les machines. Cette approche appelée "Join the Idle Queue" offre de très bonnes performances en terme de communication entre les répartiteurs et les machines, mais aussi en terme de temps d'exécution des requêtes.

Dans Gou et al. (2010) les filtres de Bloom sont utilisés pour assurer la répartition de charge dans des architecture de réseaux pour la détection d'intrusion. Afin de maintenir un équilibrage de charge entre les processeurs du réseau du système de détection d'intrusion (NIDS pour Network Intrusion Detection System), un algorithme de répartition de charge aléatoire pondéré a été développé. Dans cette approche, chaque processeur est sélectionné de façon aléatoire avec une probabilité proportionnelle à son poids. Le poids pour chaque processeur est défini périodiquement en fonction de l'historique des flux passés. Afin de calculer progressivement un nouvel ensemble de poids, l'algorithme construit des filtres de Bloom qui représentent les flux dans le réseau.

Les techniques basées sur le contenu ou "Content-aware" essaient généralement d'optimiser la localité des données en transférant les requêtes vers les machines où les données se trouvent. Dans Pail et al. (1998), les auteurs proposent un répartiteur qui envoie les requêtes concernant la même donnée au même noeud. Le but poursuivi par ce type de répartition est l'optimisation des accès disques car cela permet à terme de placer les données concernées dans le cache mémoire de la machine. En observant ceci, Fan et al. (2000) ont étudié une technique de répartition basée sur le cache, où un calcul de probabilité est effectué pour envoyer la requête vers le noeud qui a le plus de chance de contenir la donnée dans son cache. Ce type de répartition est utilisé dans des logiciels de mémoire partagée. Ces stratégies sont également appelées "Cache-aware" dans la littérature.

On peut retrouver dans le domaine des bases de données des stratégies prenant en compte le contenu du cache. Par exemple, Rohm et al. (2001) estiment le contenu des caches de données des noeuds par la création de signatures des requêtes. O'Gorman et al. (2002) ont travaillé sur la synchronisation des requêtes via un middleware dédié pour former des "équipes" de requêtes qui peuvent bénéficier des accès aux contenus des caches. Dominguez *et al.* Dominguez-Sal et al. (2012) ont démontré que la mise en cache coopérative, qui est un pool de cache commun

pour tous les noeuds du cluster, utilisant un protocole d'échange de résumés de caches peut être efficacement utilisé pour la répartition de charge en réduisant le trafic dans le réseau pour retrouver les données. Cependant cette approche n'est évaluée que dans un contexte applicatif précis, permettant une connaissance à priori de la charge générée par les requêtes.

Réduire le temps d'exécution d'une requête est le principal objectif de la quasi-totalité des algorithmes de répartition de charge. La principale limitation des approches qui prennent en compte le cache est leur complexité : le temps nécessaire au répartiteur pour choisir le noeud adéquat est relativement élevé. De plus, la plupart de ces approches ne prennent pas en considération le risque de surcharger des noeuds.

5 Etude expérimentale

Les expérimentations décrites dans cette section ont pour but de montrer le gain obtenu par l'utilisation des algorithmes WACA 1 et 2, par rapport à des algorithmes de répartition existants, et l'un par rapport à l'autre.

5.1 Description

Application Test Dans le but de comparer cette nouvelle stratégie de répartition avec d'autres algorithmes connus, nous avons implémenté un service web de recherche d'images. Cette application a été développée à l'aide du framework GRAILS en utilisant le langage Groovy, et une base de donnée HSQL. Cette application permet à ses utilisateurs d'uploader des images avec des paramètres de descriptions et permet de rechercher les images associées à des mots clés extraits des descriptions fournies par les utilisateurs. Ce type d'application multimédia a été choisie en raison des besoins importants en matière de stockage et de traitement nécessaires pour ce type de services.

Pour chaque image, les paramètres sont, entre autres, le "nom" de l'image, la "description" de l'image, le "lieu" où l'image a été prise, et la "catégorie" dans laquelle l'image peut être placée ("historique", "famille", ...). L'utilisateur a la possibilité d'ajouter de nouvelles catégories et de les éditer lors de l'envoi des images sur le site. Les paramètres "nom" et "catégorie" sont obligatoires pour enregistrer une image, ce qui permet d'avoir l'assurance de pouvoir indexer toute image, au moins sur la base de ces deux paramètres.

