MoKa : Modélisation et planification de capacité pour les systèmes multi-étagés

Jean Arnaud
INRIA
Grenoble, France
Jean.Arnaud@inria.fr

Sara Bouchenak
University of Grenoble I
Grenoble, France
Sara.Bouchenak@inria.fr

ABSTRACT

Bien que les hébergeurs d'applications multi-étagées basées sur grappe de machines permettent de passer à l'échelle les applications Web, leur configuration ad-hoc pose des problèmes en terme de performance et de coût de fonctionnement pour les applications. Cet article présente la conception et l'implémentation de MoKa, un canevas logiciel pour la modélisation, la gestion de ressources et la configuration optimale des systèmes multi-étagés. Les contributions de cet article sont les suivantes. Premièrement nous identifions deux niveaux de configuration pour les applications hébergées sur grappe de machines, une *configuration locale* qui s'applique au niveau de chaque serveur et une *configuration architecturale* qui concerne la grappe de machines hébergeant les serveurs. La combinaison de ces deux niveaux de configuration améliore les performances globales et le coût de fonctionnement des application multi-étagées hébergées sur des grappes de machines. Deuxièmement nous présentons une *fonction d'utilité* pour caractériser l'impact des configurations locales et architecturales sur les performances et le coût des systèmes multi-étagés. Troisièmement nous développons un algorithme de gestion de ressources basé sur une fonction d'utilité, pour calculer efficacement la configuration locale et architecturale de l'application multi-étages pour fournir des garanties sur la performance tout en minimisant le coût. Les expérimentations menées sur une application multi-étages de commerce en ligne mettent en évidence l'efficacité de MoKa. De plus, les expériences montrent que la combinaison des configurations locales et architecturales fournissent une précision de 100% dans la recherche de la valeur maximale de la fonction d'utilité, alors que la précision est limitée entre 20% et 90% avec un seul niveau de configuration.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Miscellaneous—*client/server, distributed applications, distributed databases*; C.4 [**Performance of Systems**]: Modeling techniques; I.6.4 [**Model Validation and Analysis**]

General Terms

Algorithms, Design, Performance

Keywords

Multi-tier systems, Data centers, Modeling, Capacity planning, QoS, Optimization.

NOTERE 2009 July 1-3, 2009, Montreal, Canada

1. INTRODUCTION

1.1 Contexte et défis

Les centres d'hébergement, ou *data centers*, hébergent une vaste gamme de services Internet, depuis les serveurs Web vers les serveurs de messagerie, en passant par les serveurs d'entreprise ou les systèmes de gestion de base de données [2, 13, 23]. Ces services sont habituellement basés sur l'architecture client-serveur, dans laquelle un serveur fournit des services en ligne à des clients y accédant en parallèle, comme la consultation de pages Web, l'envoi de courrier électronique ou encore l'achat sur des sites de vente en ligne. Pour faire face à la charge croissante sur ce type d'applications, les serveurs sont organisés en architectures multi-étagées. La figure 1 représente une application Web à trois étages qui commence avec les requêtes des clients, passant du serveur HTTP frontal gérant les pages statiques, vers un serveur d'entreprise qui exécute la logique métier de l'application et génère les pages de manière dynamique, puis vers la base de données qui stocke les données à conserver.

Cependant, la complexité des applications multi-étagés et leur faible débit pour servir des pages Web dynamiques – souvent un ou deux ordres de magnitude plus faible que les pages statiques – engendre une surcharge significative sur les centres d'hébergement [11]. Pour répondre à ces fortes charges, et permettre aux services de passer à l'échelle, une approche classique consiste à dupliquer et à répartir les serveurs dans des grappes de machines [21, 4, 19].

Pour les hébergeurs de services multi-étagés, le défi réside dans les objectifs contradictoires de haute performance, et de coût et consommation de ressources minimum. Dans le cas extrême, de hautes performances peuvent être obtenues en utilisant toutes les ressources disponibles dans le centre d'hébergement pour traiter les requêtes des clients. Inversement, il est possible de construire un centre d'hébergement à très faible coût en allouant seulement quelques ressources, ce qui induira de mauvaises performances et des périodes d'indisponibilité du service [14]. Entre ces deux extrêmes, il

existe une configuration pour laquelle le service multi-étagé atteint un niveau de performance acceptable avec un coût minimisé. Cet article traite en particulier du problème de la détermination de cette configuration optimale.

1.2 Contributions scientifiques

Les contributions de cet article sont les suivantes :

- Nous identifions et combinons deux niveaux de configuration pour les services multi-étagés hébergés sur grappe de machines : une *configuration architecturale* basée sur l'allocation de ressources dans les grappes de machines chez un hébergeur de service, et une *configuration locale* appliquée au niveau de chaque serveur. Dans l'état actuel de nos connaissances, les approches existantes sont limitées à un seul niveau de configuration [27, 17, 9, 25]. Dans cet article, nous montrons que contrairement aux approches existantes, une configuration sur les deux niveaux permet de garantir les objectifs de performance tout en minimisant le coût de fonctionnement du service multi-étages.

- Nous proposons une *modélisation* des systèmes multi-étagés qui prédit les performances et le coût de tels systèmes. Notre modèle est basé sur un réseau de files d'attente qui étend l'algorithme MVA (Mean-Value Analysis) [22].

- Nous définissons une *fonction d'utilité* pour caractériser l'impact des configurations locales et architecturales sur les performances et le coût des systèmes multi-étages.

- Nous développons un *algorithme de gestion de ressources* utilisant cette fonction d'utilité pour calculer efficacement la configuration locale et architecturale des systèmes multi-étagés.

En plus de ces contributions, cet article présente la conception et la mise en oeuvre de MoKa, un canevas logiciel pour la modélisation et la gestion de ressources des systèmes multi-étagés, ainsi que le calcul de leur configuration optimale.

Enfin cet article décrit l'évaluation de l'approche proposée en utilisant un site d'enchère en ligne basé sur une architecture multi-étages et reproduisant le comportement de *www.eBay.com*. Les résultats de nos expériences montrent que l'approche proposée ici améliore significativement les performances et le coût des systèmes. Comparée aux méthodes standard pour la gestion de ressources dans les systèmes multi-étagés, où 15% des ressources peuvent être gaspillées et 94% des requêtes client peuvent être rejetées pour cause de surcharge du service, une méthode de planification de capacité qui combine configurations locale et architecturale permet de garantir la performance de l'application tout en minimisant son coût.

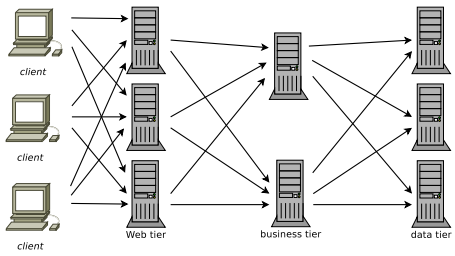

Figure 1: Application multi-étagée

1.3 Plan de l'article

La suite de cet article est organisée comme suit : La section 2 présente les systèmes multi-étagés et le cadre applicatif. La section 3 propose un modèle analytique pour les systèmes multi-étagés basés sur grappes de machines. La section 4 décrit notre solution de planification de capacité. La section 5 présente la réalisation du prototype MoKa. Les résultats d'évaluation sont discutés en section 6. La section 7 présente l'état de l'art. Nos conclusions et perspectives sont présentées en section 8.

2. CADRE APPLICATIF

2.1 Hypothèses et définitions

Les applications multi-étagées suivent le modèle client-serveur, dans lequel les clients se connectent au système qui est composé d'une série d'étages $T_1, T_2, ..., T_M$. Chaque étage construit une partie de la réponse à la requête du client. Par exemple, le système multi-étagé de la figure 1 est composé d'un étage T_1 chargé de construire la page Web de la réponse, d'un étage T_2 responsable de la logique métier de l'application, et d'un étage T_3 chargé du stockage des données persistantes de l'application.

La requête d'un client sur un système multi-étagé accède à l'étage T_1 et peut ensuite circuler à travers les étages successifs $T_2, T_3, \ldots, T_M$. Plus précisément, quand une requête est traitées par un étage T_i, soit une réponse est retournée à l'étage T_{i-1} (ou au client si $i = 1$), soit une autre requête est envoyée à l'étage suivant T_{i+1} (si $i < M$).

