Université Joseph Fourier — Master 2 Recherche - Systèmes et Logiciels

---

Gestion de ressources dans les services Internet multi-niveaux : Modélisation et optimisation

Projet réalisé par :
Jean ARNAUD

Soutenu le :
18 juin 2007

SARDES (LIG/INRIA)

Responsable :
Sara BOUCHENAK

Jury :
Laurent BESACIER
Rachid ECHAHED
Arnaud LEGRAND
Jean-Marc VINCENT

Remerciements

Mes remerciements vont à Sara Bouchenak pour son encadrement et ses conseils, et à Jean-Bernard Stefani qui m'a permis d'effectuer ce stage.

Je tiens à remercier tout particulièrement Christophe Taton pour son aide précieuse et sa disponibilité, ainsi que Jérémy Philippe et Sylvain Sicard pour l'ambiance agréable de travail dans le bureau.

Merci enfin à l'ensemble des membres du projet SARDES pour leur accueil chaleureux.

Résumé

La constante augmentation du trafic vers les sites de commerce électronique, et les services Internet en général, oblige les hébergeurs de ces services à affecter toujours plus de ressources matérielles au traitement des requêtes. Certains services nécessitent ainsi des dizaines voire des centaines de serveurs. Le coût de ces machines devient donc un réel problème, et on a désormais besoin de solutions de gestion de ressources pour minimiser leur utilisation tout en fournissant une garantie de performances pour les applications hébergées.

De plus, les services Internet classiquement proposés sous la forme de simples serveurs Web voient leur complexité augmenter avec la venue de services Internet multi-niveaux. Les services multi-niveaux sont constitués d'une série de serveurs, par exemple des serveurs Web suivis de serveurs métier, eux-même suivis de serveurs de base de données.

Les approches actuelles de gestion de ressources dans les services Internet sont hélas souvent limitées. La plupart allouent statiquement un nombre de ressources en espérant que dans la majorité des cas cela suffira à traiter la charge, mais cette approche gaspille un nombre important de ressources souvent coûteuses. Pour y remédier certaines approches allouent dynamiquement des ressources par pas fixe en fonction de la demande, mais la réactivité d'une telle approche est faible et cela induit une latence importante avant que le système n'atteigne sa configuration optimale en terme de nombre de ressources.

Nous présentons dans ce rapport une approche permettant d'atteindre directement une configuration optimisée d'un service Internet. Cette approche configure au mieux chaque serveur et alloue un nombre de ressources optimal au système multi-niveaux par rapport aux performances attendues. Nous proposons une modélisation globale de l'application intégrant un modèle au niveau local d'un serveur simple ainsi qu'une modélisation globale de l'architecture multi-niveaux de l'application.

Des validations expérimentales préliminaires de cette approche ont été effectuées sur un service Internet modélisant le comportement d'un site d'enchères en ligne comme eBay, pour lequel nous avons pu montrer la validité de notre modèle.

Mots-clefs : systèmes multi-niveaux, modélisation, auto-optimisation, gestionnaires autonomes

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Contexte de stage

Ce travail à été réalisé au sein de l'équipe SARDES, à l'INRIA Rhône-Alpes[1]. SARDES est un projet INRIA et une équipe de recherche du laboratoire LIG-IMAG[2] (CNRS, INPG, UJF) [21]. Ce projet s'inscrit dans la perspective de l'émergence d'un environnement global de traitement de l'information. Dans ce contexte, le projet s'intéresse plus particulièrement aux systèmes répartis. La forte complexité de ce genre de systèmes empêche d'en garder un contrôle total, car la taille du système ainsi que le nombre de composants logiciels et matériels ne peut être géré par un administrateur humain. L'administration de ces systèmes devenant ainsi un problème en soit, les travaux menés à SARDES ont pour but de construire des solutions logicielles pour administrer de manière autonome des systèmes répartis complexes. Les outils proposés devront pouvoir s'introspecter pour obtenir des informations sur l'état du système, et agir sur l'état du système pour adapter la configuration de ce dernier en fonction des objectifs fixés. Ces objectifs peuvent être d'améliorer les performances (auto-optimisation), de s'auto-réparer ou encore de s'auto-protéger.

1.2 Contexte applicatif

La démocratisation d'Internet ainsi que la forte visibilité donnée aux sites de commerce électronique font croître chaque jour le nombre de transactions réalisées en ligne. Mais la constante augmentation du trafic vers les sites de commerce électronique, et les services Web en général, induit des charges de requêtes toujours plus élevées sur les serveurs. Cette hausse du nombre de transactions à traiter par les services Web oblige à construire des systèmes pouvant supporter plus de charge. Les systèmes modernes ont une architecture de type multi-niveaux et sont potentiellement répliqués (duplication d'un serveur pour le passage à l'échelle). Pour permettre aux système de supporter une charge plus élevée, il est possible d'ajouter des ressources physiques (par exemple dupliquer des serveurs), mais également d'optimiser l'utilisation des ressources existantes.

[1] http ://www.inrialpes.fr
[2] http ://lig.imag.fr

1.2.1 Services Internet multi-niveaux

Dans une architecture multi-niveaux (aussi appelée multi-étages), les différents serveurs sont exécutés sur des machines séparées, entre autre pour de meilleures performances, mais aussi pour faciliter la composition de services. De plus, chaque niveau est potentiellement dupliqué pour augmenter la capacité de traitement, ce qui est nécessaire pour le passage à l'échelle. Un exemple d'architecture multi-niveaux est présenté à la figure 3.3.

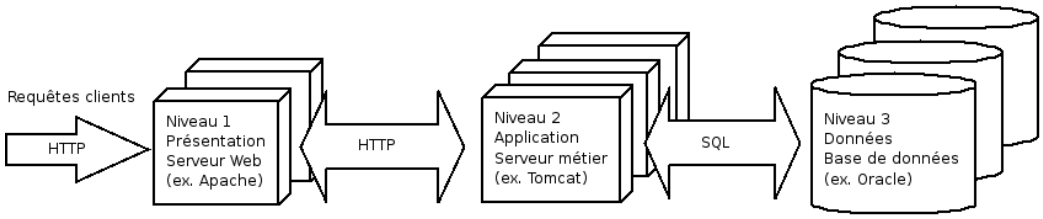

FIG. 1.1 – Architecture à 3 niveaux classique

Les requêtes des utilisateurs arrivent par le niveau 1 (présentation), qui fait office d'interface. Ensuite des requêtes sont transmises de niveaux en niveaux pour traitement. Par exemple dans le cas d'un service de vente en ligne, le niveau présentation recevra les requêtes des utilisateurs et interrogera le niveau application pour construire le résultat de la requête. Ce niveau interrogera la base de données sur le niveau 3, la mettra éventuellement à jour, construira la page et la renverra au niveau présentation qui se chargera de la transmettre à l'utilisateur. Du point de vue de l'utilisateur, l'architecture du service est et doit rester transparente.

1.2.2 Définition des critères de performance

Comme nous nous intéressons à l'optimisation de performances, il convient tout d'abord de définir clairement les critères d'évaluation de performance pour un système multi-niveaux. Menasce & Almeida considèrent que dans le cadre de services Internet, 4 critères principaux sont à considérer [16] :

- Temps de réponse pour le client (s)
- Débit (req/s)
- Disponibilité (%)
- Coût ($/(req/s))

Temps de réponse client (latence) Le premier est la latence, c'est-à dire le temps de traitement de la requête, qui est critique pour le client, car cela correspond à la période pendant lequel il attend la réponse à sa requête. La latence d'une requête est exprimée en millisecondes et sera calculée du côté client depuis l'envoi de la requête au serveur jusqu'à la réception de la réponse. Nous chercherons donc à minimiser ce critère.

Débit. C'est le nombre de requêtes traitées pendant un laps de temps donné, souvent en une seconde. Le débit se mesure du côté serveur. Ce facteur tend à être maximisé. En effet un

débit plus élevé permet d'améliorer la rentabilité du serveur. Le débit est mesuré en requêtes par secondes.

Disponibilité. Il s'agit du pourcentage de temps pendant lequel le service va être disponible. Par exemple on peut souhaiter avoir un service disponible 99% du temps. Ce critère est mesuré en pourcentage.

Coût. Le coût est habituellement associé à une mesure de performance, c'est à dire prix divisé par ratio de performance. Ceci indique combien doit être dépensé par unité de débit par exemple ($/tpm[3]) Nous cherchons évidement à minimiser ce ratio.

Charge. La charge d'un serveur Internet sera définie par le nombre de requêtes envoyées en parallèle par les clients à ce serveur. La charge se mesure en nombre de requêtes.

1.2.3 Configuration d'un serveur simple : taux de parallélisme

La grande majorité des serveurs (PostGreSQL[4], Apache[5], ...) sont multiprogrammés, c'est à dire que plusieurs processus (ou processus légers) d'un même serveur s'exécutent en parallèle sur la machine. Ceci permet de traiter plusieurs requêtes en parallèle et ainsi augmenter les performances de l'application. Plusieurs clients peuvent donc être connectés simultanément au serveur, comme illustré sur la figure 1.2.

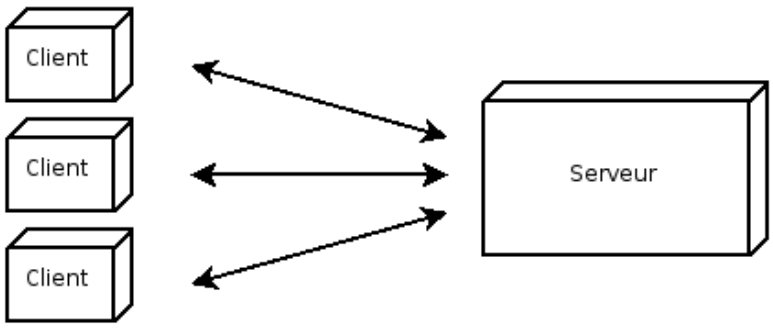

FIG. 1.2 – Fonctionnement client/serveur

Afin d'éviter l'écroulement du serveur *(trashing)*, il faut cependant définir une limite maximale au nombre de processus pouvant être exécutés en parallèle. Il s'agit du taux de multiprogrammation maximal, souvent noté MPL (pour *MultiProgrammation Level*). Le MPL est dans la majorité des cas un paramètre de configuration disponible sur les systèmes multiprogrammés.

De plus, il a été montré dans [9] qu'il existe un MPL optimal pour un contexte donné (voir section 2.3.2). Ce MPL optimal dépend principalement du type de charge injectée dans le serveur. En effet dans le cas de lectures seules, un taux de parallélisme élevé permettra de

[3] *transaction per minute*
[4] http ://www.postgresql.org
[5] http ://httpd.apache.org

traiter beaucoup de requêtes en parallèle, et augmentera ainsi le débit du serveur. Inversement, si le trafic est surtout composé d'écritures, il vaudra mieux garder un MPL faible pour éviter des effets d'invalidation de résultats de requêtes mettant à jour les mêmes données *(rollback)*. Dans ce cas, avec un MPL faible, les performances du système seront finalement meilleures. Il faut donc adapter dynamiquement le taux de parallélisme maximal en fonction du ratio lectures / écritures du flot de requêtes en entrée du serveur.

1.2.4 Configuration d'un système multi-niveaux : nombre de serveurs

Dans le cadre de systèmes multi-niveaux, nous chercherons à minimiser le nombre de serveurs utilisés. Notre but sera donc d'obtenir les meilleures performances, ou du moins des performances acceptables (respectant le SLA) tout en utilisant le moins de serveurs possibles. En effet dans la réalité les services Internet sont hébergés chez des fournisseurs de service qui disposent d'une réserve globale de machines et hébergent plusieurs services. Si on affecte plus de ressources que nécessaire à un service en particulier, moins de serveurs seront disponibles pour les autres, et le nombre de services pouvant fonctionner simultanément sera donc plus faible. Il en résultera un rapport bénéfice / coût plus faible pour l'hébergeur.

De plus la notion de ressource ne se limite pas à des machines, mais peut être par exemple la consommation électrique de ces dernières. Certaines approches cherchent à caractériser la consommation électrique des serveurs pour adapter la fréquence des processeurs à la charge à traiter [6]. Ceci permet de réduire les coûts en électricité, et donc en climatisation et matériel (durée de vie des machines allongée).

L'affectation de ressources à bon escient est donc une problématique concrète importante.

1.3 Position du problème

1.3.1 Problématique

La complexité toujours croissante des systèmes informatiques ne va pas sans poser de nombreux problèmes. Ainsi des tâches d'administration peuvent se révèler extrèmement ardues pour des systèmes à l'architecture multi-niveaux, où certains niveaux peuvent éventuellement être dupliqués. La compréhension et la modélisation de tels systèmes est un enjeu crucial actuellement, car des contraintes fortes pèsent sur les fournisseurs de services. Ces derniers doivent en effet souvent respecter des contraintes fortes concernant les performances du service fourni à leurs clients. On peut par exemple fixer un temps de réponse maximal pour un système de vente en ligne. Le non respect de ces différents critères, donnés dans le SLA[6], peut donner lieu à versement de pénalités par le fournisseur de service, qui aura donc tout intérêt à respecter le contrat. Pour pouvoir statisfaire les critères de performance du SLA, le fournisseur de service peut décider de reconfigurer son système (configuration des serveurs, nombre de duplicatas, politique de cache,...) en cas d'évolution de la charge. Cependant les effets de telles reconfigurations sont mal connus et dépendent la plupart du temps de l'expérience de l'administrateur du système. De plus la réactivité n'est pas garantie avec un administrateur humain, surtout quand les variations de charge sont rapides et importantes.

[6] *Service Level Agreement*

1.3.2 Approches existantes

La plupart des approches existantes actuellement pour traiter ces problèmes d'optimisation sont limitées soit par le niveau de connaissance du système, soit par la complexité de l'architecture prise en compte. En effet une grande partie des solutions existantes ont pour but de réagir au mieux aux changements survenant dans l'environnement du système, suivant des heuristiques plus ou moins efficaces. Ces approches ont un principe de fonctionnement commun qui est d'essayer différentes configurations du système et de mesurer laquelle donne les meilleures performances [9, 15]. Même si ce genre d'approche semble efficace dans le cas d'environnements stables ou évoluant peu rapidement, elles ne peuvent plus fonctionner si la charge varie brusquement (pics de charge).

Il apparait donc aujourd'hui qu'une modélisation des systèmes administrés est requise. Mais là encore les approches proposant une modélisation sont souvent limitées, la plupart étant soit peu génériques [1, 7], soit limitées à un niveau d'architecture (par exemple gérant seulement le taux de parallélisme maximal d'un serveur simple [9, 19, 7], ou alors le degré de duplication d'un serveur [22, 24], mais pas les deux). Il manque à ce jour une approche à la fois générique et optimisant le système à la fois au niveau serveur et au niveau architectural de l'application.

1.4 Objectifs du stage

Ce stage vise à concevoir et mettre en oeuvre un gestionnaire de ressources pour les services Internet multi-niveaux. Ses principaux objectifs sont la modélisation de services Internet multi-niveaux, l'optimisation de la configuration de tels systèmes, et la mise en oeuvre d'un gestionnaire de ressources qui permette d'optimiser dynamiquement les ressources des services Internet sur la base du modèle proposé.

Proposition d'un modèle

La phase de modélisation aura pour but de fournir un modèle complet d'une architecture multi-niveaux, intégrant à la fois le comportement local d'un serveur simple et le comportement global de serveurs multi-niveaux. Nous allons donc chercher dans cette étude à caractériser puis à modéliser des systèmes à l'architecture complexe, pour pouvoir prédire dans un environnement donné la configuration offrant les meilleures performances. Le modèle devra permettre de prédire de manière assez précise le comportement du système en fonction de la charge injectée, et de la configuration courante du système (nombre de serveurs sur chaque niveaux et configuration de ces serveurs). Nous étendrons pour cela les approches existantes visant à l'auto-optimisation au niveau architectural [22, 24] et au niveau local [9, 19, 7]. Il faudra étudier si les deux approches sont orthogonales ou non, et définir au besoin la méthode pour les coordonner. Ceci permettra de prédire pour une charge donnée la configuration globale du système qui donnera les meilleures performances.

Gestionnaire de ressources

Nous allons concevoir un gestionnaire de ressources capable de configurer dynamiquement un système multi-niveaux complexe en fonction de la charge. Le but sera d'optimiser le système à la fois au niveau local et architectural, afin de garantir des temps de réponse les plus

bas possibles, tout en utilisant un nombre de resource minimum. Pour cela nous concevrons un modèle le plus générique possible du système. Nous essaierons d'avoir un modèle facilement paramétrable, ou mieux se paramétrant automatiquement en fonction du contexte. Ce modèle permettra de prédire pour une charge et une configuration donnée quelles seront les performances du système. Notre but sera donc de maximiser une fonction de performance intégrant les différentes métriques à optimiser (latence, débit et nombre de ressources utilisées). Une fois le modèle développé, nous pourrons réaliser un prototype de gestionnaire utilisant ce modèle pour optimiser la configuration d'un système multi-niveaux afin de garantir des performances optimales. Cette optimisation se fera à deux niveaux, en élargissant l'approche présentée dans [17].

1.5 Organisation du rapport

Le rapport est organisé selon le plan suivant. Le chapitre 2 présente un état de l'art sur les domaines de la modélisation de services Internet et la construction de solutions pour la gestion de ressources.

Une modélisation d'un système multi-niveaux complexe ainsi que la conception d'un gestionnaire d'optimisation de ressources est ensuite présentées dans le chapitre 3. La mise en oeuvre de cette modélisation et la réalisation du prototype de gestionnaire pour la reconfiguration de systèmes multi-niveaux est présentée au chapitre 4. Enfin nous concluerons et nous présenterons les perspectives dégagées par ces travaux dans le chapitre 5.