Cette application a été déployée à l'aide du framework CLOUDIZER décrit en section 2 afin de permettre la comparaison entre notre nouvelle stratégie et deux stratégies de répartition éxistantes.

L'ensemble des donnés est répliqué sur toutes les machines du système, ce qui permet à toutes les machines de pouvoir répondre à n'importe quelle requête des clients.

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 noeuds esclaves afin de tester le comportement de notre stratègie de répartition vis à vis de l'hétérogéneité des machines. Les noeuds sont reliés sur un réseau local ethernet à 100Mb/s.

Paramètrage du répartiteur 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 positif égale à 0.155 d'après les formules de la section 3.1. Cette capacité à été choisie car elle correspond à la plus petite capacité des noeuds du système. Les fonctions de hachage choisies sont les fonctions SHA1 et MD5 car ces fonctions procurent une bonne répartition des valeurs avec des risques de collisions réduits.

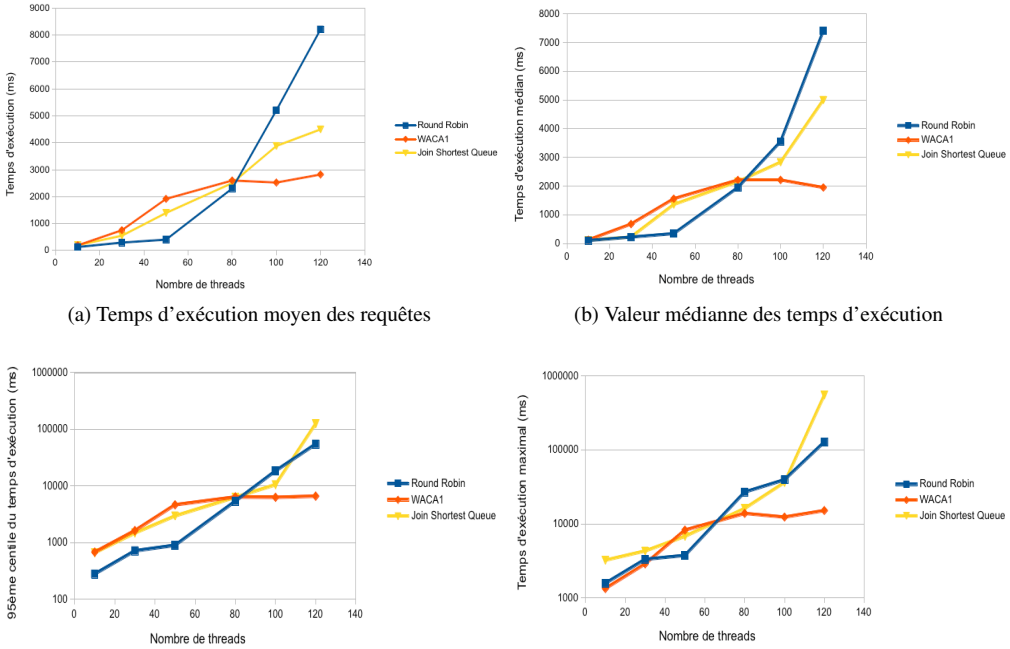
5.2 Programme de test

Le programme de test utilisé pour évaluer la performance des politiques de répartition lance une série de 10 requêtes de recherche 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, et à une stratégie de répartition de type Join Shortest Queue sur le modèle de Bonomi. (1990). 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.

5.3 Premiers résultats et discussion

A partir des données enregistrées lors des tests, nous présentons en figure 5 différentes métriques sur les temps d'exécution des requêtes de recherche sur la base d'images décrite en section 5.1. Etant donné que les applications déployées dans le framework CLOUDIZER sont vues commes 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, notre stratégie de répartition WACA fournit de meilleures performances que les deux autres stratégies examinées.