Sous sa forme basique, chaque étage est constitué d'un seul serveur. Or de nombreux clients peuvent accéder simultanément au même serveur. Pour éviter l'écroulement d'un serveur lorsque le nombre de clients y accédant de manière concurrente augmente, une approche classique est le contrôle d'admission. Cela consiste à fixer une limite au nombre maximum de clients autorisés à accéder de manière concurrente au serveur. Cette limite est un paramètre de configuration des serveurs également appelée degré de multiprogrammation (*multi-programming level* ou *MPL*). Au-dessus de cette limite, les tentatives de connexion des clients sont rejetées. Donc, la requête d'un client traitée par un système multi-étagé soit termine avec succès en retournant une réponse au client, soit est rejetée à cause de la limite de concurrence sur le serveur.

De plus, le nombre de clients N (la quantité de charge) qui essaient d'accéder de manière concurrente au système multi-étagé peut varier au cours du temps. Ceci correspond aux différents comportements des client à différentes périodes, par exemple un service de messagerie électronique subira une plus forte charge le matin que pendant le reste de la journée.

Un serveur dans un système d'hébergement multi-étagé est hébergé par une ressource (i.e. une machine, un noeud de la grappe), et une ressource est exclusivement utilisée par un serveur. Cependant, pour des raisons de passage à l'échelle, un étage est habituellement un groupe de machines constitué de plusieurs serveurs répliqués, en utilisant

des techniques de partitionement et de répartition de charge entre les machines. Dans la suite, nous considérerons que la répartition de charge est équitable entre les machines d'un étage.

Propriété P1 : unicité du goulot d'étranglement. En cas d'approvisionement inadéquat, un goulot d'étranglement peut apparaitre sur un des étages de l'application multi-étagée. Mais il ne peut y avoir simultanément de goulot d'étranglement sur plusieurs étage de l'application [6].

Propriété P2 : l'ajout de ressources ne dégrade jamais les performances. Evidemment, plus le nombre de ressources allouées au système est important, meilleures sont les performances. Cette augmentation de performances atteint une limite après un certain nombre de ressources ajoutées, mais les performances ne sont pas dégradées par l'ajout de ressources supplémentaires. Dans la suite, entre quelques dizaines et quelques centaines de machines forment une grappe de machines, et les serveurs constituant cette grappe sont homogènes, c.a.d. qu'ils ont tous la même architecture matérielle comme c'est généralement le cas dans des grappes de machines de cette taille [5].

2.2 Performances et coût

Les métriques principales pour quantifier la qualité de service (ou *QoS* pour *Quality of Service*) d'une application multi-étagée sont :

Latence des requêtes client, le temps nécessaire à l'application multi-étagée pour traiter la requête d'un client. Ceci correspond à la période de temps entre le moment où le client envoie une requête au système et le moment où il reçoit une réponse à cette requête. La latence moyenne des requêtes client (ou plus simplement *latence*) est notée ℓ. Une faible latence est souhaitée car elle permet d'avoir un système réactif du point de vue des clients.

Taux d'échec des requêtes client, définit comme le ratio entre le nombre de requêtes rejetées et le nombre total de requêtes reçues par un système multi-étagé. Le taux d'échec est noté α. Un faible taux d'échec est souhaitable car il reflète une haute disponibilité du système.

Le *contrat de qualité de service* (ou *SLA* pour *Service Level Agreement*) est un contrat négocié entre le fournisseur de l'application et l'hébergeur du service [15]. Le SLA spécifie les objectifs de qualité de service devant être garantis par l'application, comme la latence maximum ℓ_{max} ou le taux d'échec maximum α_{max} des applications multi-étagées. Il faut noter que la combinaison des objectifs de latence et de taux d'échec est importante. En effet, garantir seulement la latence maximum d'une application peut mener à une situation où l'objectif de latence maximum n'est respecté que pour 10% des requêtes, alors que 90% des requêtes sont rejetées par manque de garantie sur le taux d'échec. Dans cet article, nous montrons comment combiner ces deux objectifs de performance.

Propriété P3 : réalisme du SLA. Les contraintes de latence et de taux d'échec maximum spécifiés dans le SLA doivent être réalistes. Ces contraintes doivent être atteignables en ajoutant une certaine quantité de ressources. Par exemple, on ne peut spécifier de contrainte sur la latence qui soit inférieure au temps de service de la requête.

En plus des objectifs de performance qu'un système multi-étagé doit garantir, son coût est un autre aspect devant être pris en compte lors de l'allocation de ressources par l'hébergeur.

Le ***Coût*** d'un système multi-étagé reflète les coûts économiques et énergétiques liés au fonctionnement d'un tel système. Le coût est directement proportionnel au nombre de machines allouées au fonctionnement du système, et est noté ω. Un faible coût est évidemment souhaitable pour l'hébergeur de services.

2.3 Configuration du système

Nous définissons la configuration κ d'un système multi-étagé composé de M étages par la combinaison d'une configuration architecturale et d'une configuration locale :

$$\kappa \ < AC, LC >$$

Dans cet article, la *configuration architecturale* d'un système multi-étagé est un tableau

$$AC < AC_1, AC_2, \ldots, AC_M >$$

où AC_i est le nombre de ressources (i.e. de machines) à l'étage T_i du système. La *configuration locale* d'un système multi-étagé est un tableau

$$LC < LC_1, LC_2, ..., LC_M >$$

, où LC_i est le degré de multiprogrammation (MPL) à l'étage T_i du système. Par exemple, la configuration architecturale du système à 3 étages présenté sur la figure 1 est $AC < 3, 2, 3 >$ et la configuration locale pourrait être $LC < 200, 160, 100 >$, bien que non représentée sur la figure 1.

3. MODÉLISATION DES APPLICATIONS MULTI-ÉTAGÉES

Nous proposons dans cette section un modèle analytique pour évaluer les performances et le coût des systèmes multi-étagés hébergés sur des grappes de machines.

Le système est modélisé par un réseau fermé de files d'attentes basé sur l'algorithme MVA (*Mean Value Analysis*) [22] et son application proposée dans [25]. En plus de la quantité de charge N (nombre de clients accédant au système de manière concurrente), nous avons étendu les entrées du modèle avec les configurations locales et architecturales (κ) du système à prendre en compte. En plus de la latence ℓ, nous avons étendu le modèle pour qu'il produise en sortie le taux d'échec α et le coût ω du système multi-étagé. L'algorithme 1 présente le modèle proposé.

Le calcul de la latence moyenne des requêtes client est principalement basé sur l'algorithme MVA [22], et s'inspire de [25] (voir lignes 19 à 28 de l'algorithme 1). La seule différence est l'intégration de la réplication des serveurs, un étage pouvant être constitué de plusieurs ressources.

En plus des entrées, le modèle est paramétré pour une application multi-étagée avec : le nombre d'étages M de l'application, le *think time* Z, les *visit ratio* $V < V_1, V_2, ..., V_M >$ et les *service time* $S < S_1, S_2, ..., S_M >$ [25].

Le *think time* Z est le temps moyen entre la réception de la réponse d'une requête par un client et l'envoi par ce même client de la requête suivante. Les *visit ratio* caractérisent l'effet d'une requête client sur les différents étages de l'application. En effet, quand une requête accède à une application multi-étagée sur l'étage T_1, elle peut générer des requêtes sur l'étage T_2, puis T_3 et ainsi de suite jusqu'à T_M. Le visit ratio V_i est la moyenne des requêtes générées à l'étage T_i par une requête accédant à l'application. Les *service time* correspondent à la période de temps incompressible

Algorithme 1 : MO: Modelisation de systèmes multi-étagés

Entrées :
N : *# clients (i.e. quantité de charge)*
$\kappa < AC, LC >$: *configuration du système multi-étagé*
Sorties :
ℓ: *latence*
α: *taux d'échec*
ω: *coût (i.e. #ressources)*
Données :
M : *# étages*
Z : *think time*
$V < V_1, V_2, ..., V_M >$: *visit ratios*
$S < S_1, S_2, ..., S_M >$: *service times*