Chapitre 2

Etat de l'art

2.1 Organisation

L'étude des systèmes multi-niveaux est actuellement effectuée selon 2 axes. Le premier porte sur la modélisation mathématique de tels systèmes. Ceci a pour but de définir des modèles précis pour prédire le comportement d'un système pour une configuration donnée et sous une charge donnée. Le deuxième axe a pour but la conception de solution logicielles, principalement à base de boucles de commande, pour le monitoring et la reconfiguration de services Internet. Actuellement la majorité des travaux dans ce domaine portent sur la reconfiguration de serveurs simples (au niveau d'un seul niveau), par exemple gérer le niveau de duplication ou les caches d'une base de données. Mais des études récentes commencent à s'intéresser au problème de la reconfiguration de systèmes multi-niveaux de manière plus globale.

Après avoir défini des critères de classification pour les approches existantes, nous allons présenter dans ce chapitre un état de l'art dans les domaines de la modélisation et de l'optimisation de gestion de ressources pour les systèmes multi-niveaux.

2.2 Critères de classification

Les différents critères de classification pouvant être retenus sont :

Calcul de la configuration. Pour obtenir la meilleure configuration d'un système, il existe principalement deux types d'approches, celles basées sur une heuristique simple et celles utilisant un modèle de l'application. Les premières déplacent en général le paramètre de configuration d'un pas fixé à l'avance, et constatent si les performances ont été améliorées ou non. Au contraire, les approches basées sur modèle visent à permettre de trouver directement une valeur optimum du paramètre de configuration.

SISO/MIMO. Dans le cadre d'un article portant sur la modélisation de systèmes, l'approche porte t'elle sur un seul paramètre en entrée et en sortie (par exemple prendre en entrée le nombre de requêtes entrantes et en sortie et en sortie le temps de réponse moyen), ou alors prend elle en compte plusieurs paramètres, ce qui implique d'étudier leurs interactions ?

Cadre applicatif. Les expériences effectuées portent-t'elles sur un système réel ? Si oui lequel ? En effet nombre d'études n'ont pas été expérimentées sur un système réel, et restent donc uniquement théoriques.

Complexité du système. Nous avons vu que l'optimisation dans les systèmes de commerce électronique peut se faire en plusieurs points de l'architecture. Généralement l'optimisation se situe soit au niveau d'un serveur simple (optimisation à grain fin), soit au niveau architectural (allocation de ressources).

Valeur à atteindre. Suivant le type de système (serveur simple ou multi-niveaux), la valeur à atteindre concerne soit le MPL, soit la valeur en sortie du système/modèle (par exemple la latence ou le débit). Ce critère définit si elle est donnée (par l'administrateur ou le SLA), ou alors calculée pour atteindre l'optimal (auto-optimisation) ? Il est en effet souvent plus facile de satisfaire une valeur fixée que de trouver la meilleure configuration dans un environnement donné. Dans la réalité, trouver l'optimal n'est pas toujours utile, car la majorité des contraintes de temps de services que doivent respecter les fournisseurs de services Internet sont données dans des SLA, qui définissent généralement des bornes maximales pour le temps d'attente des clients. Il suffit donc de trouver une configuration répondant à ces critères, et non pas la configuration optimale. Cependant dans une optique de limitation des coûts, trouver la configuration optimale permet d'allouer exactement le nombre de serveurs requis et peut donc se réveler intéressante.

Variation de charge. L'étude prend-elle en compte la variation de charge en entrée du système/modèle ? Dans le cas d'une charge qui ne varie pas, le temps mis par le système à se stabiliser (*settling time*) n'est pas aussi primordial qu'avec de la variation de charge. L'étude sera aussi moins réaliste, car les systèmes réels sont soumis à des charges fluctuant de manière importante au cours du temps.

Hétérogénéité de la charge. Indique si le type de charge (proportion de lectures et d'écritures dans les requêtes) varie au cours du temps. Le type de charge peut également être simplement homogène (lecture ou écritures seules) ou alors hétérogène (lectures et écritures distribuées selon des lois réalistes par exemple). Le comportement d'un serveur peut varier de manière radicale suivant si la charge subie est composée principalement de lectures ou de mises à jour. Construire des solutions prenant en compte la variation du type de charge est donc importante pour obtenir de bonnes performances.

Généricité. L'approche présentée est-elle générique, c'est à dire peut-on la réutiliser facilement dans un autre contexte et/ou avec un autre système ? En effet certaines approches, notamment de modélisation, sont spécifiques à un système donné, prenant en compte explicitement certaines particularité du système et rendant donc l'approche peu ou difficilement utilisable dans un contexte différent.

2.3 Approches basées sur une heuristique

Nous allons présenter dans cette section une sélection de différents articles dans les domaines de la modélisation et de l'auto-configuration de services Internet. Les articles présentés

dans cette section suivent globalement tous une approche réactive. Cela signifie que Ils sont en classés selon les différents critères présentés en 2.2.

2.3.1 Taton et. al.

Présentation. Le but de l'approche présentée ici est d'ajuster automatiquement le nombre de duplicatas d'un serveur en fonction de la charge [3]. A chaque état, une boucle de commande surveille l'utilisation du processeur sur les duplicatas, et si ce taux d'utilisation dépasse un certain seuil, un réplica est ajouté. Inversement si le taux d'utilisation devient trop faible, on enleve un serveur pour libérer des ressources. L'implémentation a été réalisée dans le cadre du prototype Jade [3, 2].

Classification. Cette approche système est mise en oeuvre dans le cadre d'une application multi-étages (banc d'essai Rubis). L'approche vise à obtenir les meilleures performances possible (valeur à atteindre auto-calculée). Durant les expériences la charge varie en quantité, mais la nature de cette charge reste constante. Cette approche est générique car elle propose une approche par boîte noire de l'application administrée. L'allocation de ressource est basée sur une heuristique, en effet on ajoute ou retire des ressources de manière incrémentale.

Limites. Cette approche donne de bons résultats, mais la reconfiguration par pas successifs induit des délais de réaction assez importants, ce qui peut être génant si la charge évolue rapidement. De plus seule une optimisation au niveau du degré de duplication sur les étages est effectuée. Le taux de parallélisme maximal des serveurs n'est ainsi pas modifié.

2.3.2 Heiss et. al.

Présentation. Cet article, souvent cité dans d'autres papiers beaucoup plus récents est la base des techniques d'auto-optimisation au niveau local (serveur simple) [9, 17, 23]. En effet les auteurs montrent que pour une charge donnée, les performances d'un serveur multiprogrammé (serveur Web, SGBD) varient en fonction du taux de parallélisme maximum (MPL[1]) sous la forme d'une parabole avec un optimum global (voir figure 3.2).

De plus cet optimum varie au cours du temps, en fonction du type de charge et de la quantité de requêtes en entrée sur le système. Un tel comportement est présenté à la figure 2.2.

Dans cet article sont décrites deux approches (par incréments et par approximantion parabolique) visant à calculer le taux de parallélisme donnant les meilleurs performances pour une charge donnée. Ainsi, l'optimisation au niveau local (dite aussi *à grain fin*) a pour but d'adapter le MPL en fonction de la charge.

Classification. Cet article propose deux approches, l'une étant basée sur une heuristique (méthode par incréments) et l'autre modèle (car ayant un modèle du comportement d'un serveur). Le cadre applicatif est un simple serveur transactionnel. L'algorithme calcule une valeur optimale en fonction du type de charge et cherche à l'atteindre. Les expériences ont été effectuées sur un workload homogène. L'approche est générique car le modèle de comportement utilisé fonctionne pour l'ensemble des serveurs multiprogrammés.

[1] MultiProgrammation Level

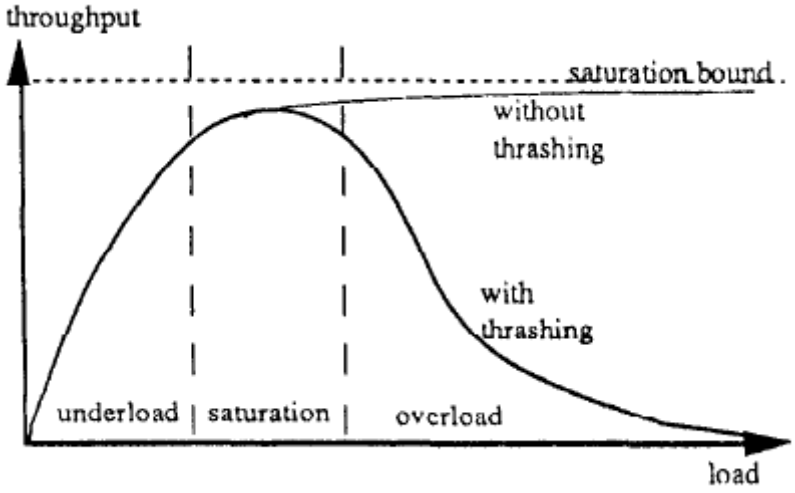

FIG. 2.1 – Débit en fonction de la charge [9]

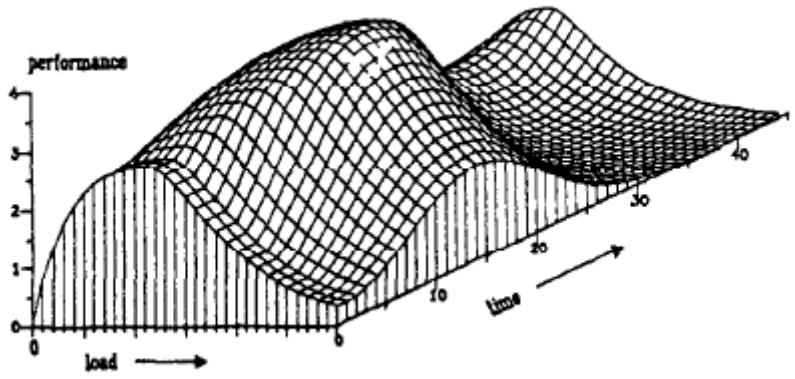

FIG. 2.2 – Comportement dynamique d'un serveur multiprogrammé [9]

Limites. Le paramétrage des algorithmes, qui est très important pour obtenir de bons resultats, reste empirique. De plus la type de charge étudié est extrèmement simple et correspond rarement au type de charge que l'on peut trouver sur un serveur réel.

2.3.3 Boucle de commande MIMO

Présentation. On s'intéresse souvent à l'optimisation d'une métrique (debit du serveur, latence moyenne des requêtes...), mais si l'on veut optimiser plusieurs parametres en meme temps il faut une approche globale et non une juxtaposition d'optimisations indépendantes, comme le montre la figure 2.3 [7]. En effet les différentes approches d'optimisation peuvent ne pas être totalement orthogonales. Par exemple optimiser un premier critère peut nuire à un second. Les auteurs présentent ici une méthode utilisant plusieurs paramètres de configuration du système, pour optimiser l'utilisation de plusieurs ressources matérielles. L'idée que la juxtaposition d'optimisations au même niveau est moins efficace qu'une optimisation globale de plus haut niveau semble se confirmer ici.

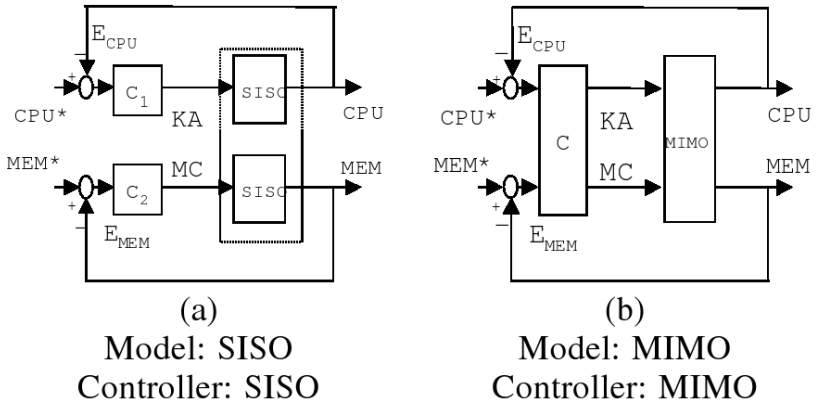

FIG. 2.3 – Comparaison des approches SISO/MIMO [7]

Classification. L'approche présentée modélise le système administré pour satisfaire des critères de performance donnés par l'administrateur (utilisation du processeur et de la mémoire). Le cadre applicatif est un serveur simple Apache, et les expériences sont effectuées sans variation de charge et avec un type de charge hétérogène. Comme la modélisation est spécifique à Apache, le controleur n'est pas générique, cependant l'approche en elle-même peut être considérée comme telle.

2.3.4 Middle-R

Présentation. Cet article présente une approche intéressante d'optimisation à 2 niveaux [17]. En effet il utilise la méthode d'approximation parabolique vue dans [9] pour optimiser le temps de réponse au niveau local (serveur), et quand cela ne suffit plus, une reconfiguration à un niveau plus global (niveau) est effectuée pour répondre à la demande.

Classification. Cet article présente une approche d'optimisation à deux niveaux, à la fois à grain fin et au niveau architectural. L'optimisation au niveau architectural est cependant

limitée, car elle se limite à faire de la distribution de charge, et non pas de l'allocation de ressources. Le cadre applicatif est une base de données répliquée (PostGreSQL). La complexité architecturale est semi-complexe, car on gère l'optimisation sur deux axes (niveau de duplication et taux de parallélisme maximal, voir figure 2.4) mais pas sur le troisième (étages de l'architecture, voir figure 2.5). Les expériences sont menées avec une charge variante et de type hétérogène. L'approche est générique car Middle-R est un intergiciel indépendant du système administré.

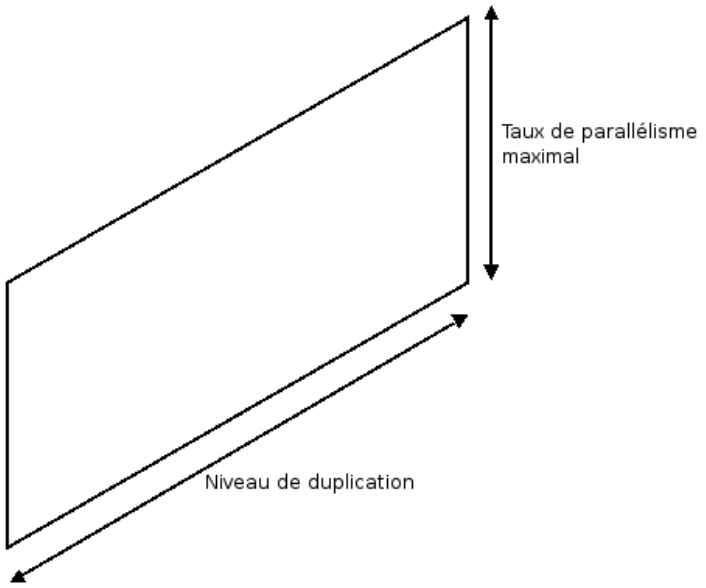

FIG. 2.4 – Optimisation sur 2 axes : niveau de duplication et taux de parallélisme des serveurs pour un étage unique

Limites. Néammoins, cette approche n'est peut-être pas viable avec une charge plus complexe collant moins au modèle de Heiss et. al. [9], car ces derniers se placaient dans le cas d'un simple serveur Web, avec une charge homogène constante. De plus, l'approche conserve les limites du modèle de [9], à savoir principalement le problème du paramétrage de l'algorithme. Au niveau global, le nombre de réplicas n'est pas modifié, mais le mapping entre objet accédé par transaction et réplica gérant la transaction évolue au cours du temps pour équilibrer la charge entre les noeuds. Les gains sont important avec une charge inégalement répartie sur les noeuds De plus, il faudrait que le nombre de réplicas puissent évoluer (dynamic provisioning), car si la charge augmente trop, le nombre de réplicas risque d'être insuffisant garantir les performances voulues. Reste enfin le problème du paramétrage de l'auto-optimisation (seuils et intervalle de mesure)

2.4 Approches basées sur un modèle

Les approches présentées dans cette section utilisent un modèle du système administré pour pouvoir calculer ses performances dans un environnement donné.

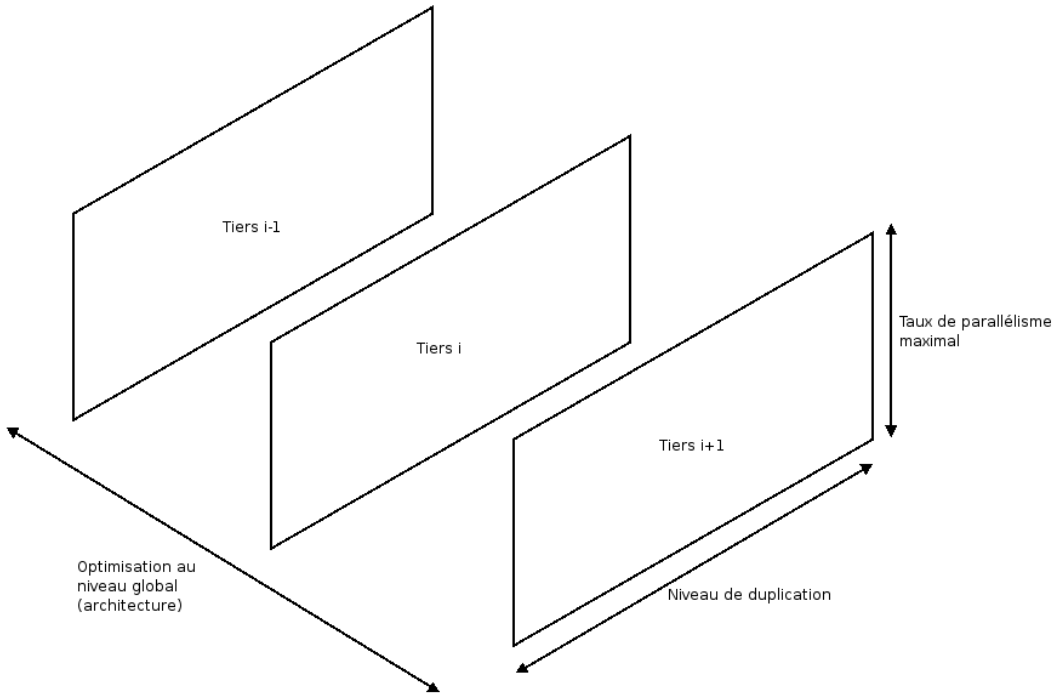

FIG. 2.5 – Optimisation sur 3 axes : niveau de duplication sur chaque étage, analyse globale (sur l'ensemble des étages de l'architecture), et taux de parallélisme des serveurs

2.4.1 Gatekeeper

Présentation. Cet article présente deux principes pour améliorer le temps de réponse d'un système multi-niveaux, particulièrement en cas de pic de charge [8]. Premièrement, le contrôle d'admission (*admission control*) sert à calculer un sous-ensemble optimal des requêtes en entrée du sytème autorisées à être exécutées, afin de limiter l'écroulement dû à un nombre trop important de requêtes dans le système. Deuxièmement, une méthode de réordonancement de requêtes (*request scheduling*) selon la technique du plus court d'abord (*SJF : shortest job first*) est utilisée pour diminuer de manière importante le temps de réponse moyen du système. Ceci est réalisé à l'aide d'un proxy, qui maintient pour chaque classe de requête le temps moyen d'exécution, ce qui permet de prédire quelles seront les requêtes les plus rapides à se terminer.