La figure 5a montre le temps d'exécution moyen par requête, mesuré en milli-secondes. 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 5b, 5c et 5d montrant respectivement les



(a) Temps d'exécution moyen des requêtes

(b) Valeur médianne des temps d'exécution

(c) 95ème centile des temps d'exécution

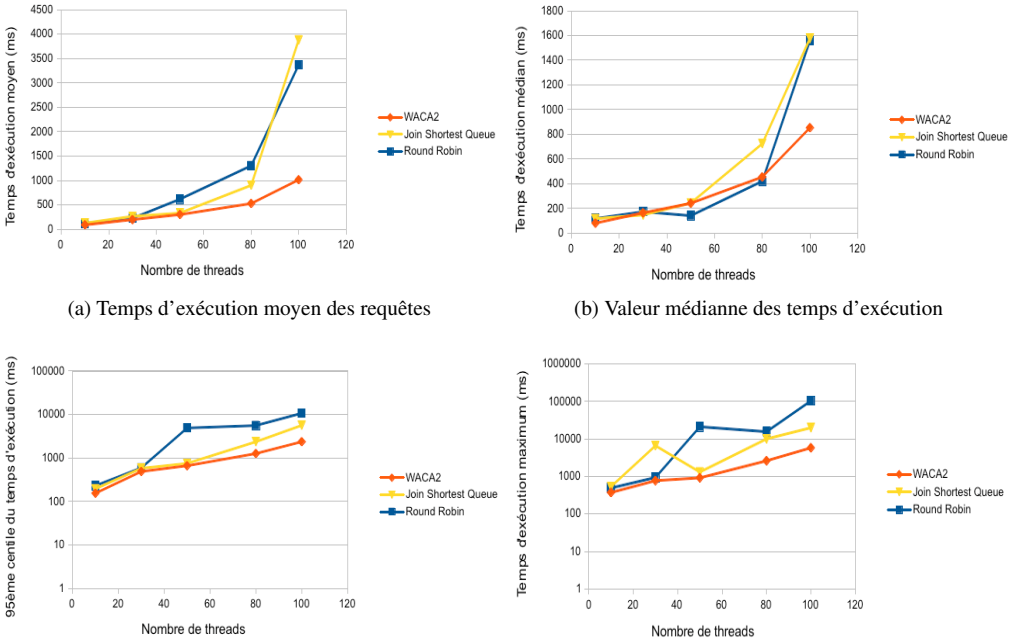
(d) Temps d'exécution maximal

FIG. 5: Mesures du temps d'exécution des requêtes, WACA version 1

valeurs médiannes, les 95ème centile et la valeur maximale du temps d'exécution des requêtes sur 5 essais. les tendances sont similaires sur tous ces graphiques.

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 trois. 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 meilleur stabilité au système, ainsi qu'un temps d'exécution des requêtes plus prévisible comme on peut le déduire des grahiques de la médianne(figure 5b) et du 95ème centile(figure 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.



(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

FIG. 6: Mesures du temps d'exécution des requêtes, WACA version 2

5.4 Experimentation de WACA version 2

De nouveau, nous avons réalisé les mêmes tests que ceux menés avec la première version de WACA décrits en section 5.2, l'application est donc la même, ainsi que la répartition des différentes machines et les paramètres de l'algorithme. Une amélioration générale des performances est visible sur ces graphiques. Ceci est dû à l'utilisation d'une nouvelle version de la plateforme CLOUDIZER, qui à été réécrite en JAVA. Les mêmes politiques de répartition ont été utilisées pour la comparaison.

5.5 Comparaison et Discussion

Les courbes présentées en figure 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'ecart 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'erreur sur une des machines esclaves.

Malgré la différence entre les deux dispositifs expérimentaux, il est possible de comparer nos deux stratégies en étudiant les gains relatifs obtenu dans chaque expérience par rapport au politiques Round Robin et JSQ. La figure 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. Nous constatons donc que WACA version 2 permet d'obtenir des temps d'exécutions inférieurs à ceux obtenus avec les politiquesRound Robin et JSQ quel que soit le niveau de concurrence de l'application, ce qui n'est pas le cas avec WACA 1.