1 **Initialisation:**
2 $R_0 = Z$; $\tau = 0$; $\omega = 0$;
3 /* *taux d'échec* */
4 **pour** $m = 1; m \leq M; i++$ **faire**
5 **si** $m == 1$ **alors**
6 $wa_m = N$;
7 **sinon**
8 $wa_m = we_{m-1} \cdot Min(1, \frac{V_m}{V_{m-1}})$;
9 $we_m = Min(wa_m, LC_m \cdot AC_m)$; /* charge accédant au système */
10 **pour** $m = M; m > 0; m--$ **faire**
11 **si** $m == M$ **alors**
12 $ar_m = Max(wa_m - we_m, 0)$;
13 **sinon**
14 $ar_m = Max(wa_m - we_m, 0) + wa_{m+1} \cdot ar_{m+1} \cdot \frac{V_m}{V_{m+1}}$;
15 $ar_m = ar_m / wa_m$; /* taux d'échec à chaque étage */
16 $\alpha = ar_1$; /* taux d'échec global */
17 $N' = N \cdot (1 - \alpha)$; /* clients admis */
18 /* *latence* */
19 **pour** $m = 1; m \leq M; m++$ **faire**
20 $Ql_m = 0$;
21 $D_m = V_m \cdot S_m / AC_m$; /* demande de service */
22 **pour** $n = 1; n \leq N'; n++$ **faire**
23 **pour** $m = 1; m \leq M; m++$ **faire**
24 $R_m = D_m \cdot (1 + Ql_m)$;
25 $\tau = (\frac{n}{R_0 + \sum_{m=1}^{M} R_m})$; /* débit */
26 **pour** $m = 1; m < M; m++$ **faire**
27 $Ql_m = \tau \cdot R_m$; /* Loi de Little */
28 $\ell = \sum_{m=1}^{M} R_m$; /* latence */
29 /* *coût* */
30 **pour** $m = 1; m \leq M; m++$ **faire**
31 $\omega = \omega + AC_m$; /* # ressources total */

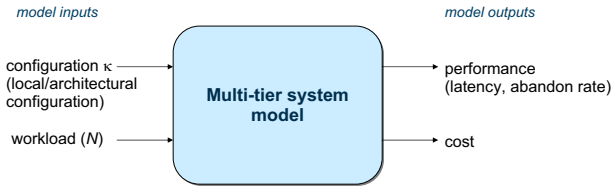

Figure 2: Modélisation d'une application multi-étagée

pour traiter une requête sur sur chaque étage. S_i représente donc le temps moyen requis pour traiter une requête sur l'étage T_i. Ces différents paramètres dépendent du comportement de l'application, et doivent donc être identifiés pour chaque application multi-étagée, par exemple en instrumentant l'application.

L'algorithme 1 calcule la latence, le taux d'échec et le coût d'une application multi-étagée. Le calcul de la latence est inspiré de [25], en étendant le modèle d'une application multi-étagée non dupliquée vers une application où chaque étage est constitué d'une grappe de machines (voir lignes 17 à 28). Notre algorithme donne une estimation du taux d'échec pour le système multi-étagé, en appliquant des techniques de contrôle d'admission basées sur la configuration du degré de multiprogrammation maximal des serveurs de l'application (lignes 3 à 16). Premièrement, on utilise le contrôle d'admission pour calculer, à partir de la charge wa_m tentant d'entrer sur l'étage T_m, quelle quantité de charge we_m peut réellement y accéder. Ensuite on calcule le taux d'échec ar_m pour chacun des étages T_m, ce qui permet d'otenir le taux d'échec global α. Enfin l'algorithme calcule le coût du système multi-étagé, correspondant au nombre total de ressources utilisées par le système (lignes 29 à 31). La complexité du modèle est de $0(NM)$, où N est le nombre de clients entrant dans le système, et M le nombre d'étages dans le système ($N >> M$).

4. PLANIFICATION DE CAPACITÉ

4.1 Fonction d'utilité

Etant donnés les objectifs de performance du contrat de qualité de service concernant la latence maximum ℓ_{max} et le taux d'échec maximum α_{max} qu'une application multi-étagée doit respecter, nous définissons la préférence de performance :

$$PP(\ell, \alpha) = (\ell \leq \ell_{max}) \cdot (\alpha \leq \alpha_{max}) \quad (1)$$

où ℓ et α représentent respectivement la latence et le taux d'échec d'une application multi-étagée. A noter que $\forall \ell, \forall \alpha, PP(\ell, \alpha) \in \{0, 1\}$, suivant si les conditions de l'équation 1 sont respectées ou non. A partir de la préférence de performance et du coût de l'application multi-étagée, nous définissons une fonction d'utilité Θ qui combine ces différents critères comme suit :

$$\Theta(\ell, \alpha, \omega) = \frac{M \cdot PP(\ell, \alpha)}{\omega} \quad (2)$$

où ω est le coût (en nombre de ressources) et M le nombre de ressources de l'application multi-étagée. M est utilisé dans l'équation 2 à fin de normalisation. Ici,

$$\forall \ell, \forall \alpha, \forall \omega, \Theta(\ell, \alpha, \omega) \in [0, 1]$$

car $\omega \geq M$ (avec au moins un serveur par étage) et $PP(\ell, \alpha) \in \{0, 1\}$.

Une valeur élevée de la fonction d'utilité signifie que d'une part les performances de l'application respectent les objectifs spécifiés dans le contrat de qualité de service (SLA), et d'autre part que le coût de fonctionnement de l'application est faible. Afin d'illustrer le comportement de la fonction d'utilité, un ensemble de données synthétiques pour la configuration d'une application à 3 étages est donné dans le tableau 1 et la figure 3. Trois charges différentes sont considérées, avec respectivement 10, 100 et 1000 clients. Dans le tableau 1, pour chaque charge, différentes configurations κ_i de l'application multi-étagée sont étudiées, avec différentes valeurs aux niveaux local et architectural. La figure 3 donne, pour chaque configuration, la valeur correspondante de la fonction d'utilité. Par exemple, avec une charge de 1000 clients, la plus haute valeur de la fonction d'utilité est obtenue avec la configuration κ_{16}, qui garanti les objectifs de performance avec un total de 9 ressources. Avec la même charge, si l'application a une configuration architecturale plus faible (donc moins de ressources), comme κ_{13}, ou une configuration locale plus faible (donc un MPL plus bas) comme κ_{14}, la préférence de performance pour la latence et le taux d'échec ne sont plus garantis. Inversement, si une configuration architecturale plus élevée comme κ_{17} ou κ_{18} est utilisée, cela augmente le coût de l'application multi-étagée sans améliorer les performances, et mène donc à une diminuation de la fonction d'utilité.

Dans le cas d'une configuration locale plus élevée, comme κ_{15}, l'application sera soumise à un degré de concurrence plus élevé et ne pourra plus respecter les objectifs de performance, ce qui réduira la valeur de la fonction d'utilité à 0. De même, avec une charge de 100 clients et 10 clients, les configurations garantissant les performances de l'application avec un coût minimal sont celles maximisant la fonction d'utilité, respectivement κ_{10} et κ_2.

Charge	Description de la configuration
10 clients	$\kappa_1 : AC < 1, 1, 1 >, LC < 50, 50, 50 >$
	$\kappa_2 : AC < 1, 1, 1 >, LC < 100, 100, 50 >$
	$\kappa_3 : AC < 1, 2, 1 >, LC < 100, 50, 100 >$
	$\kappa_4 : AC < 1, 2, 2 >, LC < 200, 100, 150 >$
	$\kappa_5 :< AC < 2, 2, 3 >, LC < 200, 200, 100 >$
	$\kappa_6 :< AC < 2, 3, 4 >, LC < 300, 200, 200 >$
100 clients	$\kappa_7 :< AC < 1, 1, 1 >, LC < 50, 50, 50 >$
	$\kappa_8 :< AC < 1, 2, 2 >, LC < 50, 50, 50 >$
	$\kappa_9 :< AC < 2, 2, 2 >, LC < 20, 50, 50 >$
	$\kappa_{10} :< AC < 1, 2, 2 >, LC < 100, 50, 50 >$
	$\kappa_{11} :< AC < 2, 2, 2 >, LC < 100, 100, 50 >$
	$\kappa_{12} :< AC < 2, 4, 4 >, LC < 200, 100, 200 >$
1000 clients	$\kappa_{13} :< AC < 2, 2, 4 >, LC < 100, 200, 100 >$
	$\kappa_{14} :< AC < 2, 3, 4 >, LC < 200, 200, 200 >$
	$\kappa_{15} :< AC < 2, 3, 4 >, LC < 500, 400, 400 >$
	$\kappa_{16} :< AC < 2, 3, 4 >, LC < 300, 200, 200 >$
	$\kappa_{17} :< AC < 2, 4, 4 >, LC < 300, 200, 200 >$
	$\kappa_{18} :< AC < 4, 3, 4 >, LC < 300, 200, 200 >$

Table 1: Exemples de configurations

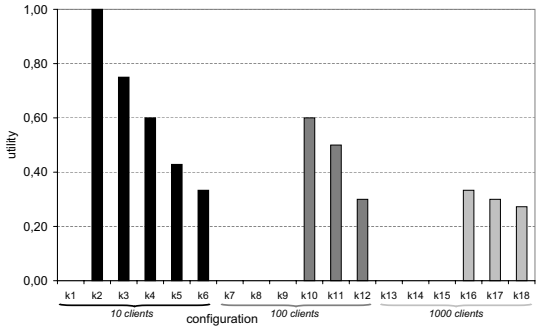

Figure 3: Fonction d'utilité

4.2 Planification de capacité pour une utilité maximale

Nous proposons une méthode de planification de capacité basée sur la fonction d'utilité présentée en 4.1, afin de calculer les configurations locales et architecturales *optimales* d'un système multi-étagé, de manière à ce que les contraintes de performances soient respectées tout en minimisant le coût de l'application. Le calcul de la configuration optimale d'une application multi-étagée est donc équivalent au calcul de la configuration pour laquelle la valeur de la fonction d'objectif est maximale (i.e. optimale, Θ^*). L'algorithme 2 décrit notre algorithme de planification de capacité pour des applications multi-étagées basées sur grappes de machines. Le fonctionnement général de l'algorithme est présenté sur la figure 4. Etant donnés une charge et des objectifs de performance en terme de latence et de taux d'échec maximum, un algorithme de planification de capacité produit une configuration initiale minimale pour l'application multi-étagée, calcule les performances de cette configuration en utilisant le modèle de l'application, puis compare les performances obtenues par rapport aux objectifs de performances. Si les objectifs de performances sont vérifiés, l'algorithme de planification de capacité retourne cette configuration et se termine, sinon il itère avec une nouvelle configuration plus coûteuse de l'application et recommence le processus.