Classification. Cet article est classé modèle, car il considère le serveur comme une capacité de traitement disponible, à laquelle sont affectées des requêtes dont on connait la durée d'exécution moyenne. Le cadre applicatif est un site de commerce en ligne, mais limité à une interraction client/serveur entre deux niveaux. Le nombre de requêtes admissibles dans le système est continuellement calculé et mis à jour en fonction du temps d'exécution de ces requêtes. Le type de charge est hétérogène car on considère plusieurs classes de requêtes avec des temps de traitement très différents, et avec des charges pouvant varier. Cette approche est générique car elle considère le système comme une boîte noire.

Limites. L'estimation du temps d'exécution des requêtes de chaque classe se faisant à faible charge, il n'est pas garanti que ces temps restent valides lorsque la charge varie de manière

brutale. La méthode de contrôle d'admission qui se base sur ces temps d'exécution risque donc de se réveler approximative voire fausse si les délais fluctuent dans des proportions trop importantes.

2.4.2 Parekh et. al.

Présentation. Les auteurs de cet article sont des spécialistes dans le domaine de la théorie de la commande, et ils définissent ici un vocabulaire jusque là utilisé de manière informelle pour parler de modélisation de systèmes à base de boucle de rétro-action (voir figure 2.6) [19].

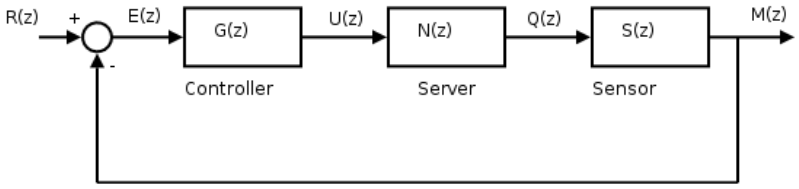

FIG. 2.6 – Boucle de rétro-action (inspiré de [19])

Cet article présente une méthode mathématique pour trouver le gain maximal utilisable pour un certain type de serveur lors d'une reconfiguration (pour minimiser les oscillations et le temps de stabilisation *settling time*) Il donne également une marche à suivre pour la modélisation mathématique d'un serveur en tant qu'une fonction de transfert.

Classification. Cette approche vise à modéliser un serveur Internet pour déterminer les paramètres de la boucle de commande d'auto-optimisation associée. Même si on dispose d'un modèle de l'application, le type d'approche est donc bien basée sur un palier de reconfiguration. L'approche est appliquée sur le cas d'un serveur simple (Lotus Note). Le type de charge est homogène, et il n'y a pas de variation de charger, le but de l'étude étant entre autres de comparer les temps de stabilisation du MPL à l'optimal.

Limites. L'approche formelle est intéressante, mais est limitée à des systèmes très simples. En effet les systèmes administrés sont des serveurs simples, avec un workload homogène et une charge qui ne varie pas.

2.4.3 Urgaonkar et. al.

Présentation. Cet article présente une modélisation globale d'un système multi-niveaux à base de réseau de files d'attente, dont un exemple est présenté sur la figure 2.7 [24].

L'approche est basée sur l'utilisation de l'algorithme MVA (*Mean Value Analysis*). Le point difficile est de calculer les données en entrée de l'algorithme. Par contre cette approche est puissante car elle permet de gérer de manière globale les particularités des applications modélisées. Les applications présentées sont le provisioning dynamique (ajout/retrait de serveurs en fonction de la charge) et la différentiation par classes pour le contrôle d'admission.

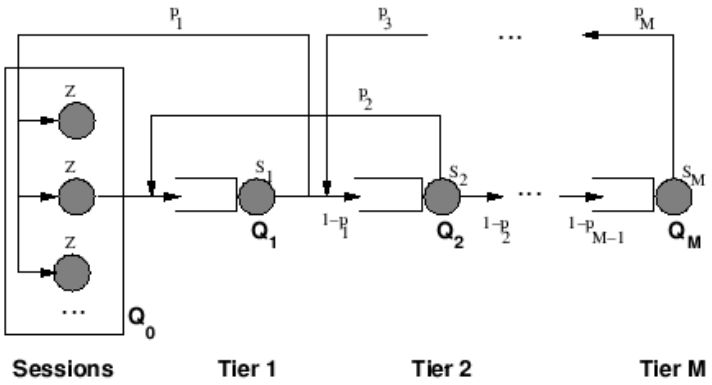

FIG. 2.7 – Exemple de réseau de files d'attente [24]

Classification. Cet article est axé modèle, mais comporte une implémentation dans un système d'allocation de ressource. Le cadre applicatif est un site de commerce en ligne multi-étages, avec donc une architecture complexe (étages potentiellement répliqués). Le but du modèle est de permettre de garantir des critères de performance donnés par l'administrateur (dans ce cas une latence à ne pas dépasser). L'approche permet de prendre en compte des charges variables, et de type hétérogène. De plus ce modèle est assez générique, car on peut le calibrer pour n'importe quel système en recalibrant ses paramètres.

Limites. Dans cette approche, seul le CPU est considéré comme étant la ressource bottleneck. Bien que cette hypothèse soit vraie dans la plupart des cas, il peut arriver que parfois ce soit la mémoire, les accès disque ou bien le réseau qui constitue le goulot d'étranglement des performances. On pourrait envisager d'étendre l'approche présentée ici en gérant donc des bottlenecks de plusieurs natures.

2.4.4 Allocation de ressources sous SLA

Présentation. Cet article présente une méthode de modélisation d'un ensemble de services formant une application multi-niveaux (voir figure 2.8) [22]. La composition de service est un cadre plus général que les architectures multi-niveaux. En effet dans le cas des architectures n-niveaux, un niveau peut interroger au plus le niveau suivant, alors que dans le cadre de la composition de services un niveau peut en interroger plusieurs autres, éventuellement distants.

L'objectif de cette étude est de faire du provisionning de manière efficace, sous la contrainte de SLA. Le critère de performance est uniquement le temps de réponse. La vision globale (top-down plutot que bottom-up) de l'optimisation permet de rajouter des resources (caches, repliques) à des emplacements non triviaux dans l'architecture. Ceci permet d'avoir des temps de réponse optimaux tout en minimisant l'utilisation de ressources. Une méthode pour évaluer le hit cache ratio est présentée, utilisant un système de caches virtuels (fonctionne sur le principe de "pêche à la ligne")

Classification. Le but de cet article est de permettre de faire de l'allocation de ressource efficace en ayant un modèle du comportement de l'application. C'est donc une approche axée modèle. Le cadre applicatif est complexe (multi niveaux, composition de services). La valeur à

atteindre est une latence maximale, donnée par un SLA. La charge, hétérogène, peut varier au cours du temps. Enfin l'approche est plutôt générique, même si le modèle doit être recalibré pour fonctionner sur un système différent.

Limites. Dans cette approche, on fait la supposition que le MPL de chaque serveur est fixe. On pourrait combiner cette approche avec les travaux présentés en [9] pour obtenir un système optimisé sur deux niveaux d'architecture.

2.4.5 Analyse de temps de réponse

Présentation. Cet article présente une méthode de modélisation d'un service à 2 niveaux (un front-end plus un niveau composé de N services, voir figure 2.8) afin de calculer de manière précise le temps de réponse moyen et de donner une approximation fiable de la variance de ce temps de réponse [1]. L'évaluation expérimentale se fait avec RUBoS et TPC-App. Les applications possibles présentées dans cet article sont le ressource provionning, l'admission control (avec et sans buffer), et la négociation de SLA.

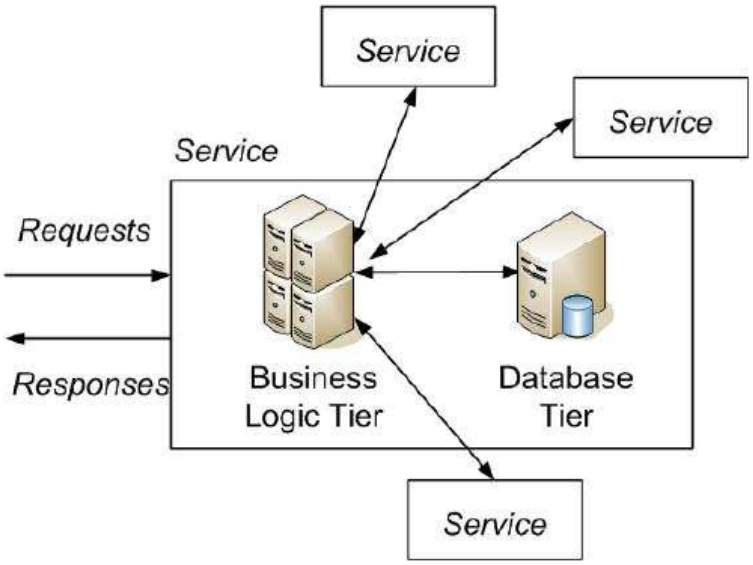

FIG. 2.8 – Application multi-niveaux basée sur la composition de services [1]

Cet article assez technique est intéressant car il est présente une méthode pour prédire la variance du temps de réponse. La variance fait rarement l'objet d'étude, alors qu'elle est souvent présente dans les SLA des fournisseurs de services.

Classification. Ce modèle permet d'estimer, outre temps de réponse du serveur, mais aussi sa variance. Le cadre applicatif est un site de commerce en ligne (architecture complexe). La charge varie, mais le type de charge pris en compte est homogène (lectures ou écritures seules).

Limites. Le cadre de l'évaluation expérimentale est très simple, alors que le modèle prend en compte des architectures évoluées. De plus la complexité des calculs ainsi que la difficulté d'obtention des paramètres en entrée du modèle rendent ce dernier peut simple d'utilisation.

2.5 Synthèse de l'état de l'art

Les premières recherches dans le domaine de l'auto-optimisation utilisaient des techniques *ad hoc*, basées sur une l'idée intuitive que l'on se faisait du comportement d'un système. Si cela pouvait donner quelques résultats sur des systèmes simples avec des charges facilement caractérisables, cette approche n'est plus viable avec des système de plus en plus complexes (multi-niveaux, répliqués) et des charges complexes. Le besoin de modèles mathématiques pour caractériser le comportement de tels systèmes s'est alors fait sentir, et de nombreux travaux ont eu pour but de proposer des modèles plus ou moins complexes pour différents systèmes. Cependant la très grande majorité des modèles se limitent à caractériser un seul serveur [9], et non pas la globalité d'un système multi-niveaux [19, 7, 8, 17]. Récemment sont apparus les premiers modèles ayant une approche de l'ensemble des niveaux du système [22, 24]. Ces modèles sont tous basés sur des réseaux de files d'attente, qui sont à la fois simples d'utilisation et suffisament puissants pour caractériser des systèmes complexes. Cependant, dans l'état actuel de nos connaissances, il n'existe pas d'approche ayant intégré à la fois l'optimisation à grain fin sur chaque serveur et l'optimisation globale de l'allocation de ressources au niveau architectural. Notre premier objectif est donc de fournir un modèle regroupant ces deux niveaux d'optimisation.

Cependant, si la construction d'un modèle est extrèmement utile, elle n'est pas une fin en soi. Ce qui manque aujourd'hui, c'est donc le retour au système. Il va falloir améliorer et simplifier les modèles actuels, pour ensuite utiliser ces derniers dans des solutions logicielles d'auto-configuration.

Les tableaux 2.9 et 2.10 récapitulent les différents articles vus en 2.3.

	Critères de comparaison			
Articles	**Calcul configuration**	**SISO / MIMO**	**Cadre applicatif**	**Complexité architecturale**
Taton et. al. [3]	Heuristique	-	multi-niveaux repliqués	complexe
Heiss et Wagner [9]	Heuristique / Modèle	-	serveur transactionnel	simple
Parekh et. al. [19]	Heuristique	-	serveur simple (Lotus Notes)	simple
Boucle de commande MIMO [7]	Modèle	MIMO	serveur simple (Apache)	simple
Gatekeeper [8]	Modèle	SISO	multi-niveaux repliqués	simple (niveau unique)
Middle-R [17]	Modèle	SISO	SGBD répliqué	simple (niveau unique)
Urgaonkar et. al. [24]	Modèle	SISO	multi-niveaux	complexe
Allocation de ressources sous SLA [22]	Modèle	MISO	composition de services	complexe
Analyse de temps de réponse [1]	Modèle	MIMO	composition de services	complexe

FIG. 2.9 – Tableau de synthèse de l'état de l'art (1/2)

	Critères de comparaison			
Articles	**Valeur(s) à atteindre**	**Variation de charge**	**Workload**	**Généricité**
Taton et. al. [3]	auto-calculée	oui	homogène	oui
Heiss et Wagner [9]	auto-calculée	oui	homogène	oui
Parekh et. al. [19]	donnée par l'administrateur	non	homogène	oui
Boucle de commande MIMO [7]	données par l'administrateur	non	hétérogène	non
Gatekeeper [8]	auto-calculée	oui	hétérogène	oui
Middle-R [17]	auto-calculée	non	hétérogène	oui
Urgaonkar et. al. [24]	donnée par l'administrateur	oui	hétérogène	oui
Allocation de ressources sous SLA [22]	données par le SLA	oui	hétérogène	oui
Analyse de temps de réponse [1]	auto-calculées	oui	simple[2]	non

FIG. 2.10 – Tableau de synthèse de l'état de l'art (2/2)

Chapitre 3

Conception du gestionnaire de ressources

Dans ce chapitre nous présenterons une approche basée sur modèle pour la gestion de ressources dans le cadre de systèmes multi-niveaux. Nous commencerons par décrire les principes généraux de l'approche, puis nous présenterons les phases de modélisation au niveaux local et architectural de l'application, et enfin nous présenterons un algorithme de configuration dynamique pour optimiser la gestion de ressources pour les services Internet.

3.1 Approche globale de gestion de ressource

Cette section présente la méthode retenue pour effectuer l'auto-optimisation d'un service Internet à la fois aux niveaux local et global.

3.1.1 Schéma général

La figure 3.1 schématise le fonctionnement de l'algorithme d'auto-optimisation qui sera appelée périodiquement. Lorsque la charge augmente, si les performances ne respectent plus des contraintes de qualité de service (généralement fixées par un SLA[1]) Nous allons donc chercher à optimiser les performances des serveurs au niveau local. En effet même si une optimisation de ce type a déjà eu lieu précédement, l'augmentation de la charge peut avoir également induit une variation dans la proportion de lectures / écritures dans les requêtes sur la base de données, et ainsi l'optimal peut s'être déplacé. Si l'optimisation au niveau local ne suffit pas, nous effectuons alors une optimisation au niveau architectural. Notre modèle au niveau architecture permettra de savoir combien de noeuds devront être ajoutés, ainsi que leur emplacement. Ainsi nous ajoutons des noeuds seulement en dernier recours pour minimiser l'utilisation des ressources matérielles.

Pour éviter de gespiller des ressources si la charge diminue, l'optimisation au niveau architectural est apellée périodiquement pour vérifier que des ressources ne sont pas allouées sans raison. Si la nouvelle configuration calculée par le modèle est différente de la configuration actuelle, alors les changements sont appliqués.

[1] *Service Level Agreement*, définit la qualité de service (par exemple latence maximale autorisée) devant être fournie par le prestataire de service

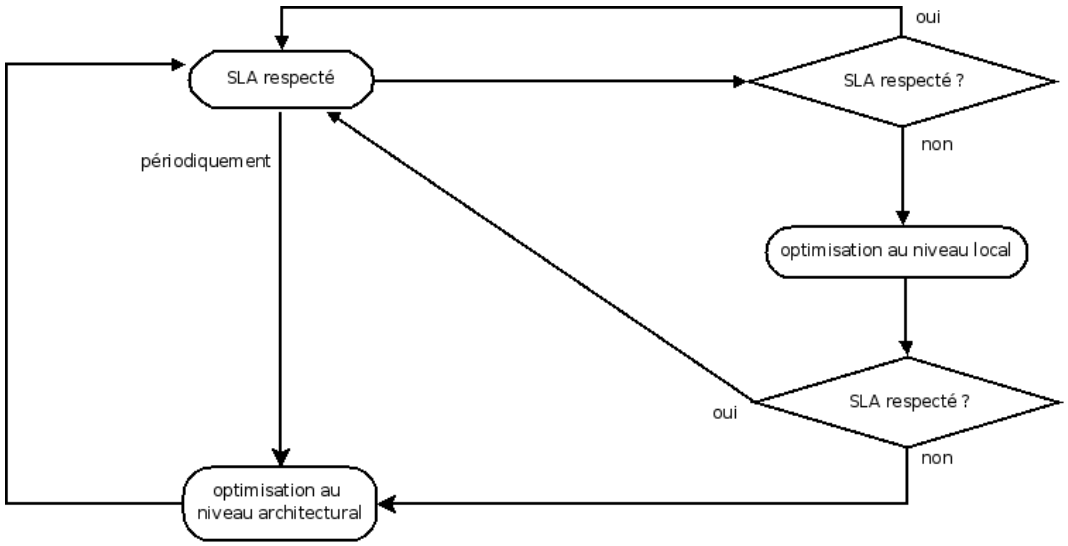

FIG. 3.1 – Principe de l'algorithme

Au niveau d'un étage (serveur simple), nous adapterons le taux de parallélisme maximal pour atteindre les meilleures performances délivrables par ce serveur. L'optimisation de services Internet commençant à être assez bien étudiée, nous nous baserons pour cela sur différents travaux comme [9] ou [23]. Nous cherche ainsi à tirer le maximum d'un serveur avant d'en ajouter un autre.