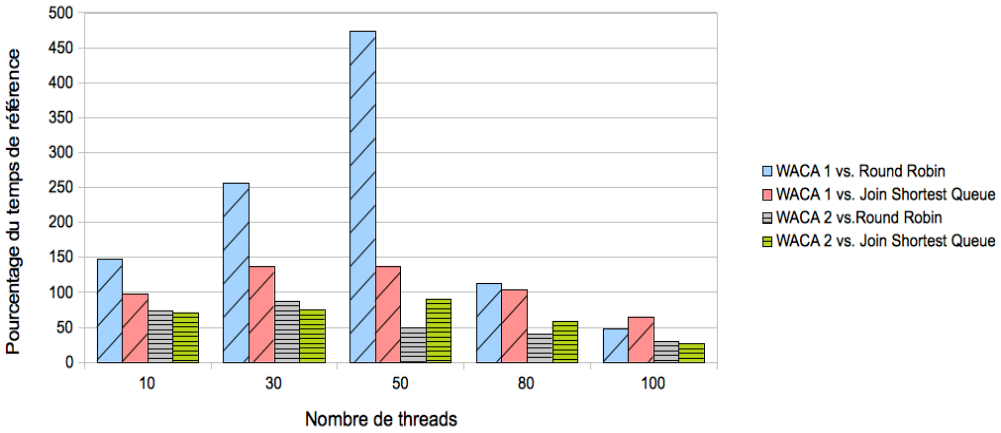


FIG. 7: Comparaisons de WACA 1 et 2

Il apparait que la principale limite de l'algorithme actuel est son temps d'exécution qui est linéaire et proportionnel au nombres de machines présentes dans le système. Au vu de la figure 6 l'algorithme semble donc adapté à un système où le gain obtenu de l'utilisation des caches se chiffre en plusieurs centaines de milisecondes.

5.6 Simulation pour le passage à l'échelle

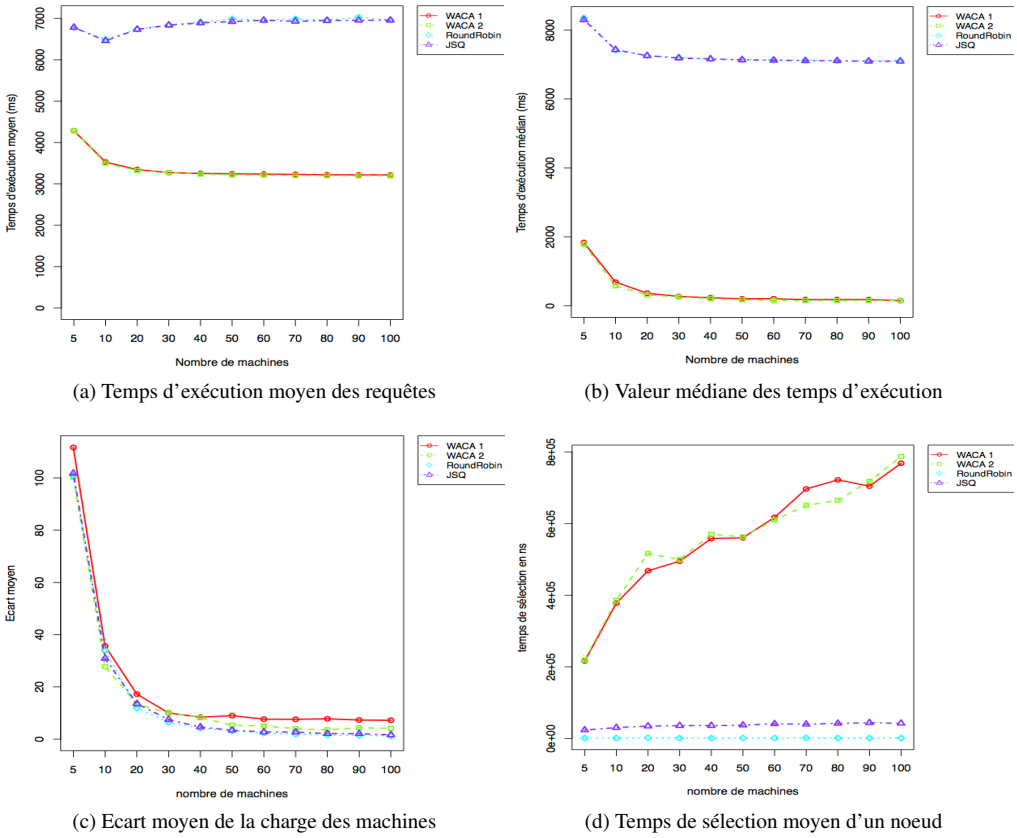
Pour tester les politiques de répartition de charge mises au point pour notre plateforme Cloudizer, nous avons développé un simulateur de répartition de charge. Cet outil nous permet de tester le comportement de nos politiques de répartition avant de les déployer en environnement de test "réel". Ce simulateur utilise la même interface (décrite en section 2) que la plateforme Cloudizer pour les politiques de répartition de charge. Ceci permet un passage simplifié de l'environnement de simulation à l'environnement de test.