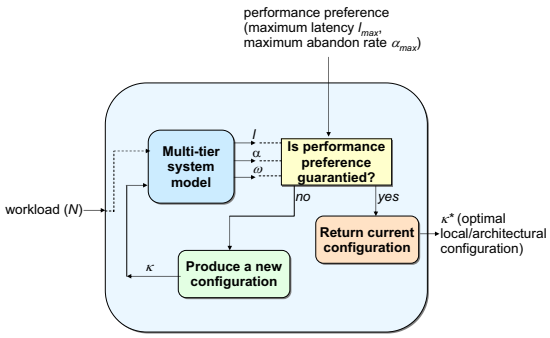

Figure 4: Algorithme de planification de capacité

Algorithme 2 : KA: Planification de capacité des applications multi-étagées - Recherche dichotomique

Entrées :
N : *# clients (i.e. quantité de charge)*
Sorties :
$\kappa^* < AC, LC >$: *configuration du système multi-étagé*
Données :
ℓ_{max} : *latence maximum*
α_{max} : *taux d'échec maximum*
M : *# étages*
$MPL_{max} < MPL_{max-1}, ..., MPL_{max-M} >$: *MPL maximum à chaque étage*
$V < V_1, V_2, ..., V_M >$: *visit ratios*
MO : *modèle de l'application multi-étagée*

Initialisation:
/* *soustraire* α_{max} *% de la charge entrante* */
$N' = N * (1 - \alpha_{max})$
$\alpha'_{max} = 0$
/* *configurations locale et architecturale initiales* */
pour $m = 1; m \leq M; m++$ **faire**
  $AC_m = 1$
  $LC_m = N' \cdot V_m$
$o^* = 0$
$\kappa^* = < \emptyset, \emptyset >$
/* *AC et LC vérifiant les conditions* ℓ_{max} *et* α'_{max} */
tant que $\ell > \ell_{max} \vee \alpha > \alpha'_{max}$ **faire**
  pour $m = 1; m \leq M; m++$ **faire**
    $AC_m = AC_m + 1$
    $LC_m = MIN(\frac{N' \cdot V_m}{AC_m}, MPL_{max-m})$
  $< \ell, \alpha, \omega > = MO(N', \kappa < AC, LC >)$
/* *minimisation du coût de* AC */
pour $m = 1; m \leq M; m++$ **faire**
  /* *recherche dichotomique de* AC^*_m *dans* $[1 \dots AC_m]$ */
  $AC'_m = \frac{AC_m}{2}$
  $LC'_m = MIN(\frac{N' \cdot V_m}{AC'_m}, MPL_{max-m})$
  $AC' = < AC_1, ..., AC'_m, ..., AC_M >$
  $LC' = < LC_1, ..., LC'_m, ..., LC_M >$
  $< \ell, \alpha, \omega > = MO(N', \kappa < AC', LC' >)$
  if $\ell > \ell_{max} \vee \alpha > \alpha'_{max}$ **then**
    poursuivre la recherche dichotomique de AC^*_m *dans* $]\frac{AC_m}{2} \dots AC_m]$
  else
    poursuivre la recherche dichotomique de AC^*_m *dans* $[1 \dots \frac{AC_m}{2}]$
/* *à la fin :* $AC < AC^*_1, ..., AC^*_m, ..., AC^*_M >$ */
/* *amélioration de LC* */
for $m = 1; m \leq M; m++$ **do**
  /* *recherche dichotomique de* $LC^*_m in\ [LC_m \dots MPL_{max-m}]$ */
  $LC'_m = \frac{MPL_{max-m} - LC_m}{2} + LC_m$
  $LC' = < LC_1, ..., LC'_m, ..., LC_M >$
  $< \ell, \alpha, \omega > = MO(N, \kappa < AC, LC' >)$
  if $\ell > \ell_{max}$ **then**
    poursuivre la recherche dichotomique de LC^*_m *dans* $[LC_m \dots LC'_m[$
  else
    poursuivre la rech. dichotomique de LC^*_m *dans* $[LC'_m \dots MPL_{max-m}]$
/* *à la fin :* $LC < LC^*_1, ..., LC^*_m, ..., LC^*_M >$ */

Recherche dichotomique. Nous proposons un algorithme de planification de capacité pour les applications multi-étagées dans l'algorithme 2. Cet algorithme est composé de 3 parties : premièrement, il garde seulement une partie de la quantité de charge entrante, en fonction du taux d'échec autorisé, puis il calcule AC_{max}, la borne maximum pour le degré de duplication sur chaque étage; deuxièmement il effectue une recherche dichotomique dans l'espace des configurations architecturales; troisièmement il améliore la configuration locale LC. Pour déterminer AC_{max}, on calcule la fonction d'utilité avec $AC = 1$ sur chaque étage, puis on ajoute une ressource sur chaque étage jusqu'à respecter les contraintes de performance. Le AC_{max} ainsi obtenu devient la borne maximale des configurations architecturales. Une recherche dichotomique est utilisée pour trouver la configuration architecturale optimale, i.e. celle minimisant la consommation de ressources tout en respectant les contraintes de qualité de service. La configuration locale est adaptée à chaque étape en fonction du nombre de clients admis et des ressources allouées au système. La troisième partie a pour but de rendre notre algorithme plus robuste aux variations de charge. Sur chaque étage, on cherche la plus haute valeur de la configuration locale qui respecte les contraintes de qualité de service. Ainsi en utilisant cet algorithme, des reconfigurations du système peuvent être évitées dans le cas de faible variations de charge. La complexité de notre algorithme de planification de capacité est de :

$$O(N.M.AC_{max} + N.M^2.(\log_2(AC_{max}) + \log_2(LC_{max})))$$

où N est le nombre de clients entrant sur le système, M le nombre d'étages de l'application ($N >> M$), AC_{max} le degré de duplication maximum (sur l'étage constituant le goulot d'étranglement), et LC_{max} la valeur maximale de la configuration locale.

Recherche exhaustive. Un algorithme de planification de capacité utilisant une recherche exhaustive sur toutes les configurations locales et architecturales possibles d'un système multi-étagé est décrit dans l'algorithme 3. Son principe est de construire l'ensemble Φ_{AC} de toutes les configurations architecturales possibles et l'ensemble Φ_{LC} de toutes les configurations locales possibles. Toutes les combinaisons de Φ_{AC} et Φ_{LC} sont des configurations possibles du système. On teste ces configurations de manière exhaustive afin de trouver celle qui maximise notre fonction d'objectif, tout en minimisant la consommation de ressources. Afin de construire les ensembles Φ_{AC} et Φ_{LC}, on doit borner les espaces des configurations locale et architecturale. Pour les configurations locales, on choisit la valeur maximale du MPL, qui dépend du logiciel utilisé sur l'étage considéré. Pour déterminer la valeur maximum de AC, on calcule la valeur de la fonction d'objectif avec $AC = 1$ sur chaque étage, puis on ajoute une ressource sur chaque étage jusqu'à ce que les contraintes de performances soient vérifiées. Le AC_{max} résultant sera la borne supérieure pour les configurations architecturales. La complexité de l'algorithme exhaustif de planification de capacité est de :

$$O(N.M.AC_{max} + N.M.(LC^M_{max} * AC^M_{max}))$$

où N est le nombre de clients accédant au système, M le nombre d'étages du système ($N >> M$), AC_{max} le degré de duplication maximum (sur l'étage constituant le goulot d'étranglement), et LC_{max} la valeur maximale de la config-

uration locale.