Deuxièmement, quand l'optimisation au niveau d'un serveur ne suffit plus, une optimisation cette fois-ci globale sera lancée. Elle aura pour but d'ajouter un ou plusieurs serveurs à l'endroit optimum de l'architecture, ce qui est parfois un problème non-trivial. Nous utiliserons pour cela un modèle à base de files d'attente, qui est actuellement la méthode donnant les meilleurs résultats [22, 24].

Idéalement, il faudrait prendre en compte à la fois le taux d'arrivée des requêtes et le type de charge, ceci afin d'implémenter des mécanismes de réordonancement et de différentiation de classes de session qui permettent d'améliorer grandement les performances du système [8].

La modélisation de notre système devra être effectuée à deux niveaux. En effet nous voulons à la fois optimiser la configuration des serveurs au niveau local (serveur simple), mais aussi améliorer l'affectation de ressources au niveau architectural.

3.2 Modélisation de services Internet multi-niveaux

3.2.1 Modélisation au niveau local

La première étape du travail est donc d'obtenir un modèle simple et fiable du fonctionnement d'un serveur unique.

Fonction de performance.

Ce modèle devra être capable de fournir une estimation de la configuration du serveur qui donnera les meilleures performances, en fonction de paramètres connus de l'état du système. Ces données peuvent concerner les ressources matérielles (par exemple l'utilisation du CPU,

de la mémoire ou du disque) ou caractériser le type de charge appliquée au serveur. En effet nous avons vu que le comportement d'un serveur de bases de données était différent suivant si les requêtes en entrée comportaient en majorité des lectures ou des écritures.

Nous pouvons définir plusieurs critères de performance pour un serveur. Cela peut être un taux d'utilisation du processeur ou de la mémoire le plus proche possible de 95%. Cela peut également être de minimiser la latence et de maximiser le débit. Dans le cadre des services Internet, le critère de performance le plus significatif est ce dernier [11]. Nous voulons en effet un débit maximum pour le serveur, tout en minimisant les temps d'attente (latence) chez les clients.

La fonction de performances retenu pour un serveur simple sera donc le ratio du débit sur la latence en fonction de la charge (en nombre de clients) :

$$fperf_{\text{local}}(\text{charge}) = \frac{\text{débit}}{\text{latence}} \tag{3.1}$$

Nous présentons dans cette section le modèle de serveur utilisé. Ce modèle sera capable de fournir en fonction de la charge et du taux de multiprogrammation maximal une estimation des performances du serveur administré.

Approche retenue.

Au niveau local, nous utiliserons la modélisation proposée dans [9], qui a été reprise avec succès dans le prototype Middle-R [17]. Cette approche par approximation parabolique est illustrée sur la figure 3.2. Elle consiste à trouver les paramètres de la parabole se rapprochant le plus du comportement du serveur.

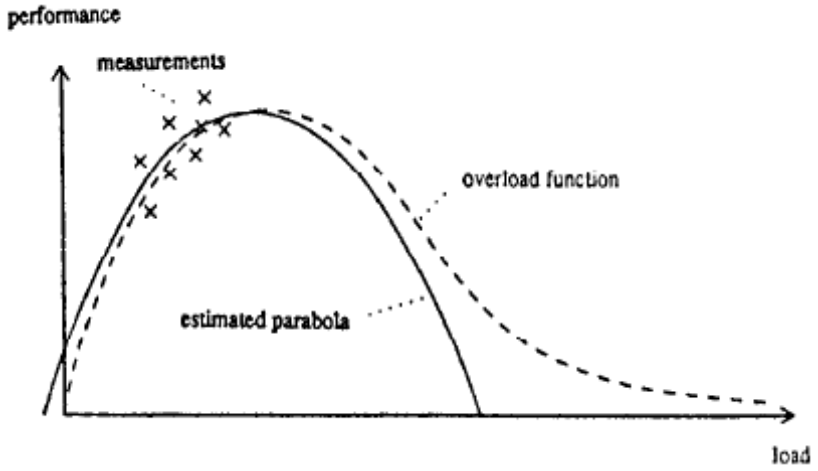

FIG. 3.2 – Approximation parabolique [9]

La formule 3.2 est l'équation standard d'une parabole.

$$y = ax^2 + bx + c \tag{3.2}$$

Comme nous cherchons à représenter les performances du système en fonction de sa charge, nous pouvons écrire :

$$\text{performances} = a \times \text{charge}^2 + b \times \text{charge} + c \tag{3.3}$$

Notre but va donc être de trouver les valeurs des paramètres a, b et c. A noter que ces paramètres peuvent changer en fonction de l'environnement, typiquement le ratio écritures/-lectures *(workload mix)*.

Calibration.

Pour calibrer notre modèle, nous allons faire fonctionner le serveur instrumenté pour récupérer des informations sur les performances du système, en l'occurence le débit du serveur de base de données, en fonction de la charge (nombre de clients). Une fois les données obtenues, nous utilisons la méthode des moindres carrés pour trouver les paramètres a, b et c qui minimiseront l'écart moyen entre la parabole et les données mesurées. La partie 4.2 présente la phase de calibrage et de validation de ce modèle dans notre cadre applicatif.

3.2.2 Modélisation au niveau architectural

Dans un deuxième temps, il faudra construire un autre modèle au niveau global du système administré.

Fonction de performance.

Ce modèle sera capable de prévoir pour une configuration donnée (nombre d'étages, degré de duplication à chaque étage, nombre de clients et temps de traitement à chaque étage) le temps de réponse moyen de l'application. La fonction de performance associée exprimera la performance du système global en fonction de la configuration du système. Cette configuration est représenté par un M-uplet (M étant le nombre de niveaux de l'application), dont chaque élément sera le degré de duplication à un étage.

La fonction de performance sera néammoins différente suivant le contexte. En effet si le SLA n'est pas respecté la fonction de performance dépendra de la latence du système. Par contre si le SLA est respecté, nous chercherons à optimiser le nombre de ressources matérielles. La formule 3.4 présente la fonction de performances au niveau global.

$$fperf_{\text{global}}([r_1 \dots r_M]) = \begin{cases} \frac{1}{\text{latence}} \text{ si latence} > \text{latence}_{SLA} \\ \\ \frac{1}{NB\text{ressources}} \text{ sinon} \end{cases} \tag{3.4}$$

Ceci nous permettra de déterminer à quel endroit de l'architecture l'allocation de ressources donnera les meilleurs résultats. A ce niveau, une ressource consistera en un duplicata d'un serveur au niveau d'un étage. Une approche de la sorte, ayant une vision globale de l'application, a récemment donné de bons résultats, comme présenté dans [22].

Après avoir décrit la modélisation des performances d'un serveur en 3.2.1, nous présentons dans cette le modèle choisi pour représenter l'architecture de l'application multi-étages administrée.

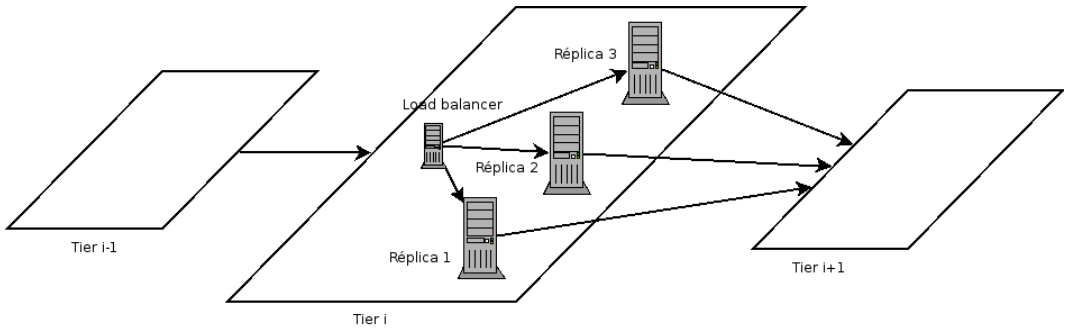

FIG. 3.3 – Duplication au niveau d'un étage

Approche retenue.

La modélisation du système au niveau architectural suivra l'approche proposée par Urgaonkar et. al. dans [24]. Cette approche utilise un réseau de files d'attente pour modéliser l'application (voir figure 3.4). Dans ce modèle, chaque file représente un serveur de l'application, et les requêtes passent d'une file à l'autre suivant des lois probabilistes.

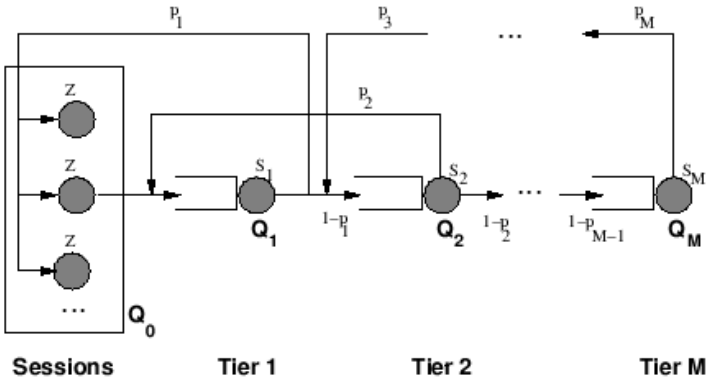

FIG. 3.4 – Exemple de réseau de files d'attente [24]

L'algorithme MVA[2] est utilisé pour calculer le temps de réponse moyen pour l'utilisateur final. Comme indiqué en 2.4.3, le paramétrage de l'algorithme MVA nécessite d'instrumenter le système à modéliser de manière assez précise. En effet il faut fournir des informations sur les temps de traitement minimum des requêtes, ce qui peut être obtenu en mesurant les temps de réponse des différents étages du système lorsque ce dernier n'est pas saturé.

Une fois le modèle correctement paramétré, nous serons capable de prédire le temps de réponse de l'application en fonction de sa configuration. Nous chercherons alors quelle est la configuration donnant le meilleur ratio performances/coût.

Algorithme MVA.

L'algorithme MVA (algorithme 1) a été proposé en 1980 par Reiser et Lavenberg [20]. Il permet de calculer le temps d'exécution moyen (latence) d'une requête dans un réseau de files

[2] *Mean Value Analysis*

d'attente.

Les paramètres en entrée de l'algorithme sont les suivants :

N **: Nombre de clients.** Le nombre de sessions clientes accédant à l'application depuis l'étage 0.

$\bar{S}_m$ **: Latence moyenne induite à chaque étage (ou** *service time*). A chaque étage de l'application, les requêtes sont retardées d'une durée correspondant au temps de service de la requête à cet étage. Ce temps de service dépend entre autres du temps processeur requis et des accès au disque ou à la mémoire.

V_m **: ratio de visite à l'étage** m **(ou** *visit ratio*). Le ratio de visite d'un étage d'une application multi-niveaux est le nombre moyen de fois que cet étage est appelé pendant l'exécution d'une requête. Ainsi d'un point de vue statistique, si on note λ_{req} le nombre total de requêtes traitées par l'application pendant une certaine durée t, et λ_i le nombre de requêtes traitées par l'étage i durant la période t, alors le ratio de visite pour l'étage i peut être calculé :

$$V_i = \frac{\lambda_i}{\lambda_{req}} \tag{3.5}$$

$\bar{Z}$ **: temps d'attente moyen (ou** *think time*). Lorsque un client reçoit la réponse à sa requête, il s'écoule un temps variable avant qu'il n'envoie une autre requête vers le système. Nous devons fournir la moyenne de ces temps d'attentes à l'algorithme pour estimer de quelle manière les clients vont charger le système.

La table 3.5 donne la signification des différentes variables de l'algorithme 1.

Symbole	**Signification**
M	Nombre d'étages dans l'application
N	Nombre de sessions clientes
Q_m	File d'attente à l'étage $T_m (1 \leq m \leq M)$
Q_0	File de stockage des sessions clientes
$\bar{Z}$	Temps d'attente chez le client entre la réception d'une réponse et l'envoie de la requête suivante (think time)
$\bar{S}_m$	Latence moyenne induite par l'étage m
$\bar{L}_m$	Longueur de la file d'attente à l'étage m
τ	Débit du système
$\bar{R}_m$	Temps de réponse moyen à l'étage m
$\bar{R}$	Temps de réponse moyen de l'application
$\bar{D}_m$	Demande moyenne à l'étage m
V_m	Ratio de visite pour l'étage m

FIG. 3.5 – Notations utilisées dans l'algorithme MVA

Algorithme 1 : Algorithme MVA

Entrées : N, $\bar{S}_m$, V_m, $1 \leq m \leq M$, $\bar{Z}$
Sorties : $\bar{R}_m$ (latence moyenne sur la file Q_m), $\bar{R}$ (temps de réponse moyen)

début
  $\bar{R}_0 = \bar{D}_0 = \bar{Z}$
  $\bar{L}_0 = 0$
  pour $m = 1$ *à* M **faire**
    $\bar{L}_m = 0$
    $\bar{D}_m = V_m.\bar{S}_m$ /* *demande de service* */
  /* *on insère les N clients un par un* */
  pour $n = 1$ *à* N **faire**
    pour $m = 1$ *à* M **faire**
      $\bar{R}_m = \bar{D}_m.(1 + \bar{L}_m)$ /* *latence moyenne* */
    $\tau = (\frac{n}{\bar{R}_0 + \sum_{m=1}^{M} \bar{R}_m})$ /* *débit* */
    pour $m = 1$ *à* M **faire**
      $\bar{L}_m = \tau.\bar{R}_m$ /* *loi de Little* */
    $\bar{L}_0 = \tau.\bar{R}_0$
  $\bar{R} = \sum_{m=1}^{M} \bar{R}_m$ /* *temps de réponse moyen* */
fin

Prise en compte de la duplication.

L'algorithme 1 présente une version de l'algorithme MVA qui considère un système multi-étages de manière générale, sans faire d'hypothèse sur le degré de duplication au niveau de chacun de ses étages. Dans notre cadre applicatif, nous souhaitons disposer d'un modèle fournissant le temps de réponse d'une application en fonction de sa configuration, notamment le degré de duplication au niveau de chaque étage.

Pour cela nous allons faire l'hypothèse que les requêtes seront équitablement distribuées entre chacune des répliques du serveur par un répartiteur de charge.

Ainsi si on note r_i le nombre de répliques d'un serveur à l'étage i, nous allons remplacer dans notre modèle la file Q_i par r_i files d'attentes $Q_{i,1}, ..., Q_{i,r_i}$. Le ratio de visite de chacune des répliques de cet étage sera donc :

$$V_{i,j} = \frac{V_i}{r_i} \tag{3.6}$$

En faisant la supposition de l'utilisation d'un répartiteur de charge équitable, le ratio de visite diminue donc proportionellement en fonction du nombre de répliques ajoutées. Ajouter plus de répliques d'un serveur ne permet pas de réduire le temps de service (S_m) qui est incompressible, mais autorise à traiter plus de requêtes en parallèle ce qui augmente donc le débit et diminue parallèlement la latence.

Nous proposons donc d'utiliser l'algorithme 2 pour prendre en compte la duplication des serveurs au niveau des étages de l'application.

Algorithme 2 : Algorithme MVA avec duplication

Entrées : N, $\bar{S}_m$, V_m, r_m, $1 \leq m \leq M$, $\bar{Z}$
Sorties : $\bar{R}_m$ (latence moyenne sur la file Q_m), $\bar{R}$ (temps de réponse moyen)

début
  $\bar{R}_0 = \bar{D}_0 = \bar{Z}$
  $\bar{L}_0 = 0$
  pour $m = 1$ *à* M **faire**
    $\bar{L}_m = 0$
    $\bar{D}_m = \frac{V_m.\bar{S}_m}{r_m}$ */* demande de service */*
  / on insère les N clients un par un */*
  pour $n = 1$ *à* N **faire**
    pour $m = 1$ *à* M **faire**
      $\bar{R}_m = \bar{D}_m.(1 + \bar{L}_m)$ */* latence moyenne */*
    $\tau = (\frac{n}{\bar{R}_0 + \sum_{m=1}^{M} \bar{R}_m})$ */* débit */*
    pour $m = 1$ *à* M **faire**
      $\bar{L}_m = \tau.\bar{R}_m$ */* loi de Little */*
    $\bar{L}_0 = \tau.\bar{R}_0$
  $\bar{R} = \sum_{m=1}^{M} \bar{R}_m$ */* temps de réponse moyen */*
fin

Symbole	Signification
r_m	Degré de duplication (nombre de serveurs en parallèle) à l'étage m de l'application

FIG. 3.6 – Notations supplémentaires pour la duplication

3.2.3 Intégration des deux modèles

Comme notre but est d'optimiser conjointement l'application sur les deux niveaux de l'architecture (voir figure 3.7), les fonctions de performance 3.1 et 3.4 vont être utilisées simultanément et de manière synchronisée par notre solution d'auto-optimisation. Le chapitre 3 présente la phase de conception de cette solution.