Le but de la simulation effectuée est de montrer que la politique de répartition de charge utilisée tire avantage des ressources supplémentaires disponibles lorsqu'on augmente le nombre de machines dans le système.

Les diagrammes de la figure 8 montrent les résultats de simulation des politiques de répartition sur un modèle de l'application de recherche d'images utilisée en section 5.1. Ce modèle est caractérisé par les temps d'éxécution observés pour les requêtes en cache ou hors-cache. Il est apparu qu'en moyenne, le temps d'éxécution d'une requête hors cache est d'environ 7

secondes, et une requête avec les données en cache s'éxécute en moyenne en 50 mili-secondes. Le système peut répondre à un ensemble de 150 requêtes différentes avec des temps de réponse similaires.

Le simulateur utilisé génère un processus d'arrivée de requêtes suivant une loi de Poisson de paramètre $\lambda = 1.0$, toutes les 30 milisecondes. La distribution des différentes requêtes suit une loi de Zipf de paramètres $s = 1.4$ et $N \in \{1, 2, .., 150\}$, où N représente le numéro de la requête. Cette loi a été choisie car elle est représentative de la distribution des requêtes dans les systèmes de recherche par contenu utilisés dans des travaux similaires tels que ceux de Dominguez-Sal et al. (2012). Les résultats présentés en figure 8 montrent sont tirés de simulation fondées sur ce modèle d'application.



(a) Temps d'exécution moyen des requêtes

(b) Valeur médiane des temps d'exécution

(c) Ecart moyen de la charge des machines

(d) Temps de sélection moyen d'un noeud

FIG. 8: Résultats de simulation

De même que précedemment, les simulations portent sur les algorithmes Round Robin, Join Shortest Queue (JSQ), Waca 1 et Waca 2. Les graphiques 8a et 8b présentent respectivement la moyenne et la médiane des temps d'exécution des requêtes sur 5 simulations. Il montrent clairement que les algorithmes WACA 1 et 2 peuvent prendre en compte la présence de nouvelles ressources pour améliorer le temps de réponse moyen, et ce jusqu'à un nombre important de machines. Le graphique 8c présente l'écart moyen observé du nombre de requêtes

par noeuds. On constate ici l'amélioration apportée par l'historique de WACA 2 qui permet de réduire l'écart de charge entre les machines par rapport à la version 1. Enfin, le graphique 8d montre des mesures du temps de traitement moyen d'une requête par les algorithmes de répartition. Il apparait clairement que de ce point de vue les politiques WACA ne sont pas aussi performantes que le Round Robin ou le JSQ qui s'éxécutent en temps quasi-constant. Cependant, les mesures effectuées sont de l'ordre de la centaine de micro-secondes, ce qui reste négligeable comparé aux temps de réponses d'un serveur web, qui se situent autour de la centaine de mili-secondes comme montré dans Nielsen (1993).

6 Conclusion et perspectives

Afin de bénéficier de manière transparente de la mise en cache des données par les systèmes d'exploitations modernes, nous avons conçu et implanté une nouvelle stratégie de répartition de charge qui mélange avec succès les stratégies fondées sur les caches et les stratégies fondées sur la charge des machines. A l'aide de notre propre plateforme de répartition de charge il nous a été possible de comparer cette stratégie à des politiques de répartition existantes. Cette démarche expérimentale nous a permis d'établir les pistes d'amélioration de l'algorithme et de la plateforme qui se diviseront en deux axes.

Un premier axe concernant l'algorithme : il s'agira d'améliorer l'efficacité et la stabilité à large échelle. Un aspect critique de cette performance est le temps de recherche dans les filtres pour sélectionner la machine cible : ce temps est actuellement linéaire avec le nombre de machines. L'utilisation de structures de données plus efficaces pour stocker les noeuds et les filtres permettra de réduire drastiquement ce temps de recherche. Il est ensuite possible d'améliorer la stabilité en distribuant l'algorithme en utilisant des méthodes de propagation de rumeurs pour communiquer les filtres de Bloom depuis les noeuds vers le(s) répartiteur(s). La troisième piste d'amélioration de l'algorithme est la prise en compte des données matérielles fournies par les machines du système pour adapter la taille des filtres de manière dynamique et mieux évaluer la charge des machines.