Algorithme 3 : Planification de capacité des applications multi-étagées -Recherche exhaustive

Entrées :
N : #clients (i.e. quantité de charge)
Sorties :
$\kappa^* < AC, LC >$: *configuration de l'application multi-étagée*
Données :
ℓ_{max} : *latence maximum*
α_{max} : *taux d'échec maximum*
M : *# étages*
$MPL_{max} < MPL_{max-1}, ..., MPL_{max-M} >$: *MPL maximum à chaque étage*
MO : *modèle de l'application multi-étagée*

Initialisation:
/ configurations locale et architecturale initiales */*
pour $m = 1; m \leq M; m++$ **faire**
  $AC_m = 1;$
  $LC_m = MPL_{max-m};$
$o^* = 0;$
$\kappa^* =< \emptyset, \emptyset >;$
/ déterminer AC_{max} comme $MAX\{AC_i\}$ */*
$< \ell,\alpha,\omega > = MO(N,\kappa < AC, LC >);$
tant que $\ell > \ell_{max} \vee \alpha > \alpha_{max}$ **faire**
  pour $m = 1; m \leq M; m++$ **faire**
    $AC_m = AC_m + 1;$
  $< \ell,\alpha,\omega > = MO(N,\kappa < AC, LC >);$
$AC_{max} = AC_1;$
/Φ_{AC} : toutes les configurations architecturales possibles */*
$\Phi_{AC} = \{AC < 1, ..., 1 >, ..., AC < AC_{max}, ..., AC_{max} >\}$
/ $|\Phi_{AC}| = (AC_{max})^M$ */*
/Φ_{LC} : toutes les configurations locales possibles */*
$\Phi_{LC} = \{LC < 1, ..., 1 >, ..., AC < MPL_{max-1}, ..., MPL_{max-M} >\}$
/ $|\Phi_{LC}| = \prod_{m=1}^{M} MPL_{max-m}$ */*
pour chaque $AC \in \Phi_{AC}$ **faire**
  pour chaque $LC \in \Phi_{LC}$ **faire**
    $< \ell,\alpha,\omega > = MO(N,\kappa < AC, LC >)$
    si $\Theta(\ \ell,\alpha,\omega) > o^*$ **alors**
      $o^* = \Theta(\ \ell,\alpha,\omega);$
      $\kappa^* =\kappa;$

4.3 Propriétés des algorithmes de panification de capacité

Nous évaluons par la suite les propriétés suivantes de algorithmes de planification de capacité :

La *précision* d'un algorithme de planification de capacité est le ratio entre le résultat de la fonction d'objectif quand on applique cet algorithme de planification, et le résultat de la fonction d'objectif quand on applique l'algorithme optimal de planification de capacité, ce dernier étant basé sur une recherche exhaustive.

L' *efficacité* d'un algorithme de planification de capacité est le temps nécessaire à l'algorithme pour calculer la configuration de l'application multi-étagée.

4.4 Preuve d'optimalité de la planification de capacité

Dans cette section, nous prouvons l'optimalité de la configuration produite par l'algorithme de recherche dichotomique, i.e. que la précision de la configuration est de 100%. Soit k la configuration produite par l'algorithme de recherche dichotomique étant données les contraintes l_{max} et α_{max} et une charge N. Par soucis de clarté, nous utilisons $\Theta(k)$ pour représenter l'utilité de cette configuration (au lieu de $\Theta(\ell, \alpha, \omega)$, voir 4.1). Soit k^* la configuration optimale pour un ensemble de contraintes données. On définit la précision de la configuration k comme suit :

$$acc(k) = \frac{\Theta(k)}{\Theta(k^*)}$$

Nous devons prouver que toute les configurations fournies par notre algorithme ont une précision égale à 1, ce qui est la valeur maximale. On procède par contradiction, en supposant qu'il existe une autre configuration k' telle que $\Theta(k') > \Theta(k)$.

$$\begin{aligned} & \exists k', \Theta(k') > \Theta(k) \\ \Leftrightarrow \quad & \exists k', \frac{M \cdot PP_{k'}(\ell,\alpha)}{\omega_{k'}} > \frac{M \cdot PP_k(\ell,\alpha)}{\omega_k} \end{aligned}$$

En supposant que les contraintes peuvent toujours être vérifiées (propriété P3), $PP_k(\ell,\alpha) = 1$ et $PP_{k'}(\ell,\alpha) = 1$. Donc :

$$\begin{aligned} & \exists k', \frac{1}{\omega_{k'}} > \frac{1}{\omega_k} \\ \Leftrightarrow \quad & \exists k', \omega_{k'} < \omega_k \\ \Leftrightarrow \quad & \exists k', \exists i \in 1; M, AC'_i < AC_i \end{aligned}$$

où AC et AC' sont les configurations architecturales de k et k' respectivement. Comme le coût total de k' est inférieur à celui de k, il y a au moins un étage de k' avec un degré de duplication plus faible que celui de k. On note k_{max} et k'_{max} le degré de duplication maximum de k et k' respectivement. k_{max} et k'_{max} sont calculés au début de notre algorithme pour borner l'espace des configurations architecturales.

Premier cas. Dans ce premier cas, on suppose que $k'_{max} > k_{max}$, i.e. qu'il existe au moins un étage i tel que $AC'_i < AC_i$ et un étage j tel que $AC'_j > AC_j$. La première partie de notre algorithme, qui calcule k_{max}, se termine lorsque les contraintes de qualité de service sont respectées, avec une configuration architecturale $< k_{max} \ldots k_{max} >$. Si notre algorithme se termine avec un k_{max} donné, les contraintes de performances sont vérifiées avec cette configuration, et il n'y a besoin d'ajouter des ressources sur aucun étage (propriété P1). Ce premier cas est donc impossible.

Deuxième cas. Maintenant, nous supposons que $k_{max} = k'_{max}$, donc k et k' ont le même degré de duplication maximum. Donc il existe un étage i tel que $AC'_i < AC_i$. Mais d'après la propriété P2 et la construction de l'algorithme dichotomique, au moins AC_i ressources sont requises sur l'étage i pour respecter les contraintes de performances :

$$\forall AC'_i, AC'_i < AC_i \Longrightarrow PP_{k'}(\ell,\alpha) = 0$$

Donc il est impossible d'avoir k' vérifiant les contraintes de performances l_{max} et α_{max} et ayant un coût inférieur à celui de k. Ainsi, la précision de la configuration k produite par l'algorithme de recherche dichotomique est toujours de 1, i.e. $acc(k) = 1$.

5. MOKA

Nous avons conçu et implémenté un canevas logiciel nommé MoKa pour la modélisation et la planification de capacité des applications multi-étagées. Ce prototype intègre l'implémentation du modèle analytique proposé en section 3 et l'algorithme de planification de capacité vu en section 4. De plus, MoKa est un framework ouvert, capable d'intégrer et de comparer différents modèles de performance et algorithmes de planification de capacité. En particulier, il est capable d'évaluer la *précision* et l'*efficacité* des algorithmes de planification de capacité.

Interface utilisateur. De plus, nous avons développé une interface Web pour MoKa, afin de permettre aux utilisateurs et aux administrateurs d'exécuter MoKaà distance depuis un navigateur Web et de calculer les configurations architecturales et locales d'applications multi-étagées[1]. Cet outil est utile pour les administrateurs, qui peuvent entrer les paramètres de leur système, des informations sur le type de charge à supporter, et MoKa tracera l'évolution des configurations locales et architecturals optimales (et donc suggérées) quand la charge augmente. La figure 5 présente l'interface utilisateur de MoKa.

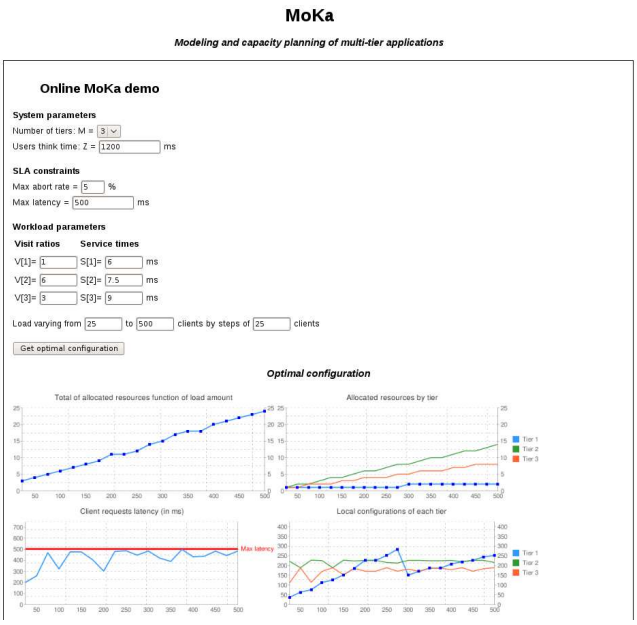

Figure 5: Interface Web de MoKa

6. EVALUATION

6.1 Environnement d'évaluation

Environnement matériel. Les expériences ont été réalisées sur une grappe de machines compatibles x86, avec des AMD bi-Opteron à 2,2GHz et 4 Go de RAM. Les machines sont connectées via un LAN Ethernet à 10 Gb/s.