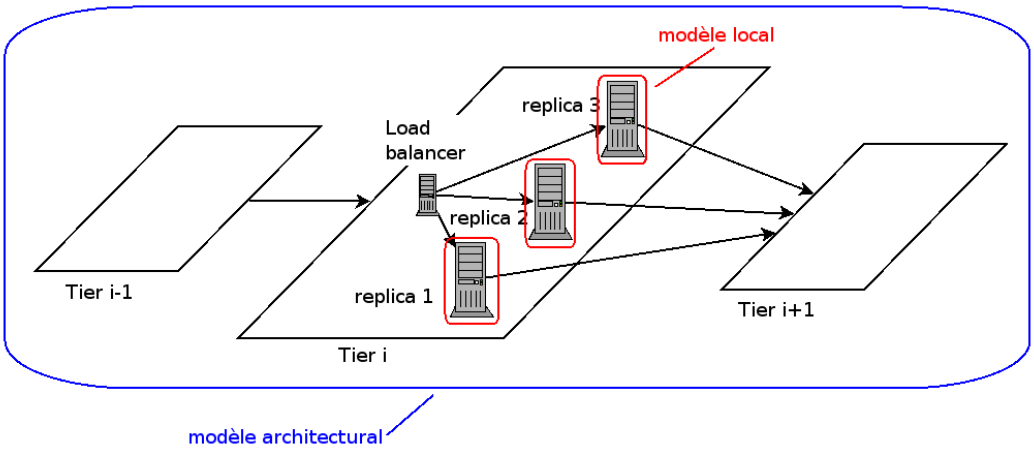

FIG. 3.7 – Niveaux de modélisation

3.3 Algorithme de configuration dynamique

Nous présentons dans cette section l'utilisation des modèles pour l'optimisation de gestion de ressources dans les systèmes multi-niveaux. Toutes les informations nécessaires au modèle (temps de service, débit, ...) sont mesurée en continu pendant le fonctionnement de l'application.

Respect du SLA

L'algorithme 3 utilise simultanément le modèle local et le modèle architectural pour optimiser l'allocation de ressources dans un système multi-niveaux.

La latence subie par les clients est donc mesurée en permanence ce qui permet de vérifier que le SLA est bien vérifié (dans le cas où le SLA porte sur la latence côté client). Si le SLA n'est plus vérifié, une optimisation au niveau local est lancée sur chaque serveur de l'application. Cette optimisation est effectuée de manière synchrone étage après étage pour éviter des phénomènes oscillations. En effet la reconfiguration d'un étage va avoir des effets sur la performance des étages voisins (débit accru par exemple), donc leur configuration optimale sera modifiée. Ainsi il faudra prendre en compte ces nouvelles données avant de reconfigurer les étages voisins, d'où l'application de la reconfiguration locale étage après étage.

Si cette optimisation locale ne suffit pasn le modèle architectural est alors utilisé pour trouver une configuration (en terme de nombre de serveurs au niveau de chaque étage) respectant le SLA et utilisant un minimum de ressources matérielles.

Nous faisons l'hypothèse de toujours disposer d'un nombre suffisant de machines en réserve pour pouvoir respecter le SLA. Dans certains cas réels le nombre de machines peut être

insuffisant, et nous aurons alors besoin d'autres techniques relatives à la qualité de service[3].

Economie de ressources

La charge appliquée sur le service Internet pouvant augmenter mais aussi diminuer, il faut une solution pour réduire le degré de réplication aux étages de l'application lorsque la charge diminue. Comme notre modèle au niveau architectural permet de trouver une configuration optimisée de l'application en terme de ressources, ce modèle est apellé périodiquement pour déterminer si des ressources peuvent être économisées.

Ce traitant périodique (algorithme 4) ne devra pas effectuer de reconfiguration si une autre optimisation (au niveau local ou architectural) due au non respect du SLA est en cours.

D'un point de vue technique, nous éviterons l'exécution simultanée de deux reconfigurations grâce à l'utilisation d'un verrou limitant l'accès à la variable okReconf.

Algorithme 3 : Traitant SLA non respecté

```
début
    si SLA non respecté alors
        //calcul nouveau MPL optimal
        MPL_calc = getOptimalMplFromModel() ;
        si |MPL_cur − MPL_calc| > mplTreshold et okReconf alors
            //si l'écart entre le MPL courant et celui calculé
            //est supérieur à une certaine limite, on reconfigure
            okReconf = false ;
            reconfMplServeur(MPL_calc) ;
            okReconf = true ;
        sinon
            //sinon, optimisation au niveau architecture
            config_calc = getOptimalArchitectureFromModel(SLA) ;
            si config_calc ≠ config_cur et okReconf alors
                okReconf = false ;
                setSystemConfig(config_calc) ;
                okReconf = true ;
fin
```

Algorithme 4 : Traitant optimisation périodique

```
début
    config_calc = getOptimalArchitectureFromModel(SLA) ;
    si config_calc ≠ config_cur et okReconf alors
        okReconf = false ;
        setSystemConfig(config_calc) ;
        okReconf = true ;
fin
```

[3] *QoS : quality of service*

Le fonctionnement de ces algorithmes est donc soumis à l'utilisation d'un modèle capable de calculer le MPL optimal pour un serveur multiprogrammé, ainsi que les performances d'un système multi-étages en fonction de la configuration de son architecture. Les sections 3.2.1 et 3.2.2 décrivent le processus de modélisation à ces deux niveaux d'architecture.

Utilisation des modèles dans l'algorithme d'optimisation

Les fonctions importantes dans les algorithmes sont *getOptimalMplFromModel()* et *getOptimalArchitectureFromModel()*. La première retourne à partir du modèle local le taux de parallélisme optimal pour le serveur administré, tandis que la seconde utilise le modèle architectural pour trouver une configuration (degré de duplication à chaque étage) qui respecte le SLA tout en minimisant les ressources utilisées. Les algorithmes 5 et 6 détaillent ces fonctions.

Algorithme 5 : getOptimalMplFromModel()

```
début
    MPL_nouveau = MPL_ancien
    //calcul des parametres de la parabole
    ApproximationParabolique() ;
    si a < 0 alors
        MPL_nouveau = -b/2a
    return MPL_nouveau ;
fin
```

L'algorithme 6, donnant le degré de duplication optimal à chaque étage, commence par chercher une borne supérieure au degré de duplication des étages, pour ensuite effectuer une exploration exhaustive de l'espace restant. La figure 3.8 schématise ce fonctionnement. Le degré de duplication optimal à chaque étage y est indiqué. L'algorithme commence par augmenter le degré de duplication simultanément à chaque étage jusqu'à ce que ce nombre soit suffisant pour que le SLA soit respecté. Ceci permet de trouver la borne supérieure que l'on retrouve sur la figure 3.8. Nous définissons plus formellement la borne supérieure B_{sup} comme suit :

Soit D_i l'ensemble des configurations (degrés de réplication) de l'étage i permettant de satisfaire le SLA. Alors :

$$B_{sup} = \max\left(\min_{i=1}^{M} r_i\right) \tag{3.7}$$

Cet algorithme fait donc l'hypothèse (réaliste selon nous) qu'ajouter des ressources à un étage qui n'est pas saturé ne permettra pas de diminuer la charge et donc la quantité de ressources requises sur un autre étage.

Cependant dans la plupart des cas, un étage en particulier consistue un goulot d'étranglement pour les performances de l'application, tandis que les autres étages demandent moins de ressources pour satisfaire le SLA. Nous pouvons donc effectuer ensuite une exploration exhaustive dans l'espace des configurations restant (borné par B_{sup}) pour trouver la configuration demandant le moins de ressources matérielles. Cette méthode d'exploration n'est pas forcément optimale et son coût, $O(M \times B_{sup})$, relativement élevé. Il existe donc très probablement une méthode plus efficace.

Algorithme 6 : getOptimalArchitectureFromModel(SLA)

début
- //détermination de la borne supérieure
- latence=SLA+1 ;
- borne=0 ;
- **tant que** *latence*>*SLA* **faire**
  - borne++ ;
  - //appel de l'algo MVA avec un degré de duplication *borne* à chaque étage
  - latence=MVA(..., borne ... borne) ;
- //exploration exhaustive de l'espace restant
- $r_{opt} = tableau[M]$;
- //tableau des degres de replication optimal rempli à *borne* par defaut
- **pour** $m = 1 ; m \leq M ; m++$ **faire**
  - $r_{opt}[m] = borne$
- //pour chaque M-uplet de configuration possible
- **pour tous les** $(r_1 \dots r_m \dots r_M)$ *où* $(1 \leq r_m \leq borne)$ **faire**
  - latence=MVA(..., $r_1 \dots r_M$) ;
  - //si le SLA est respecté, on teste si ce cas est optimal
  - **si** $latence \leq SLA$ *et* $\sum_{m=1}^{M} r_m < \sum_{i=1}^{M} r_{opt}[i]$ **alors**
    - **pour** $i = 1 ; i \leq M ; i++$ **faire**
      - $r_{opt}[i] = r_i$
- return r_{opt} ;

fin

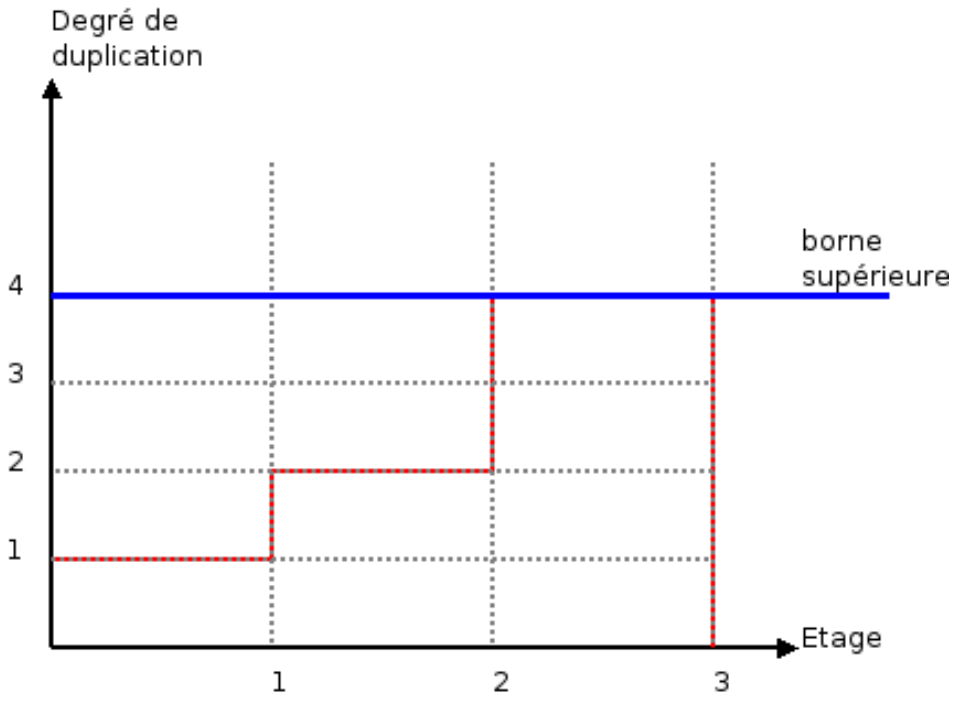

FIG. 3.8 – Exploration de l'espace d'entrée pour l'algorithme global

Coordination

L'auto-optimisation est effectuée à la fois au niveau local par l'adaptation du taux de multiprogrammation maximal de chaque serveur multiprogrammé, et au niveau architectural par l'adjonction ou la suppression de duplicatas à chaque étage de l'application multi-tiers. Notre gestionnaire d'auto-optimisation optimise donc simultanément ces deux paramètres. La synchronisation est importante et nous cherchons à optimiser en premier lieu les performances de chaque serveur avant de rajouter des noeuds supplémentaires au niveau d'un étage de l'application.

Chapitre 4

Mise en oeuvre des modèles

Dans ce chapitre nous présenterons la phase de réalisation d'un prototype ayant pour but de valider l'approche proposée dans les chapitres précédents. Nous commencerons par décrire l'environnement expérimental, puis nous validerons les modèles retenus sur cet environnement. Enfin nous proposerons un prototype de gestionnaire d'auto-optimisation.

4.1 Environnement expérimental

Cette section décrit les environnements matériels et logiciels utilisés pour valider notre approche.

4.1.1 Environnement matériel

Pour nos expériences nous avons utilisé la plate-forme Grid'5000[1]. Ce projet, démarré en 2003, a pour but de fournir à terme un ensemble de 5000 processeurs accessibles pour des travaux de recherches. En mai 2007, 2942 processeurs sont déjà disponibles, répartis sur plusieurs sites en France (voir la carte 4.1). Cette grille regroupe les grappes de machines de Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis et Toulouse. Ces grappes sont reliées entre elles par des réseaux rapides (10Gb/s). Le matériel est principalement composé d'AMD Opteron et d'Intel Xeon.

Pour nos expériences, nous avons utilisé plus particulièrement la grappe de Lille[2]. Cette grappe est constituée de 100 machines munies d'AMD Opteron bi-processeurs modèles 248, 252 et 285 (fonctionnant entre 2,2 et 2,6 GHz), embarquant 4 Go de mémoire et reliées entre elles par un réseau Gigabit Ethernet.

4.1.2 Environnement logiciel

Application Internet.

Comme environnement d'expérimentation, nous utilisons le banc d'essai RUBiS[3]. RUBiS (*Rice University Bidding System*) est un banc d'essai modélisé selon un site de vente en

[1] https ://www.grid5000.fr
[2] https ://www.grid5000.fr/mediawiki/index.php/Lille :Home
[3] http ://rubis.objectweb.org/

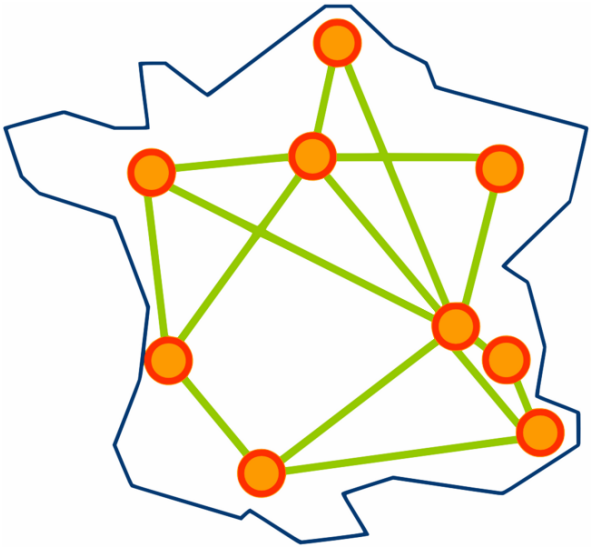

FIG. 4.1 – Carte des sites de Grid'5000

ligne (eBay,...). Il permet 26 interactions, comme ajouter de nouveaux utilisateurs, ajouter des articles au panier, passer une commande, etc. Le type de trafic étant paramétrable, nous avons utilisé le *browsing mix* (100% de requêtes Web en lecture seule) et le *bidding mix* (15% de requêtes Web en écriture).

Emulateur client.

Un émulateur génère des clients Web qui envoient des requêtes de consultation, de vente et d'enchères sur le site Web via son interface. Chaque session d'un client est une suite d'interactions, séparées par un temps d'attente entre requêtes[4] moyen de 7 secondes, conformément à la norme TPC-W[5]. Une phase de chauffe à lieu au début de chaque expérience, pendant laquelle on fait fonctionner le système sans collecter de mesures sur ses performances.

Logiciels utilisés.

L'application est constituée d'un étage métier (Tomcat[6]) et d'un étage données (MySQL[7]). Afin de distribuer la charge entre les différentes répliques de la base de données, nous utilisons le répartiteur de charge Sequoia[8]. La figure 4.2 résume l'architecture logicielle de l'application.

Nous avons utilisé la version 5.0.37 du système de gestion de bases de données MySQL, la version 5.5.23 de Apache Tomcat, et la version 2.10.6 du répartiteur de charge Sequoia.

4.1.3 Instrumentation

Pour valider le modèle à files d'attente, nous avons instrumenté le banc d'essai RUBiS pour fournir au modèle les paramètres requis en entrée. La figure 4.3 présente la méthode utilisée

[4] *think time*
[5] http ://www.tpc.org
[6] http ://tomcat.apache.org/
[7] http ://www.mysql.com/
[8] http ://sequoia.continuent.org

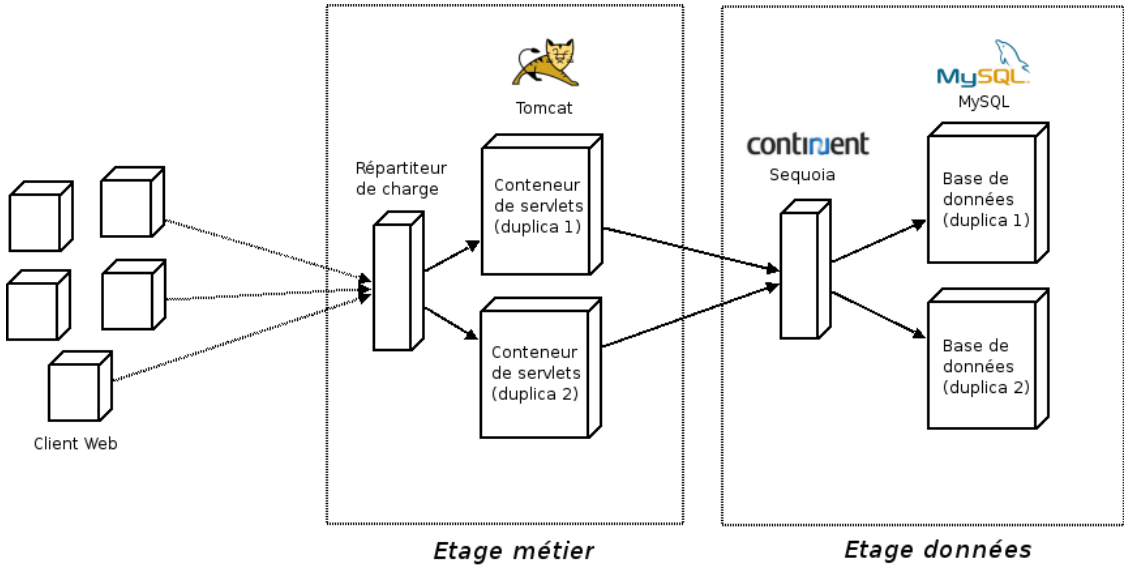

FIG. 4.2 – Architecture logicielle

pour collecter des informations de performance sur le système réel pendant son fonctionnement.