Le deuxième axe concerne la plateforme Cloudizer qui est en développement actif à l'ISEP. Premièrement le *répartiteur* doit être adapté pour supporter un nombre plus important de requêtes simultanées, car il est actuellement un point central et donc critique dans cette architecture. Ce service constitue donc un goulot d'étranglement, car chaque requête nécessite un appel au service qui maintient la liste des noeuds à jour. Ces améliorations ouvriront la voie à des expérimentations à plus large échelle sur des plateformes de type "infrastructures à la demande" telle que la plateforme EC2 d'Amazon (2012).

Références

Amazon (2012). Amazon elastic compute cloud (amazon ec2). http ://aws.amazon.com/ec2/.

Appleby, A. (2008). Murmurhash, 2008.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*.

Bonomi., F. (1990). On job assignment for a parallel system of processor sharing queues. *IEEE Transactions on Computers*.

Dominguez-Sal, D., J. Aguilar-Saborit, M. Surdeanu, et J. L. Larriba-Pey (2012). Using Evolutive Summary Counters for Efficient Cooperative Caching in Search Engines. *IEEE Transactions on Parallel and Distributed Systems 23*(4), 776–784.

Fan, L., P. Cao, J. Almeida, et A. Broder (2000). Summary cache : a scalable wide-area web cache sharing protocol. *Networking, IEEE/ACM Transactions on 8*(3), 281 –293.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Phd thesis, University of California.

Gou, C., R. Zhao, et J. Diao (2010). A load-balancing scheme based on bloom filters. In *2010 second international Conference on Future Networks*.

JETTY (2012). http ://jetty.codehaus.org/jetty/.

Liu, H. et S. Wee (2009). Web server farm in the cloud : Performace evaluation and dynamic architecture. In Springer (Ed.), *Lecture Notes In Computer Science*, Volume 5931, pp. 369–380.

Lu, Y., Q. Xie, G. Kliot, A. Geller, J. Larus, et A. Greenberg (2011). Join-idle-queue : A novel load balancing algorithm for dynamically scalable web services. *The 29th International Symposium on Computer Performance, Modeling, Measurements and Evaluation*.

Nielsen, J. (1993). *Usability Engineering*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc.

O'Gorman, K., D. Agrawal, et A. El Abbadi (2002). Multiple query optimization by cache-aware middleware using query teamwork. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pp. 274.

Pail, V. S., M. Aront, G. Bangat, M. Svendsent, P. Druschelt, W. Zwaenepoelt, et E. Nahumq (1998). Locality-Aware Request Distribution in Cluster-based Network Servers. *Distribution*, 205–216.

Rivest, R. L. (1991). The MD5 Message-Digest algorithm (RFC 1321).

Rohm, U., K. Bohm, et H.-J. Schek (2001). Cache-aware query routing in a cluster of databases. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pp. 641 –650.

Shvachko, K., H. Kuang, S. Radia, et R. Chansler (2010). The hadoop distributed file system. *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on 0*, 1–10.

Summary

Numerous applications are relying on web services deployed on cloud computing architectures. These services are handling massive data and large quantities of files to answer an increasing number of users requests. This data can be of different types, ranging from multimedia files to measurements from sensors. In addition to data volume, web applications must also deal with customer's satisfaction, which implies Quality Of Service agreements. Based on these facts, task allocation on clustered machines becomes a requirement for many applications. Application efficiency will then largely depend on the chosen allocation policy.

WACA: Politique de répartition de charge pour services Web

This article presents a new load balancing policy, named WACA (Workload And Cache Aware load balancing) which takes the machines cache content into account in addition to their load. This policy allows to reduce user requests execution time by using cache summaries constructed using Bloom filters. WACA is a centralized load balancing algorithm which allocates requests to the machine which is the most likely to contain targetted data, without overloading any machine in the system. This policy was developped and tested by using a web application balancing framework named CLOUDIZER, and the strategy was compared to other traditional approaches.