Environnement logiciel. Nous avons utilisé le noyau Linux dans sa version 2.6.18 et les middleware suivants : le container de servlet Apache Tomcat 5.5.23 [24], le serveur de bases de données MySQL 5.0.37 [23], et les répartiteurs de charge PLB 0.3 [20] et Sequoia 2.10.6 [8].

Application. Nos expériences sur MoKa ont été réalisées avec un *site de vente aux enchères* basé sur Rubis, un banc d'essai d'application J2EE multi-étagée [1]. Rubis définit plusieurs interactions Web (enregistrement de nouveaux clients, achat et vente d'objets...). Il fournit un émulateur de client permettant de paramétrer le type de charge. Ceci nous permet de faire varier la charge pendant l'expérience. Le banc d'essai collecte des statistiques sur l'application comme la latence moyenne des requêtes client. Nous avons utilisé une version améliorée de Rubis 1.4.2, dans laquelle nous avons ajouté de nouvelles statistiques comme le taux d'échec des requête client. Le site de vente aux enchères a été déployé comme un système multi-étagé et dupliqué sur une grappe de machines. Le premier étage, ou *front-end*, intégre le serveur Web et d'entreprise dupliqué, et le deuxième étage, ou *back-end*, est constitué de serveurs de bases de données dupliqués.

Calibration du modèle et de la planification de capacité. Le modèle de performance et l'algorithme de planification de capacité sont calibrés à partir de mesures effectuées hors-ligne sur le site de vente aux enchères. Les paramètres utilisés pour les algorithmes 1, 2 et 3 sont donnés dans la table 2.

Modélisation	Planification de capacité
$M = 2$	$\ell_{max} = 1\ s$
$Z = 4\ s$	$\alpha_{max} = 10\%$
$V < 1, 4.32 >$	$V < 1, 4.32 >$
$S < 7.92\ ms, 19\ ms >$	$MPL_{max} < 1000, 1000 >$

Table 2: Paramètres de calibration

6.2 Validation du modèle

Le but de cette expérience est de valider la précision du modèle, en comparant les prédictions du modèle avec les mesures effectuées sur le système en fonctionnement réel. La configuration architecturale est $AC < 1; 1 >$, et la quantité de charge augmente à chaque point de mesure. La figure 6 compare les latences prédites et mesurées en fonction de la quantité de charge. L'erreur moyenne entre mesures et prédictions est inférieure à 12%.

6.3 Configurations ad-hoc vs configurations optimisées

Dans les expériences suivantes, nous comparons tout d'abord le comportement de systèmes multi-étagés configurés de manière ad-hoc (avec des configurations locale et architecturale fixées), avec le comportement de systèmes optimisés basés sur la méthode de planification de capacité pour une meilleure utilité proposée dans cet article. Ici, la planification de capacité a pour but de déterminer les meilleures configurations locale et architecturale d'un système multi-étagé de manière à ce que 90% des requêtes client soient traitées en moins de 1 seconde (voir tableau 2), et que le coût total du système soit minimisé. Le tableau 3 décrit premièrement les deux configurations ad-hoc considérées, $AA - AL_1$ et $AA - AL_2$, avec une configuration architecturale ad-hoc et une configuration locale ad-hoc (les autres configurations du tableau sont décrites en 6.4). $AA - AL_1$ et $AA - AL_2$ ont été choisies pour représenter respectivement une petite configuration (avec quelques machines et un niveau de concur-

[1] *http://sardes.inrialpes.fr/research/moka/*

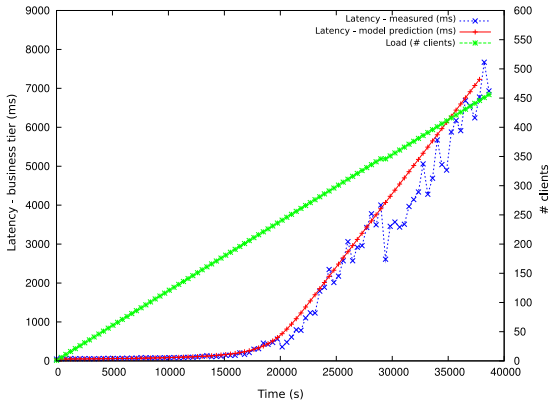

Figure 6: Site de vente aux enchères - Précision du modèle

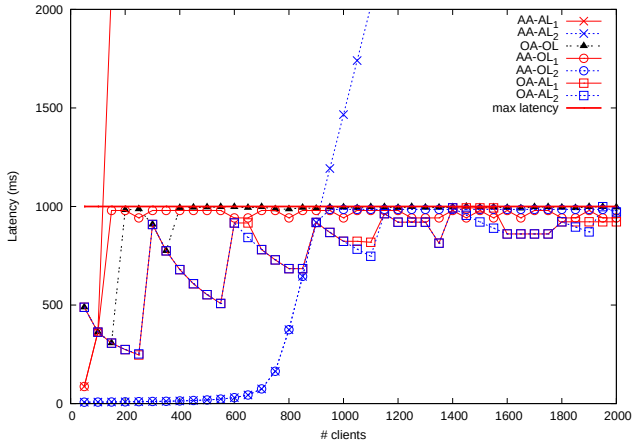

Figure 7: Site de vente aux enchères – Latence

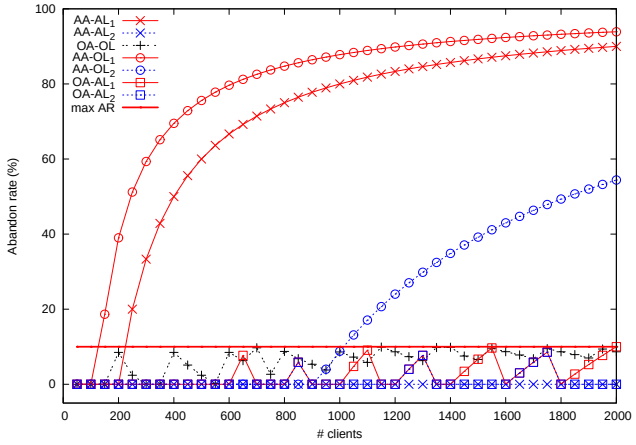

Figure 8: Site de vente aux enchères – Taux d'échec

rence limité sur les serveurs), et une grande configuration. De plus, la configuration locale de $AA-AL_1$ est fixée aux valeurs par défaut du MPL de Tomcat et MySQL (respectivement 200 et 100) pour le site de vente aux enchères, et une valeur intermédiaire de 150 pour le site Web de la Coupe du Monde. $OA-OL$ représente le système multi-étagé où les configurations locale et architecturales sont optimisées en utilisant notre méthode de planification de capacité. Les figures 7 et 8 présentent respectivement les variations de latence et de taux d'échec du site multi-étagé de vente aux enchères lorsque la charge varie. Ici, $AA-AL_1$ n'est pas capable de garantir la préférence de taux d'échec α_{max} lorsque la charge augmente car la configuration est trop petite (pas assez de machines et un MPL trop faible). Comme la concurrence est limitée, $AA-AL_1$ fournit néammoins une faible latence[2] qui respecte la préférence ℓ_{max}. Inversement, $AA-AL_2$ représente une grande configuration (plusieurs machines et une concurrence élevée sur les serveurs). Ainsi, aucune requête n'est abandonnée et toutes les requêtes sont exécutées en parallèle, ce qui a un impact direct sur la latence qui augmente au délà de la limite ℓ_{max} quand la charge augmente. La configuration entièrement optimisée $OA-OL$ contraste en garantissant à la fois les préférences de performance α_{max} et ℓ_{max}.