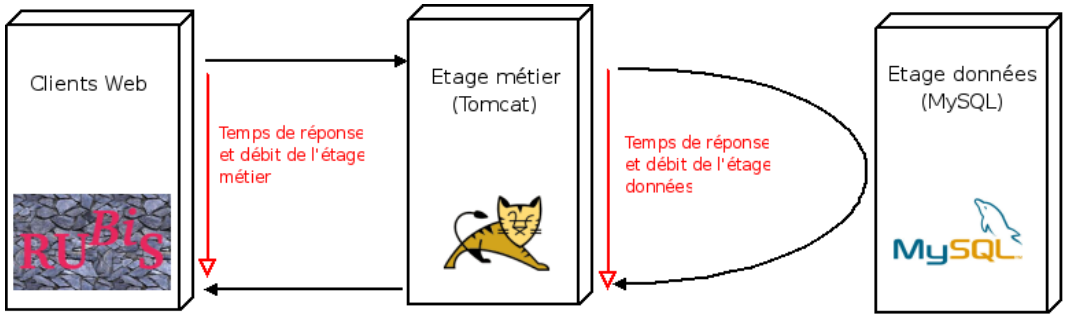

FIG. 4.3 – Points de mesure pour l'instrumentation de RUBiS

Temps de traitement.

Le principe est qu'à l'émission de chaque requête par un étage i à l'étage suivant $i+1$, on mesure le temps de traitement de la requête par l'étage suivant. Dans le cadre d'un système multi-étages, ce temps peut comprendre éventuellement des temps de traitement d'autres étages. En effet l'étage $i+1$ peut envoyer à son tour des requêtes au niveau $i+2$, et ce de manière récursive.

Pour cela on note X_i le temps passé par une requête à l'étage i ainsi qu'aux étages X_j suivants[9] ($j > i$). Les X_i sont les temps de traitement mesurés par notre instrumentation de RUBiS. Nous pouvons alors en déduire facilement les temps réels de traitement pour chaque

[9]Nous distinguons ainsi le *residence time* X_i du *service time* S_i, ce dernier ne comportant que le temps passé par une requête à l'étage i sans compter les étages suivants

étage (S_i). Pour une application composée de M étages, on a pour le dernier étage :

$$\bar{S}_M \approx \bar{X}_M \tag{4.1}$$

Nous pouvons ensuite calculer récursivement les S_i moyens pour les autres étages :

$$\bar{S}_i = \bar{X}_i - (\frac{V_{i+1}}{V_i}).\bar{X}_{i+1} \tag{4.2}$$

Utilisation de la programmation par aspects (AOP).

L'instrumentation consiste à mesurer les durées d'exécution et à compter à chaque niveau de l'application le nombre d'appels aux méthodes envoyant une requête au niveau suivant dans l'architecture.

L'ajout de code pour effectuer ces mesures a été réalisé avec AspectJ[10], ce qui nous a permis d'injecter du code directement autour des méthodes voulues sans avoir à modifier le code source du banc d'essai RUBiS. Le tissage d'aspect permet en effet d'ajouter des aspects non-fonctionnels aux applications (comme instrumenter ses performances) de manière totalement orthogonale.

4.2 Validation du modèle au niveau local

Dans cette section, nous présentons pour exemple l'application du modèle local à l'étage données de l'application RUBiS.

4.2.1 Principe

Le modèle à grain fin s'applique sur un serveur simple. Même si dans notre cadre expérimental les serveurs font partie d'un système multi-étages, nous ne prendrons pas en compte la structure en étages ou la duplication potentielle de ces étages. Seul le serveur en lui-même est modélisé ici.

Nous avons vu en 3.2.1 que le meilleur critère de performance pour un serveur Web est le ratio du débit sur la latence. Nous utiliserons donc ce critère par la suite.

Le principe du modèle à grain fin est d'approximer la forme de la courbe de performance à une parabole. Pour trouver les paramètres de cette parabole, nous utilisons la méthode des moindres carrés.

4.2.2 Méthode des moindres carrés

Soit f la fonction à déterminer, ici $f(x) = ax^2 + bx + c$. Les paramètres à déterminer sont donc a, b et c. On va chercher les valeurs de ces paramètres qui minimiseront la somme des écarts au carré (voir équation 4.3).

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2 \tag{4.3}$$

Les y_i sont des valeurs mesurées sur le système réel. Pour obtenir ces valeurs, nous instrumentons puis faisons fonctionner le système à modéliser en essayant de couvrir l'ensemble de l'espace d'entrée des paramètres.

[10] http ://www.eclipse.org/aspectj

4.2.3 Calibration pour une charge en lecture

Nous allons donner maintenant un exemple de calibration du modèle à grain fin. Pour cela nous faisons fonctionner le système RUBiS en faisant varier le nombre de clients. Nous avons préalablement instrumenté le système pour obtenir des informations sur le débit et la latence des requêtes au niveau de chaque étage.

Paramétrage.

L'architecture du service Internet est composé d'un serveur Tomcat, un répartiteur de charge Sequoia et d'une base de données MySQL. Le nombre de clients augmente régulièrement par palier de 20 au cours de l'expérience. Chaque point représente une simulation de 5 minutes. La limite sur le taux de parallélisme maximal est fixée à un niveau très élevé sur les deux étages de l'application (1000 pour la base MySQL et 5000 pour le serveur Tomcat).

Les figures 4.4, 4.5, 4.6 et 4.7 présentent les données d'expérience du banc d'essai RUBiS avec des clients consultant la description de produits *(browsing mix)*, ce qui ne génère aucune mise à jour de la base de données.

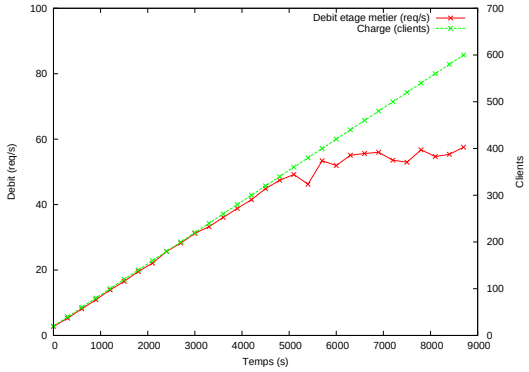

FIG. 4.4 – Débit au niveau métier (Tomcat) - browsing mix

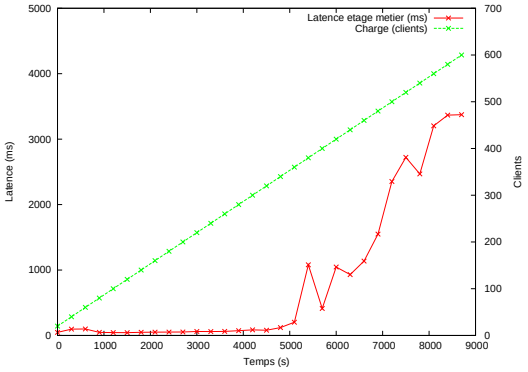

FIG. 4.5 – Latence au niveau métier (Tomcat) - browsing mix

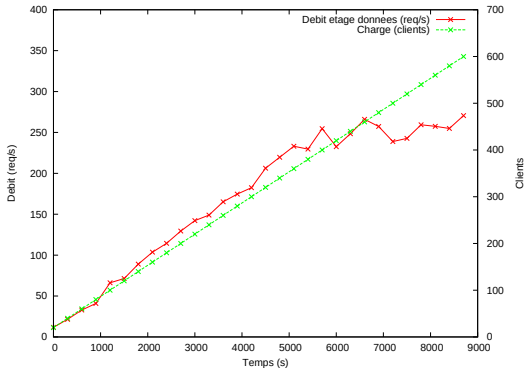

FIG. 4.6 – Débit au niveau données (MySQL) - browsing mix

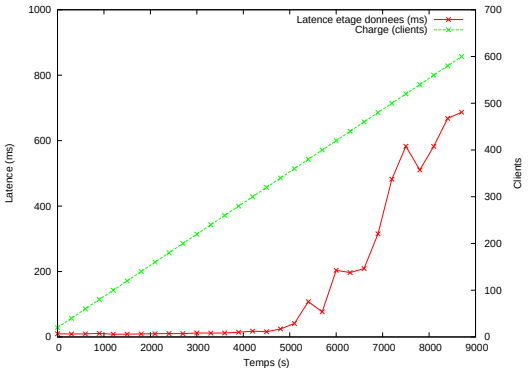

FIG. 4.7 – Latence au niveau données (MySQL) - browsing mix

On voit qu'à partir d'un certain nombre de clients, le système commence à saturer. Le nombre de requêtes traitées n'augmente plus, et le temps de traitement des requêtes croit

linéairement. Notre but va être de caractériser le nombre de client optimal devant être admis dans le système pour avoir les meilleurs performances (ratio débit/latence). Pour cela nous traçons la courbe de performance de l'étage MySQL, qui correspond à ce ratio. la figure 4.8 présente ce ratio de performance. Il faut garder à l'esprit que la valeur du ratio de performance n'a pas de sens en soit, seule la forme de la courbe et donc la comparaison entre différentes valeurs de performance a un intérêt.

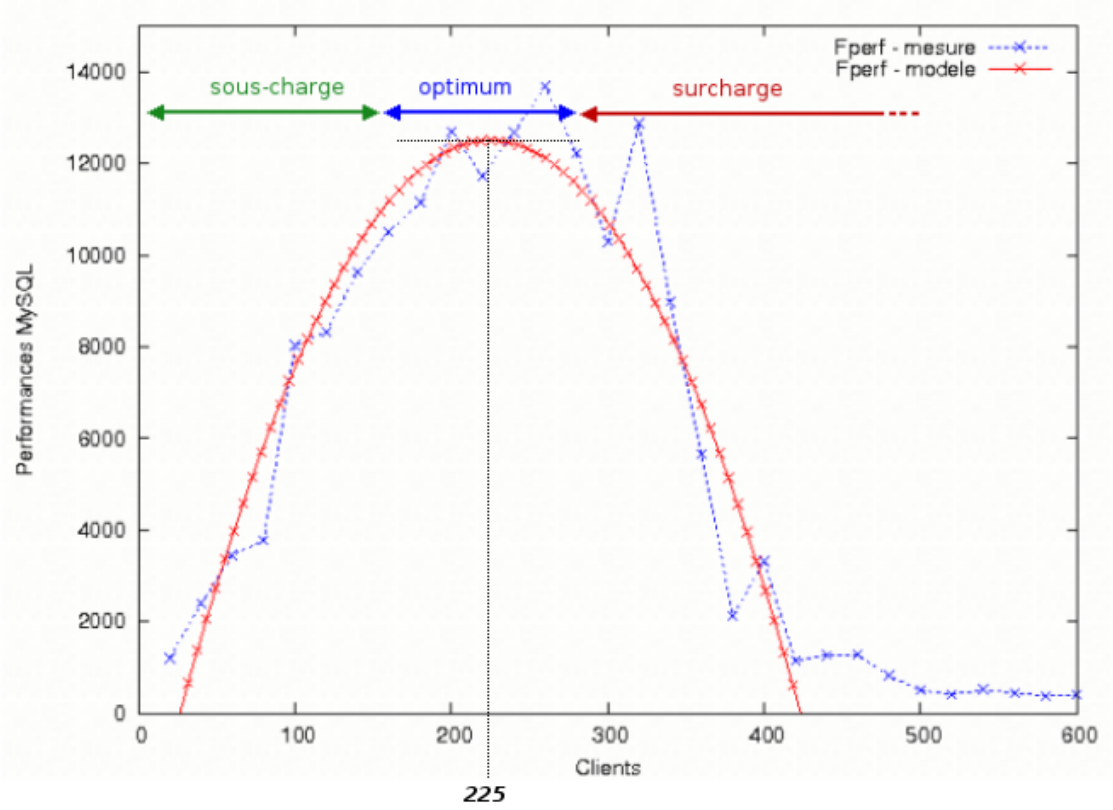

FIG. 4.8 – Performances de l'étage données (MySQL) - *browsing mix*

Conformément à la théorie, ce ratio a la forme d'une parabole. Nous effectuons donc une regression quadratique pour obtenir les paramètres de cette parabole. Dans notre cas, les paramètres obtenus pour l'étage données sont :

```
a = -0.3177
b = 142.9
c = -3553.5
```

La parabole associée est tracée en rouge sur la figure 4.8. Une fois cette parabole obtenue, nous pouvons facilement déterminer quel est l'antécédent du maximum. L'équation 4.4 donne l'axe de symétrie d'une parabole d'équation $f(x) = ax^2 + bx + c$.

$$x = \frac{-b}{2a} \tag{4.4}$$

Ainsi dans notre cas, le maximum de performance est atteint avec un nombre de client aux alentours de 225.

Performances au niveau métier.

Même si les valeurs numériques changent, la courbe de performance pour l'étage métier à la même forme que celle de l'étage données. L'approximation parabolique à ce niveau de l'application donne les paramètres suivants :

```
a = -0.014565
b = 6.55
c = -188
```

L'optimal en nombre de client, représenté sur la figure 4.9, est le même que pour le niveau données. En effet la base de données constitue le goulot d'étranglement de l'application, et les performances du niveau métier sont donc limitées par celles du niveau données. Comme sur chacun de ces étages le degré de duplication est de 1, les courbes sont au même emplacement. Avec un degré de duplication supérieur à l'étage données, nous pourrions constater le déplacement de l'optimal à l'étage métier.

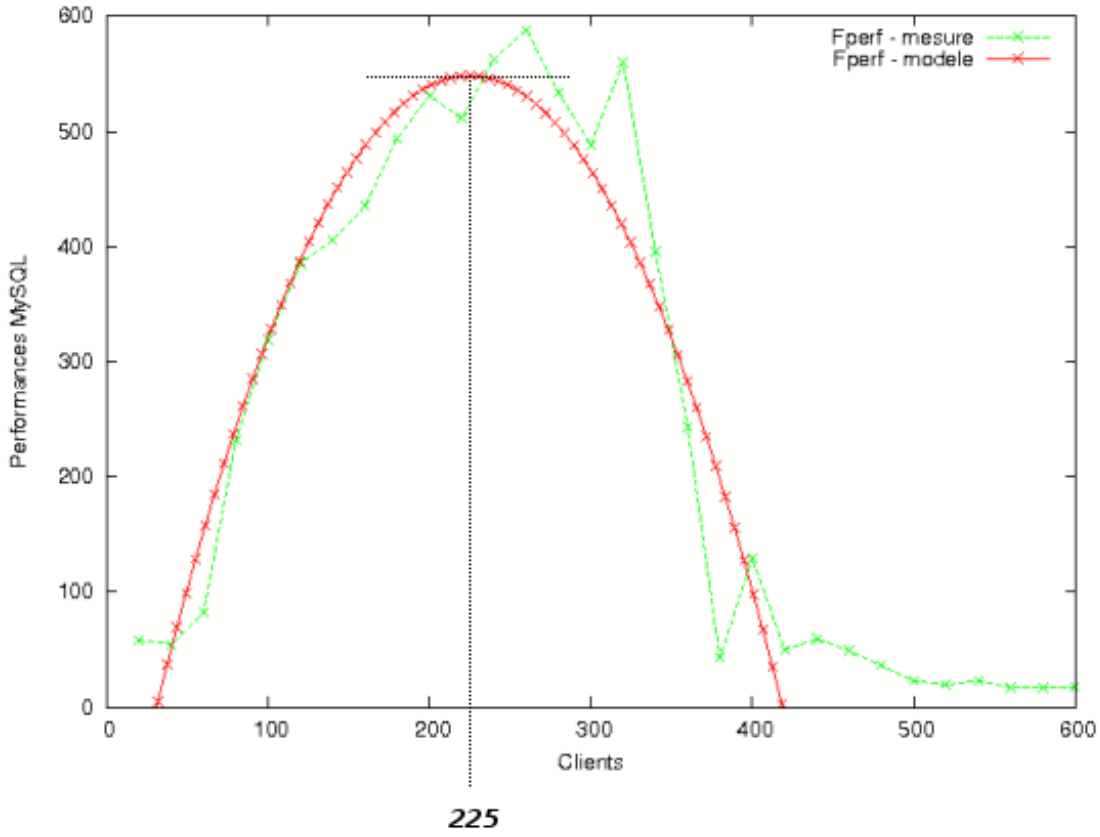

FIG. 4.9 – Performances de l'étage métier (Tomcat) - *browsing mix*

4.2.4 Vérification de la validité du résultat

Cas d'erreur.

Nous avons vu que pour que cette méthode d'approximation parabolique fonctionne, nous devons disposer de plusieurs points de mesure répartis dans l'espace d'entrée des paramètres afin d'obenir une parabole nette, sinon l'approximation parabolique risque de donner des résultats aberrants.

Heiss & Wagner ont mis en évidence différents cas d'erreur [9]. Ces erreurs surviennent dans le cas où la forme de la courbe de performance varie brutalement, et qu'on ne dispose de pas assez de points de mesure (figure 4.10), ou alors quand la courbe de performance est très écrasée et ne permet pas de déterminer le sens de la parabole (figure 4.11).