Config.	architecturale	locale
AA-AL_1	Ad-hoc $AC<1,2>$	Ad-hoc $LC<200,100>$
AA-AL_2	Ad-hoc $AC<6,15>$	Ad-hoc $LC<300,200>$
OA-OL	Optimisée	Optimisée
AA-OL_1	Ad-hoc $AC<1,2>$	Optimisée
AA-OL_2	Ad-hoc $AC<6,15>$	Optimisée
OA-AL_1	Optimisée	Ad-hoc $LC<200,100>$
OA-AL_2	Optimisée	Ad-hoc $LC<300,200>$

Table 3: Configurations du système

6.4 Optimisation locale vs optimisation architecturale

Les travaux précédents portant sur la planification de capacité de systèmes multi-étagés se limitent habituellement à un seul niveau de configuration, soit la configuration locale [9, 17, 27], soit la configuration architecturale [25]. Nous pensons que la combinaison de ces deux niveaux d'optimisation améliore le comportement global du système et permet de garantir simultanément plusieurs critères de performance, tout en économisant des ressources. Dans la suite, nous comparons des systèmes multi-étagés uniquement optimisés au niveau local, des systèmes uniquement optimisés au niveau architectural, et des systèmes où l'optimisation est effectuée sur les deux niveaux de configuration. En plus des configurations précédement présentées, le tableau 3 décrit 4 configuration additionelles [3] , nommées $AA-OL_1$, $AA-OL_2$, $OA-AL_1$ et $OA-AL_2$. $AA-OL_1$ et $AA-OL_2$ représentent deux configurations avec une configuration architecturale ad-hoc, et une configuration locale optimisée avec un contrôle d'admission comme présenté dans [17]. $OA-AL_1$ et $OA-AL_2$ représentent deux configurations dont la configuration locale est fixée de manière ad-hoc, et la con-

[2]Latence des requêtes terminées avec succès

[3]Ou plus précisément, des méthodes de construction de configuration,i.e. des méthodes de planification de capacité.

figuration architecturale est optimisée suivant une approche similaire à [25]. La figure 7 montre que les configurations $AA-OL_1$ et $AA-OL_2$ vérifient la préférence de performance ℓ_{max} car elles sont capables d'optimiser leur configuration locale en limitant la concurrence sur les serveurs. Cependant, ceci a un impact direct sur le taux d'échec qui croît lorsque la charge augmente, et donc ne respecte plus la préférence de taux d'échec comme on peut le voir sur la figure 8. Le taux d'échec croît plus tard avec $AA-OL_2$ qu'avec $AA-OL_1$ car le premier représente une configuration avec plus de ressources qui absorbent une partie de la charge. Enfin $OA-OL$ est une configuration optimisée localement et architecturalement, qui est capable de garantir à la fois les préférences de performance α_{max} et ℓ_{max}. Les configurations $OA-AL_1$ et $OA-AL_2$, avec une configuration architecturale optimisée et une configuration locale ad-hoc sont également capables de garantir les préférences de latence et de taux d'échec, comme illustré sur les figures 7 et 8. Mais le coût de ces configurations est élevé, comme nous allons le voir.

En plus de la comparaison des performances, nous évaluons et comparons aussi le coût (i.e. le nombre de ressources) de toutes les configurations comme décrit sur la figure 9. Evidemment, $AA-AL_1$, $AA-AL_2$, $AA-OL_1$ et $AA-OL_2$ ont un coût constant car leur configuration architecturale est fixée de manière ad-hoc, avec un coût de 3 pour $AA-AL_1$ et $AA-OL_1$, et un coût de 21 pour $AA-AL_2$ et $AA-OL_2$. Par contre le coût de $OA-AL_1$, $OA-AL_2$ et $OA-OL$ croît lorsque la charge augmente, car leur configuration architecturale est optimisée en fonction. $OA-AL_1$ a un coût plus élevé que $OA-AL_2$. Ceci est du au fait que $OA-AL_1$ a une configuration locale plus faible que celle de $OA-AL_2$, et donc une plus faible concurrence sur les serveurs est autorisée avec $OA-AL_1$. Donc, pour que $OA-AL_1$ garantisse la préférence de taux d'échec α_{max}, cette configuration a besoin d'augmenter son nombre de ressources et ainsi accroitre le nombre global de requêtes traitées en parallèle. $OA-OL$ présente un coût plus faible que $OA-AL_1$ et $OA-AL_2$ car, à l'inverse de ces dernières, $OA-OL$ est capable d'optimiser la configuration locale du système et donc de maximiser l'usage des serveurs lorsque c'est possible, avant d'utiliser des ressources supplémentaires.

En résumé, nos expériences montrent que comparée aux approches avec seulement une configuration locale où jusqu'à 94% des requêtes peuvent être rejetées, et comparée aux approches avec seulement une configuration architecturale où jusqu'à 15% des ressources peuvent être gaspillées, une approche qui combine à la fois configuration locale et architecturale permet de garantir les préférences de performance tout en minimisant le coût du système.

6.5 Précision de la planification de capacité

La figure 10 compare la *précision* des différentes méthodes de planification de capacité données dans le tableau 3, c'est à dire le ratio entre la valeur de la fonction d'utilité de la configuration retournée par un algorithme de planification de capacité, et la valeur de la fonction d'utilité de la configuration optimale. Les nombres de la figure 10 sont des valeurs moyennes résultantes des expériences décrites sur les figures 7, 8 et 9. Ces valeurs montrent que notre méthode combinant les configurations locale et architecturale fourni une précision de 100% avec une configuration optimale de l'application multi-étagée. Ainsi dans ces expériences, un

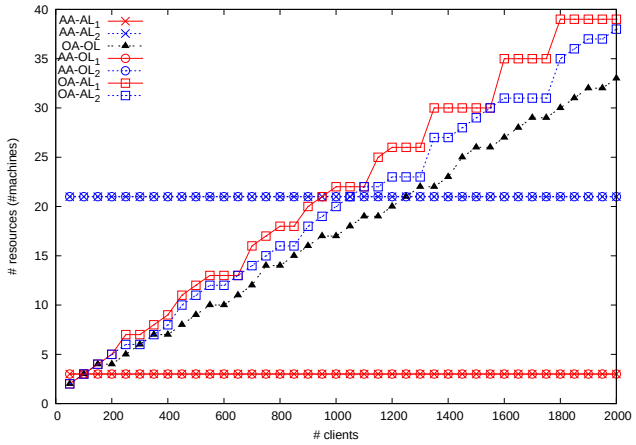

Figure 9: Site de vente aux enchères – Coût

seul niveau d'optimisation architecturale limite la précision entre 83% et 90%, un seul niveau d'optimisaton locale limite la précision entre 22% et 28%, et la précision d'une configuration ad-hoc est de 22%. Cependant, il est important de noter que des méthodes de planification de capacité seulement basées sur l'optimisation locale ou sur une configuration ad-hoc peuvent induire une précision de 0% dans le cas de charges élevées avec un faible taux d'échec autorisé.

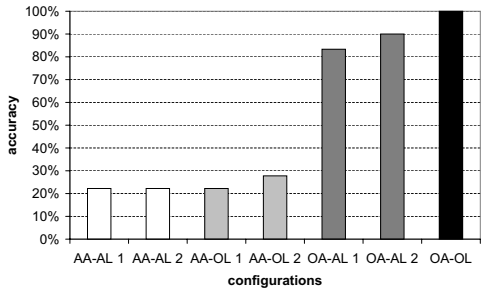

Figure 10: Site de vente aux enchères – Précision de la planification de capacité

6.6 Efficacité de la planification de capacité

La figure 11 compare le temps moyen de calcul des différentes méthodes de planification de capacité. Ces résultats correspondent aux expériences présentées dans les figures 7, 8 et 9. Globalement, les résultats montrent que plus on prend en compte de niveaux de configuration, plus le temps de calcul de la configuration augmente. Néammoins, il faut noter que comparée aux méthodes restraignant leur planification de capacité au niveau architectural, une méthode combinant configuration locale et architecturale n'augmente pas le temps de calcul. Au contraire, cette dernière méthode améliore l'efficacité de l'algorithme de planification de capacité car il produit des configurations architecturales plus faibles, et donc réduit le nombre de pas de l'algorithme. Afin de comparaison, nous considérons le cas de l'algorithme de

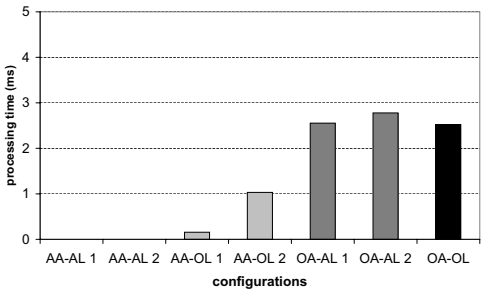

Figure 11: Site de vente aux enchères – Efficacité de la planification de capacité

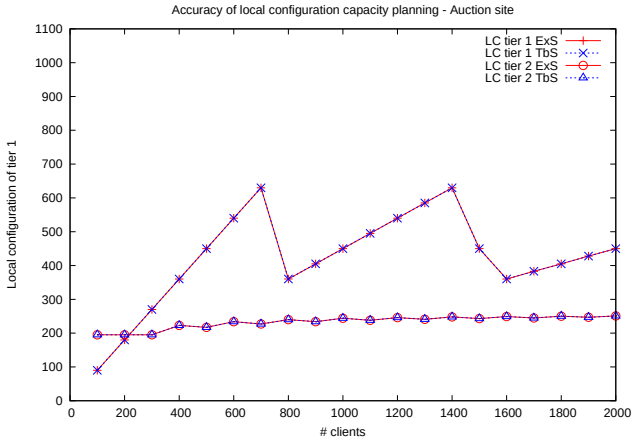

Figure 12: Site de vente aux enchères - Précision de la planification de capacité – niveau local

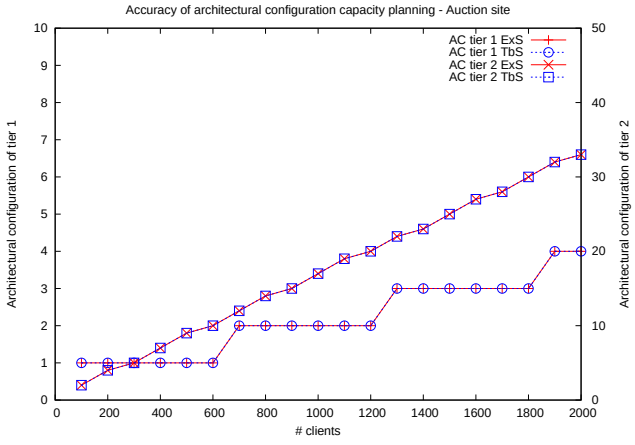

Figure 13: Site de vente aux enchères - Précision de la planification de capacité – niveau architectural

planification de capacité basé sur une recherche exhaustive (qui produit une configuration locale et architecturale optimale). Cet algorithme prend 11 minutes pour calculer la configuration optimale du site d'enchères, qui compte deux étages, avec 200 clients, et prend plus de 2 heures avec une charge de 600 clients.

7. ETAT DE L'ART

La planification de capacité est un point critique pour la qualité de service et la disponibilité des hébergeurs de services [16]. Alors que la plupart des projets existant prennent en compte des applications à un seul étage [9, 17, 27, 12, 26, 10], notre approche diffère car elle prend en compte des applications multi-étagées. De plus, les techniques utilisées pour effectuer la planification de capacité peuvent différer. D'un côté certains projets effectuent du contrôle d'admission au niveau local [9, 17, 27, 12, 18, 10], alors que d'autres utilisent l'allocation de ressources au niveau architectural des systèmes [25, 3, 26]. Une méthode d'adaptation des systèmes à la fois aux niveaux local et architectural a été proposée dans [18]. Cependant la configuration architecturale ne prend pas en compte l'allocation de ressources, mais la gestion de la répartition de charge entre un nombre fixé de machines. Cette solution est cependant complémentaire à celle que nous proposons dans cet article. Notre approche diffère donc des autres projets car elle effectue la planification de capacité des systèmes multi-étagés à la fois au niveau local et architectural, permettant un gain de qualité de service significatif et une minimisation des coûts. Contrairement aux approches basées sur des heuristiques qui ne garantissent pas la qualité de service des applications [3, 7, 10], cette approche est basée d'une part sur un modèle analytique prédisant les performances de l'application, et d'autre part sur un algorithme de planification de capacité efficace pour la configuration optimale des applications multi-étagées, avec des garanties strictes sur la qualité de service et l'utilisation des ressources. Le modèle proposé étend le modèle à files d'attente MVA *(Mean-Value Analysis)* [22]. Il intègre premièrement la configuration architecturale (i.e. la duplication) et la configuration locale (i.e. le contrôle d'admission) des systèmes multi-étagés. Il calcule ensuite l'impact des configurations locale et architecturale sur les performances et le coût du système. De plus, le modèle étendu calcule de nouvelles métriques de coût et de performance, comme le taux d'échec des requêtes client (un critère de qualité de service) et le coût de l'application multi-étagée en terme de nombre de machines affectées au service.

8. CONCLUSION

Dans cet article, nous présentons la conception et l'implémentation d'une méthode de planification de capacité pour les applications multi-étagées hébergées sur grappes de machines. La méthode proposée se base sur quatre éléments : (i) La combinaison de deux niveaux de configuration, local et architectural, pour les applications multi-étagées. Ceci permet d'économiser jusqu'à 15% de ressources avec une précision de 100% ; (ii) Un modèle analytique pour les applications multi-étagées qui calcule les performances et le coût des configurations ; (iii) Une fonction d'utilité qui caractérise l'impact des configurations locale et architecturale des applications multi-étagées sur leurs performances et leur coût ; (iv) Un algorithme de planification de capacité pour

calculer efficacement la configuration optimale des applications multi-étagées. Ces éléments sont implémentés dans le prototype de modélisation et de planification de capacité MoKa. Nos expériences menées sur un site Web et un site de vente aux enchères montrent l'intérêt de cette approche.

Remerciements

Les expériences présentées dans cet articles ont été réalisées en utilisant la plate-forme expérimentale Grid'5000, une initiative du Ministère de la Recherche français avec l'action ACI GRID, INRIA, CNRS, RENATER et d'autres partenaires[4].

9. REFERENCES

[1] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. In *The IEEE 5th Annual Workshop on Workload Characterization (WWC(5)*, Austin, TX, Nov. 2002.

[2] Apache. Apache HTTP Server. http://httpd.apache.org/.

[3] S. Bouchenak, N. D. Palma, D. Hagimont, and C. Taton. Autonomic Management of Clustered Applications. In *IEEE International Conference on Cluster Computing (Cluster 2006)*, Barcelona, Spain, Sept. 2006.

[4] B. Burke and S. Labourey. Clustering With JBoss 3.0.

[5] R. Buyya. *High Performance Cluster Computing - Volume 1*. Prentice Hall, 1999.

[6] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. In *ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[7] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. In *The 3rd IEEE International Conference on Autonomic Computing (ICAC 2006)*, Dublin, Ireland, June 2006.

[8] Continuent. Sequoia. http://sequoia.continuent.org/.

[9] Y. Diao, J. L. Hellerstein, S. Parekh, H. Shaikh, and M. Surendra. Controlling Quality of Service in Multi-Tier Web Applications. In *26th International Conference on Distributed Computing Systems (ICDCS 2006)*, Lisbon, Portugal, July 2006.

[10] S. Elnikety, J. Tracey, E. Nahum, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *The 13th international conference on World Wide Web*, 2004.

[11] X. He and O. Yang. Performance evaluation of distributed web servers under commercial workload. In *Embedded Internet Conference 2000*, San Jose, CA, Sept. 2000.

[12] H.-U. Heiss and R. Wagner. Adaptive load control in transaction processing systems. In *17th International Conference on Very Large Data Bases (VLDB 1991)*, Barcelona, Spain, Sept. 1991.

[4]voir https://www.grid5000.fr

[13] IBM. WebSphere Server. http://www.ibm.com/.

[14] Iron Mountain. The Business Case for Disaster Recovery Planning: Calculating the Cost of Downtime, 2001. Iron Mountain.

[15] J. Lee and R. Ben-Natan. *Integrating Service Level Agreements*. Wiley, 2002.

[16] D. A. Menasce and V. A. F. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall, 2001.

[17] D. A. Menasce, D. Barbara, and R. Dodge. Preserving qos of e-commerce sites through self-tuning: A performance model approach. In *ACM Conference on Electronic Commerce (EC'01)*, Tampa, FL, Oct. 2001.

[18] J. Milan-Franco, R. Jimenez-Peris, M. Patino-Martinez, and B. Kemme. Adaptive middleware for data replication. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 175–194, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[19] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems (TOCS)*, 23(4), 2005.

[20] PLB. PLB - A free high-performance load balancer for Unix. http://plb.sunsite.dk/.

[21] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. 2001.

[22] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2), 1980.

[23] Sun Microsystems. MySQL. http://www.mysql.com/.

[24] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.

[25] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. Analytic modeling of multitier internet applications. *ACM Transactions on the Web (ACM TWEB)*, 1(1):2, 2007.

[26] D. Villela, P. Pradhan, and D. Rubenstein. Provisioning servers in the application tier for e-commerce systems. *ACM Trans. Interet Technol.*, 7(1):7, 2007.

[27] Q. Zhang, L. Cherkasova, and N. Mi. A Regression-Based Analytic Model for Capacity Planning of Multi-Tier Applications. *Journal of Cluster Computing*, 11(3), 2008.