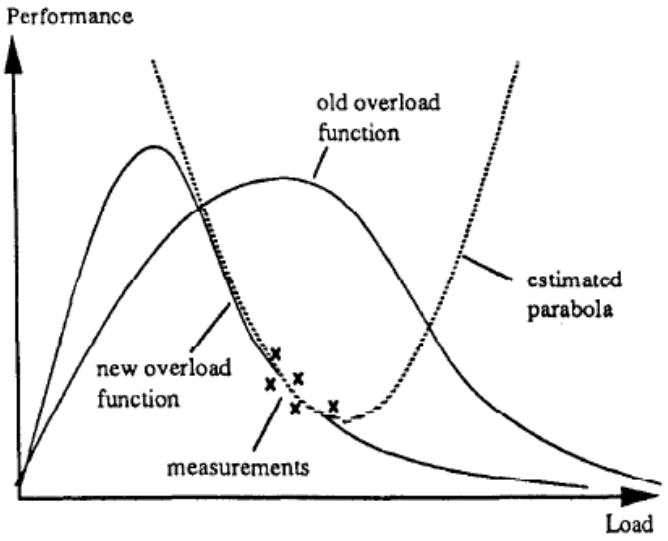

FIG. 4.10 – Cas d'erreur : Evolution rapide de la parabole [9]

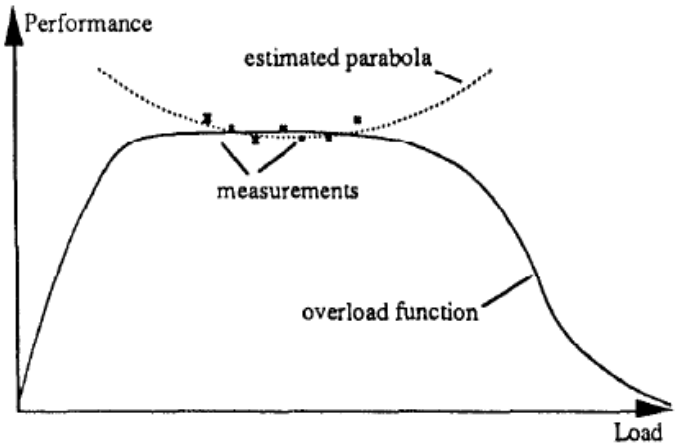

FIG. 4.11 – Cas d'erreur : Courbe de performance en plateau [9]

Dans les deux cas les erreurs sont dues à un manque de mesures et/ou de précision de

celles-ci.

Vérification.

Nous pouvons facilement déterminer si le résultat fourni par l'approximation parabolique est aberrant ou non. En effet si le coefficient a de la parabole est supérieur à zéro, la parabole va être décroissante puis croissante, ce qui n'est pas valide. Pour que la valeur calculée par le modèle puisse être considérée comme valide, nous contrôlons donc que $a < 0$. Dans l'exemple présentée figure 4.16, c'est bien le cas.

4.2.5 Calibration avec une charge comportant des écritures

Le modèle par approximation parabolique permet de déterminer le nombre optimal de clients devant accéder en parallèle au service pour atteindre les performances optimales. Cependant si cet optimal était constant, il suffirait de le déterminer statiquement une fois pour toutes. Or nous avons vu que cet optimal varie au cours du temps [9], en fonction notamment du type de charge appliquée sur le système (par exemple la proportion de mises à jour dans les requêtes pour une base de données). Une approche dynamique mesurant en tant réel les performances du système pour adapter sa configuration permettrait donc d'obtenir de meilleures performances qu'une optimisation effectuée à un moment particulier et non modifiée par la suite.

Pour vérifier l'intérêt d'une telle approche, nous allons appliquer la méthode d'approximation parabolique sur le système RUBiS, avec cette fois ci une charge de type enchères *(bidding mix)*, qui contient 15% de requêtes Web effectuant des mises à jour. Au niveau de la base de données, ceci génère une proportion d'environ 6% de mises à jour au niveau de la base de données, car une requêtes Web contenant une mise à jour effectue aussi des lectures.

Comme précédement, l'expérience est effectuée avec un système composé d'un serveur Tomcat, un répartiteur de charge Sequoia et une base de donnée MySQL. Les résultats sont présentés sur les figures 4.12, 4.13, 4.14 et 4.15. Chaque point sur la courbe représente une moyenne de la métrique sur une durée de 5 minutes. Le nombre de clients augmente de manière linéaire.

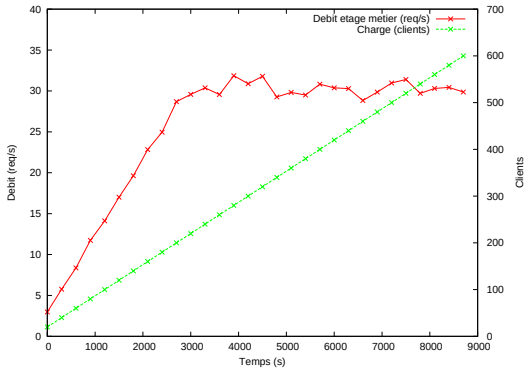

FIG. 4.12 – Débit au niveau métier (Tomcat) - bidding mix

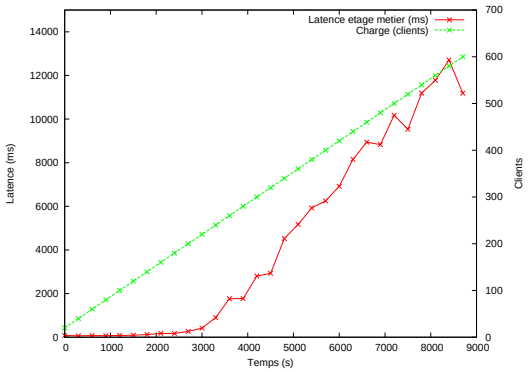

FIG. 4.13 – Latence au niveau métier (Tomcat) - bidding mix

Nous remarquons que la forme de la courbe ressemble à celle obtenue précédement avec un type de trafic de requêtes différent, mais qu'elle diffère au niveau de certains paramètres

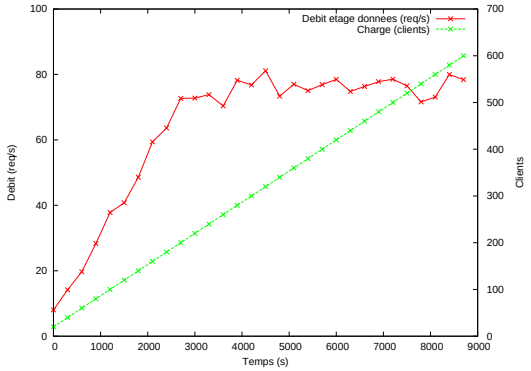

FIG. 4.14 – Débit au niveau données (MySQL) - bidding mix

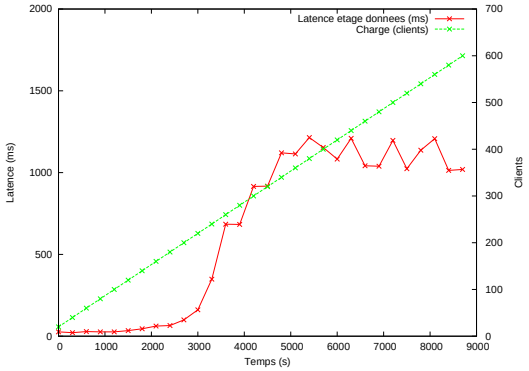

FIG. 4.15 – Latence au niveau données (MySQL) - bidding mix

(point d'inflexion du à la saturation plus bas par exemple).

L'approximation parabolique sur le ratio de performance obtenu à partir des données du serveur MySQL donne les paramètres suivant :

```
a = -0.085
b = 20.8
c = -127.8
```

La parabole associée est présentée sur la figure 4.16. Le calcul de l'optimal donne un nombre de client égal à 122.

4.2.6 Intérêt d'une configuration dynamique

Ainsi en passant de 0% de requêtes Web en écriture à 15%, le nombre de clients optimal à diminué de plus de 100 clients. Ceci s'explique par le fait que des lectures peuvent être plus facilement parallélisées que des mises à jour, car ces dernières nécessitent des verrous en accès sur la base de données. Nous avons peut donc résumer en disant que plus le nombre d'écritures est important, plus le nombre de clients optimal sera faible.

Comme la proportion de lectures / écritures évolue au cours du temps, nous avons donc bien besoin d'une approche permettant de reconfigurer dynamiquement le degré de parallélisme maximal.

La figure 4.17 illustre le déplacement des paraboles approximant la courbe de performance des serveurs MySQL avec des charges sans mises à jour *(browsing)* et avec 15% de mises à jour *(bidding)*.

4.3 Validation du modèle architectural

Cette section présente la phase d'évaluation du modèle à file d'attente au niveau architectural de l'application. Nous y présentons la méthode de collecte des données, ainsi que la calibration du modèle. Les résultats du modèle sont enfin comparés aux mesures effectuées sur le système réel pour vérifier la justesse de notre approche.

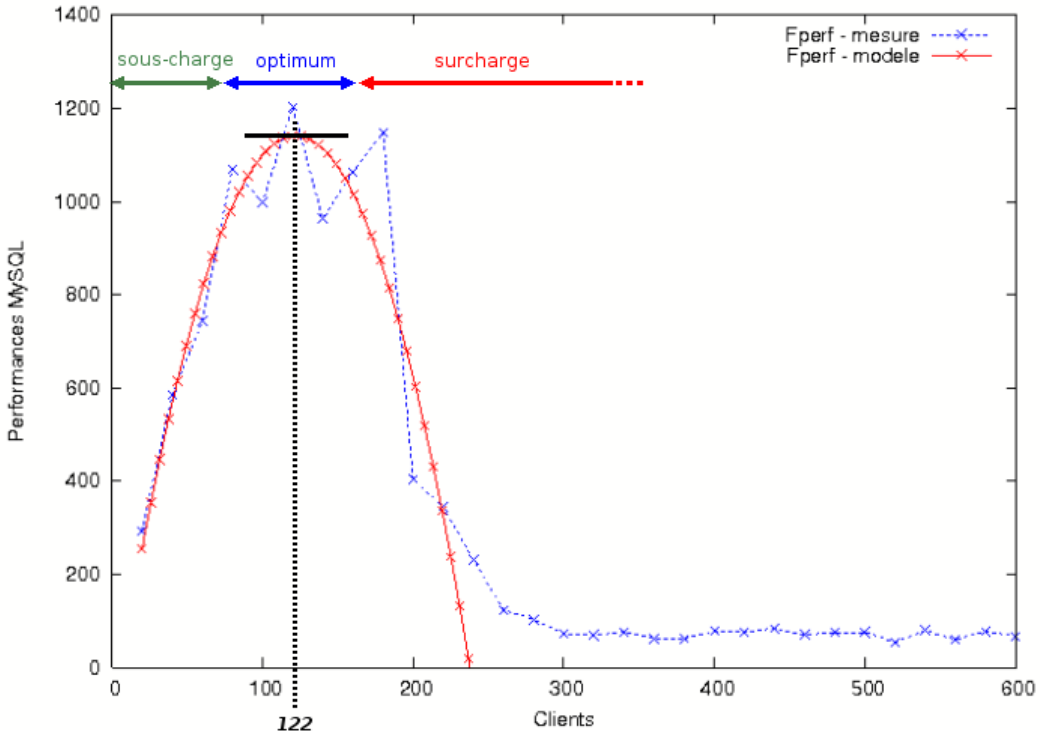

FIG. 4.16 – Performances de l'étage données (MySQL) - *bidding mix*

4.3.1 Principe

Au niveau architectural de l'application, nous utilisons un modèle fermé à files d'attente. A partir de ces hypothèses, nous pouvons appliquer l'algorithme MVA [20] qui permet de déduire le temps de réponse global d'un réseau de files d'attente formant une application multi-étages.

Nous allons utiliser ce modèle pour prédire le temps de réponse du système dans une configuration donnée, et si ce temps de réponse n'est pas conforme par rapport au SLA, trouver une configuration qui donne un temps de réponse meilleur tout en essayant d'économiser le nombre de machines utilisées.

4.3.2 Calibration

Comme nous l'avons vu en 3.2.2, l'algorithme MVA nécessite différents paramètres en entrée qui peuvent être obtenus grâce à l'instrumentation que nous effectué sur les composants du banc d'essai RUBiS. En utilisant les données d'expérience présentées dans la section 4.2.5, nous pouvons obtenir les valeurs des paramètres en entrée.

Calcul des paramètres

Pour obtenir la latence aux étages intermédiaires de l'application alors que nous ne connaissons que le temps de réponse à l'entrée du système, nous appliquons le calcul vu en 3.2.2. Notre application est composée de 2 étages (Tomcat et MySQL). Nous commençons par calculer le ratio de visite entre les deux étages (V_2). Comme chaque client accède à l'étage Tomcat, il suf-

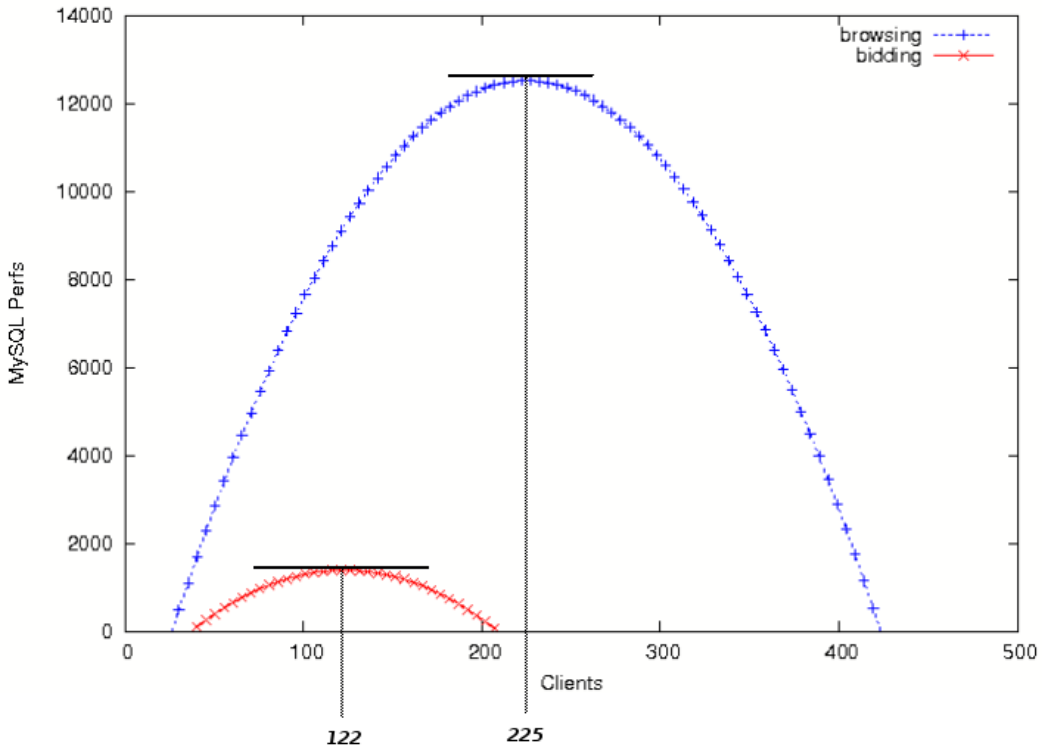

FIG. 4.17 – Approximations paraboliques avec deux types de charge

fit de comparer le débit de requêtes entre les étages 1 (Tomcat) et 2 (MySQL). Nous obtenons ainsi $V_2 = 2,68$ (et $V_1 = 1$).

L'instrumentation du système nous fournis le *residence time* moyen ($\bar{X}_m$) pour chaque étage m de l'application ($1 \leq m \leq M$). Pour le dernier étage (ici $M = 2$) nous avons $\bar{S}_2 \approx \bar{X}_2 = 11,8$. Ce temps est obtenu en faisant fonctionner le système à faible charge pour connaître le temps de traitement d'une requête individuelle.

Nous pouvons ensuite calculer récursivement les S_i moyens pour l'étage Tomcat :

$$\begin{aligned} \bar{S}_1 &= \bar{X}_1 - (\frac{V_2}{V_1}).\bar{X}_2 && (4.5) \\ &= 45,5 - 2,68 \times 11,8 && (4.6) \\ &= 13,876 && (4.7) \end{aligned}$$

Tableau de synthèse

La figure 4.18 présente le tableau de synthèse des différents paramètres et leur valeur associée.

Paramètre	**Signification**	**Valeur**
M	Nombre d'étages dans l'application	2
N	Nombre de sessions clientes	varie
$\bar{Z}$	Temps d'attente chez le client (think time)	7 s
$\bar{S}_1$	Latence moyenne induite par l'étage 1	13,876
$\bar{S}_2$	Latence moyenne induite par l'étage 2	11,8
V_1	Ratio de visite pour l'étage 1	1
V_2	Ratio de visite pour l'étage 2	2,68
r_1	Degré de duplication à l'étage 1 de l'application	1
r_2	Degré de duplication à l'étage 2 de l'application	1

FIG. 4.18 – Calibration du modèle MVA

4.3.3 Validité de l'approche

Une fois le modèle paramétré, nous pouvons comparer la prédiction du modèle avec les données mesurées. La figure 4.19 présente le temps de réponse du système en fonction du nombre de clients, ainsi que la prédiction du modèle pour ce même nombre de client. On peut vérifier que le modèle concorde bien avec la réalité, car l'écart moyen entre la prédiction du modèle et la mesure effectuée sur le système réel est de 242 ms, ce qui donne une erreur moyenne de 12% environ. Cette erreur est en partie due au temps d'expérience pour les mesures, chaque point représentant la moyenne des latences sur une exécution de 7 minutes. Comme RUBiS envoie des requêtes ayant des durées d'exécution variables, la latence moyenne peut donc fluctuer au cours d'une expérience si sa durée n'est pas assez longue. En diminuant le nombre de points de mesure mais en augmentant la durée d'expérience à chaque point nous obtiendrions probablement une erreur moyenne moindre.

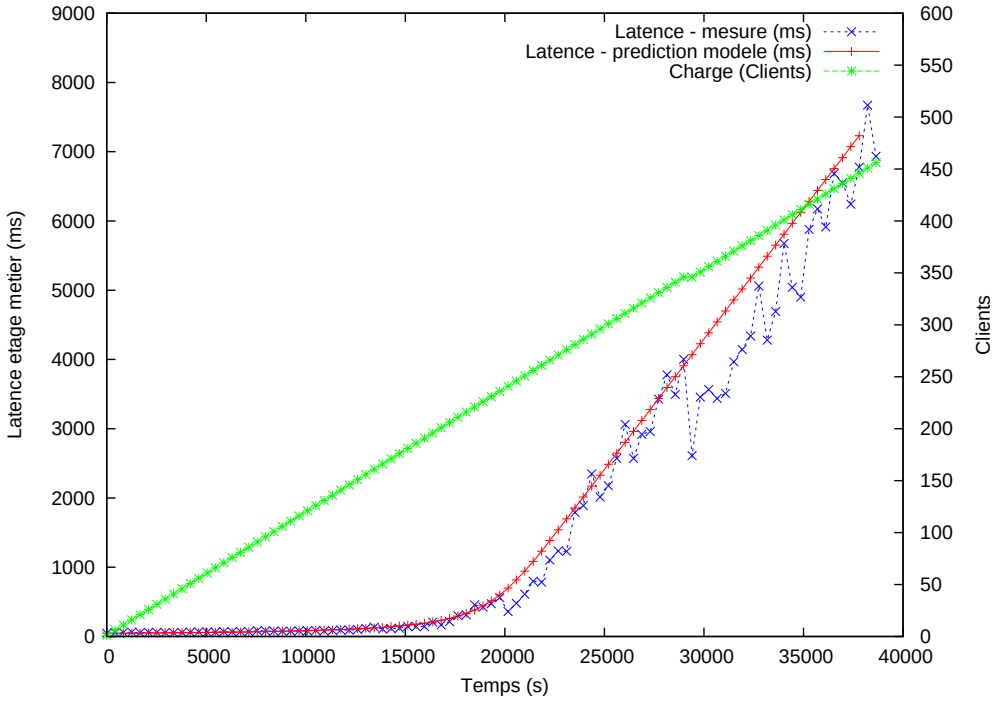

FIG. 4.19 – Prédiction du modèle et mesure réelle

4.4 Réalisation d'un gestionnaire de ressources pour optimisation dynamique

Une fois les modèles définis et validés, ils vont être intégrés dans des gestionnaires autonomes. Ces gestionnaires seront construits sur la base de boucles de commande (voir figure 4.20), qui recueuillent des informations sur le système administré via des sondes logicielles, calculent la nouvelle configuration, et appliquent cette dernière sur le système grâce à des actionneurs. A terme, ces gestionnaires seront intégrés dans la plate-forme Jade (voir 4.4.1).

4.4.1 La plate-forme Jade

La solution d'auto-optimisation présentée dans ces pages s'inscrit dans le cadre de Jade [4], un environnement pour la construction d'outils d'administration autonome. Jade est basée sur le modèle à composants Fractal [5]. La vision de l'application comme un ensemble de composants interconnectés permet de reconfigurer dynamiquement l'application pendant son exécution.

Le fonctionnement de Jade est basé sur le principe des gestionnaires autonomes. Ces derniers interragissent avec l'application administrée via des sondes et des actionneurs pour récupérer des informations sur l'état du système, en déduire une action à effectuer, et appliquer cette action sur le système. Chaque gestionnaire forme donc une boucle de commande indépendante et considérant le système administré comme une "boîte noire" générique.

Différents gestionnaires apportant des propriétés non fonctionelles à l'application peuvent ainsi être appliqués. La figure 4.21 illustre l'utilisation de deux gestionnaires autonomes (auto-optimisation et auto-réparation) à une application administrée.

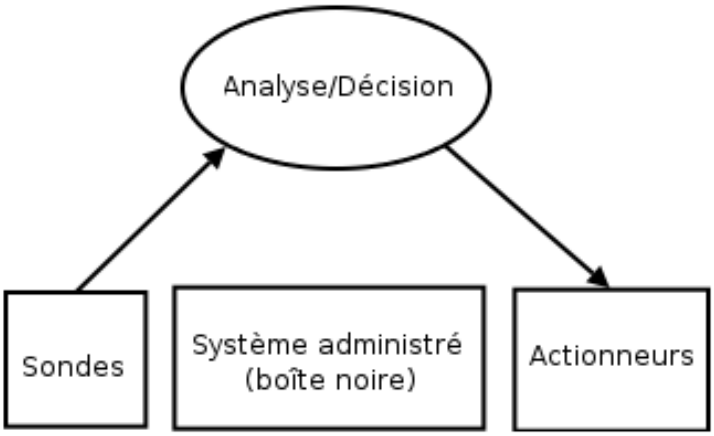

FIG. 4.20 – Principe d'une boucle de commande

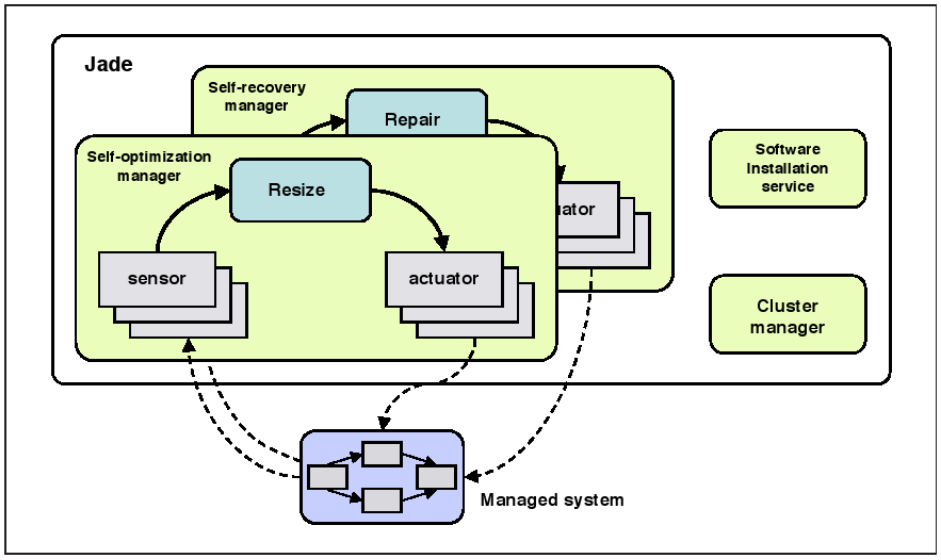

FIG. 4.21 – Boucles de commande dans Jade [4]

4.4.2 Configuration dynamique au niveau local

Une boucle de commande sera placée sur chaque serveur individuel. Elle servira à éventuellement reconfigurer le duplicata en fonction de l'environnement. La partie d'analyse/décision de la boucle intégrera le modèle présenté en 3.2.1. Nous jouerons principalement sur le taux de parallélisme maximum du serveur. Sur chaque niveau de l'application, l'optimisation au niveau local peut être vu sur un axe orthogonal au niveau de duplication (voir figure 2.4).

4.4.3 Configuration dynamique au niveau architectural

A un niveau plus élevé dans l'architecture, une boucle de commande sera en charge de l'allocation (ou la désallocation) de ressources au niveau des niveaux. Le modèle général obtenu en 3.2.2 sera intégré dans la boucle, qui trouvera éventuellement un changement de configuration à effectuer, et l'appliquera. La figure 2.5 schématise cette optimisation.

4.4.4 Configuration dynamique globale

Coordination.

L'avantage de cette approche est de tirer le meilleur parti de chaque serveur avant de dupliquer lorsque la charge ne peut vraiment plus être traitée par le nombre de serveur actuellement affecté.

Une des difficultés posées par la présence de plusieurs gestionnaires autonomes est leur coordination. En effet il faut que le fonctionement simultané des gestionnaires n'entraine pas d'oscillations dans l'état du système. Prenons par exemple le cas d'un niveau surchargé, l'optimisation au niveau local aura pour effet d'améliorer le débit de chaque serveur, qui seront donc moins chargés. Si les gestionnaires n'étaient pas coordonnées, la boucle de commande au niveau architecture pourrait voir la baisse de charge sur chaque duplicata (file d'attente plus courte par exemple) et ainsi décider de supprimer des serveurs de traitement au niveau de ce niveaux. On traiterait alors autant de requêtes avec moins de serveurs, mieux configurés.

Par contre si l'optimisation au niveau global se fait avant l'optimisation au niveau local, de nouveaux serveurs seront ajoutés, et deviendront inutiles après optimisation au niveau local.

Il faut donc que les gestionnaires soient bien synchronisés, c'est-à-dire que l'optimisation à grain fin (au niveau local) soit effectuée sur chaque noeud (serveur) avant d'appliquer l'optimisation au niveau global. Ceci suit dans l'ordre le principe de tirer le meilleur d'un serveur avant d'éventuellement en ajouter un autre.

Architecture proposée.

La solution proposée est présentée sur la figure 4.22. A chaque niveau de l'application (métier, données, ...), le premier serveur héberge un composant logiciel chargé de collecter des informations de performances. Ces informations sont transmises à un gestionnaire d'optimisation de ressources qui a partir de ces données, exécute les algorithmes d'optimisation présentés en 3.3. Si un changement est décidé concernant la configuration des serveurs au niveau local, l'ordre est transmis à l'ensemble des serveurs du niveau concerné. Ainsi nous collectons des statistiques sur un seul serveur mais nous appliquons les changements sur l'ensemble des serveurs de son niveau. Ce système fonctionne en faisant les hypothèses que le surcoût induit par l'instrumentation reste faible, que les machines soient toutes identiques au moins au sein d'un étage, et que les répartiteurs de charge soient équitables.

Si un changement au niveau architectural est requis, le gestionnaire de ressources ajoute ou retire directement les ressources aux étages concernés.

La synchronisation entre les deux optimisations est réalisée de manière programmatique grâce à un verrou. Lorsque un des deux algorithmes (voir section 3.3) décide d'effectuer une reconfiguration, il entre en section critique, empêchant ainsi l'autre algorithme de lancer une optimisation de manière concurrente.

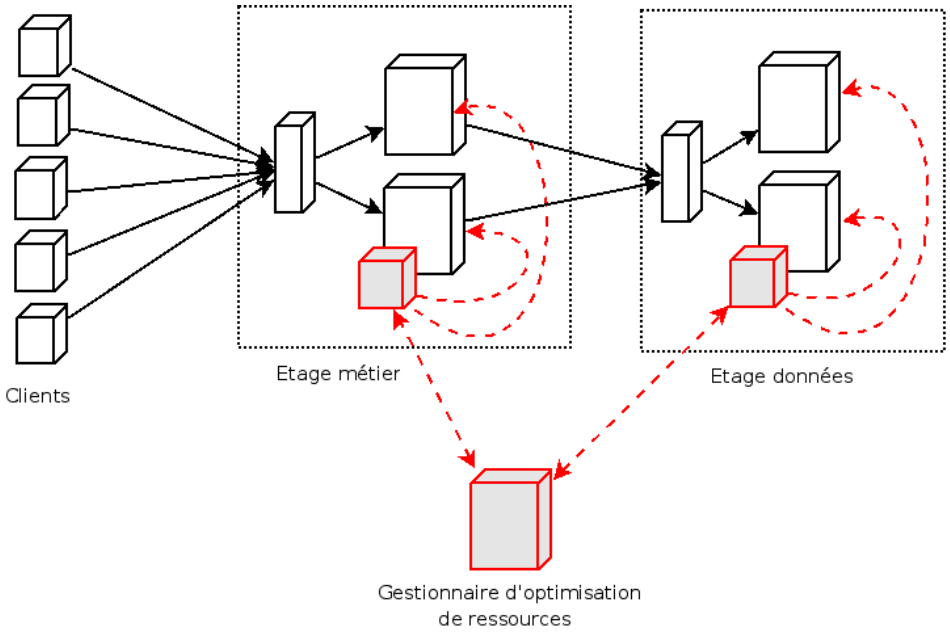

FIG. 4.22 – Gestion de ressources : solution architecturale

Chapitre 5

Conclusion

5.1 Synthèse

Nous avons vu dans ce projet les méthodes actuelles utilisées pour administrer de manière automatique des systèmes potentiellement complexes (dupliqués, à plusieurs niveaux). Les gestionnaires autonomes utilisés ont pour but d'optimiser les performances du système lorsque la charge qui lui est appliquée varie. Les approches actuelles tendent à montrer que l'optimisation d'un système ne peut plus se contenter d'approches *ad hoc* basées sur des heuristiques, mais nécessitent une part plus ou moins importante de connaissance du système, et donc de modélisation. Construire des gestionnaires autonomes qui restent génériques dans ce contexte devient donc un défi en soit. Nous avons proposé ici une approche visant à fournir un modèle simplifié d'une architecture multi-niveaux, afin de pouvoir effectuer de l'allocation de ressources de manière optimale. Ce gestionnaire global fonctionne en parallèle des optimiseurs locaux, qui adaptent les paramètres de configuration de chaque serveur dans le système à l'environnement courant. Pour l'instant les approches existantes se concentrent soit sur l'optimisation du fonctionnement d'un serveur simple, soit sur l'allocation de ressources au niveau architectural. L'utilisation simultanée de ces deux approches devrait donner des résultats optimums, tant au niveau performances qu'utilisation des ressources.

5.2 Perspectives

Dans les prochains moins nous allons nous baser sur les modèles présentés dans ce rapport pour construire des gestionnaires autonomes d'auto-optimisation intégrés à Jade. Ces gestionnaires effectueront une optimisation à deux niveaux, en utilisant au mieux chaque serveur et en minimisant le nombre de ressource utilisées pour respecter un objectif de performance donné.

Parallèlement, nous sommes actuellement en collaboration avec le projet NECS[1], où collaborent automaticiens et informaticiens pour fournir des modèles plus précis de serveurs Web. Les premiers résultats sont encourageants et un modèle analytique permettant de caractériser finement un serveur est sur le point d'être finalisé [14].

Enfin a plus long terme, nous avons pour objectif de prendre en compte d'autres critères de performance et paramètres de réglage sur les systèmes. Par exemple nous pensons que l'optimisation de la consommation d'énergie constitue un défi majeur, étant donnés les impacts

[1] http ://necs.inrialpes.fr/

en termes écologiques et financiers que génère l'utilisation de milliers de machines en parallèle chez les hébergeurs de services Internet.

Bibliographie

[1] Sandjai Bhulai. Analysis of end-to-end response times of multi-tier internet services. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 2007.

[2] Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sacha Krakowiak, Adrian Mos, Noël De Palma, Vivien Quéma, and Jean-Bernard Stefani. Architecture-Based Autonomous Repair Management : An Application to J2EE Clusters. In *The 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005)*, Orlando, FL, USA, October 2005.

[3] Sara Bouchenak, Noël De Palma, Daniel Hagimont, and Christophe Taton. Autonomic management of clustered applications. In *IEEE International Conference on Cluster Computing*, Barcelona, Spain, September 2006.

[4] Sara Bouchenak, Noël de Palma, Daniel Hagimont, and Christophe Taton. Autonomic management of clustered applications. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'06)*, Barcelona, Spain, September 2006.

[5] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12) :1257–1284, 2006.

[6] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centres. In *Symposium on Operating Systems Principles*, pages 103–116, 2001.

[7] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In *Network Operations and Management Symposium*, April 2002.

[8] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *WWW '04 : Proceedings of the 13th international conference on World Wide Web*, pages 276–286, New York, NY, USA, 2004. ACM Press.

[9] Hans-Ulrich Heiss and Roger Wagner. Adaptive load control in transaction processing systems. In *VLDB*, pages 47–54, 1991.

[10] Klaus Herrmann, Gero Muhl, and Kurt Geihs. Self-management : The solution to complexity or just another problem ? *IEEE Distributed Systems Online*, 6(1) :1, 2005.

[11] Krishna Kant and Prasant Mohapatra. Scalable internet servers : issues and challenges. *SIGMETRICS Performance Evaluation Review*, 28(2) :5–8, 2000.

[12] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003.

[13] William LeFebvre. Cnn.com : Facing a world crisis. In *LISA*, 2001.

[14] Luc Malrait and Sylvain Durand. Modélisation et identification d'un serveur de bases de données. Master's thesis, INPG, 2007.

[15] Daniel A. Menascé. Automatic qos control. *IEEE Internet Computing*, 7(1) :92–95, 2003.

[16] Daniel A. Menascé and Virgilio Almeida. *Capacity Planning for Web Services : metrics, models, and methods.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[17] J.M. Milan-Franco, Ricardo Jiménez-Peris, Marta Patino-Martinez, and Bettina 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.

[18] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI*, pages 61–76, 2004.

[19] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. *Real Time Systems Journal*, 23(1-2), 2002.

[20] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2) :313–322, 1980.

[21] SARDES Project. SARDES. http ://sardes.inrialpes.fr/.

[22] Swaminathan Sivasubramanian, Pierre Guillaume, Maarten van Steen, and Sandjai Bhulai. Sla-driven resource provisioning of multi-tier internet applications. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 2006.

[23] Christophe Taton, Sara Bouchenak, Noël De Palma, Daniel Hagimont, Sacha Krakowiak, and Jean Arnaud. Administration autonome de services Internet : Expérience avec l'auto-optimisation. In *5ème Conférence Française sur les Systèmes d'Exploitation (CFSE 2006)*, Le Canet en Roussillon, France, October 2006.

[24] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *SIGMETRICS '05 : Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 291–302, New York, NY, USA, 2005. ACM Press.