Introduction de fonctionnalités d'auto-optimisation dans une architecture de selfbenchmarking

El Hachemi Bendahmane

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

El Hachemi BENDAHMANE

Thèse dirigée par **Patrice MOREAUX** et
codirigée par **Bruno DILLENSEGER**

préparée au sein du **Laboratoire LISTIC**
dans **l'École Doctorale SISEO**

Introduction de fonctionnalités d'auto-optimisation dans une architecture de selfbenchmarking

Thèse soutenue publiquement le 25 septembre 2012
devant le jury composé de :

Mme Nihal PEKERGIN
Professeur des Universités, Université de Paris XII – Créteil, rapporteur
M. Mikaël SALAÜN
Professeur des Universités, École des Mines de Nantes - Nantes, rapporteur
M. Thomas BEGIN
Maître de Conférences, Université Claude Bernard – Lyon, examinateur
M. Laurent BROTO
Maître de Conférences, INPT/ENSEEIHT - Toulouse, examinateur
M. Didier DONSEZ
Professeur des Universités, Université Joseph Fourier – Grenoble, examinateur

Remerciements

Lorsque vous résolvez un problème, vous devriez remercier Dieu et passer au problème suivant. [David Dean Rusk]

Protocole oblige : d'abord la famille,
mes parents, mes frères et sœurs.

Ensuite, et par ordre chronologique, j'ai une pensée pour

Limoges : Chercheurs et personnel du laboratoire XLIM. Je cite parmi les professeurs K. Tamine, J-L. Lanet, D. Ghazanfarpour, parmi les collègues : Salah, Djalil, Saif, Nassima et Thibault. Aussi les doctorants qui sont aujourd'hui docteurs : Nadir et Boussad.

Grenoble : Je n'oublierai jamais les trois ans passés à Orange Labs. Je cite Christian Bayle, Alexandre Lefebvre, Aimé Vareil, Véronique, Collete, Christelle, ... et la liste est longue. Merci à toutes et à tous.

Annecy : Le temps passé au LISTIC. Je remercie le personnel du LISTIC de l'Université de Savoie, mes collègues et amis doctorants : Nabil, Renaud, Fabien et la liste est longue ...

Enfin un Remerciement Spécial à mes deux encadrants de thèse :

Bruno Dillenseger de France Télécom, d'abord et avant tout pour ses qualités humaines, pour ses conseils et sa disponibilité, mais aussi pour son sérieux et son rigueur. Merci pour toutes les heures d'*Extreme Progamming* que nous avons passées ensemble devant l'ordinateur. Je ne divulgue pas un secret si je dis que Bruno, en dehors de ses qualités techniques, est quelqu'un avec qui on a toujours envie de travailler car tout simplement, c'est quelqu'un de bien.

Patrice Moreaux du LISTIC, pour tout l'effort qu'il a fait pour l'aboutissement de cette thèse.

Et enfin, un grand remerciement à Nihal Pekergin et Mikaël Salaün d'avoir accepté de rapporter sur mon manuscrit, pour leur retour rapide pour leurs commentaires de qualité et leurs conseils avisés.

Un grand merci à Didier Donsez, Laurent Broto et Thomas Begin d'avoir accepté de faire partie de jury de ma thèse, merci beaucoup.

Table des matières

Table des figures

Liste des tableaux

Listings

Chapitre 1

Introduction générale

Sommaire

1.1 Motivations

1.1.1 Test de performance et benchmarking

La complexité croissante des systèmes informatiques, en termes de passage à l'échelle, répartition et architecture, rend de plus en plus délicate l'évaluation de leur performance. Si les approches à base de modèles permettent d'obtenir rapidement des prévisions de performance avec des moyens réduits, cette grande complexité, ainsi que le manque ou l'absence de spécification formelle, en limitent la qualité des résultats ou la faisabilité. Ainsi, les techniques de test par injection de trafic et observation des performances du système sont très largement utilisées.

Le test de performance est souvent motivé par le *benchmarking*, pratique qui vise à obtenir une mesure de performance comparable entre différentes alternatives techniques. Ces alternatives peuvent consister en différents paramétrages d'un même système, ou à des implantations partiellement ou totalement différentes du système. Dans le domaine des serveurs d'application Java EE, on peut ainsi être amené à chercher le meilleur paramétrage d'une machine virtuelle Java donnée (e.g. politique du *garbage collector*, taille du tas), ou à choisir entre différentes machines virtuelles Java. Il en va de même pour tous les autres éléments de l'application visée : serveur HTTP, servlets, type d'EJB, système de gestion de base de données (SGBD), etc.

Le benchmarking peut donc servir à optimiser un système, afin d'en obtenir les meilleures performances. Pour cela, l'équipe de test doit effectuer des campagnes de test sur plusieurs paramétrages ou configurations du système, et comparer les résultats. En réalité, ce principe d'optimisation est incontournable dans toute pratique de benchmark. En effet, si le benchmarking vise à obtenir des mesures de performances significatives et effectivement comparables, encore faut-il que les différents éléments alternatifs comparés soient testés dans leur paramétrage optimal. Par exemple, pour choisir entre un SGBD A et un SGBD B, il est nécessaire de réaliser un benchmark de A et B dans leur paramétrage optimal pour l'application et le trafic de requêtes définis.

1.1.2 Optimisation

L'optimisation d'un système informatique consiste à améliorer son fonctionnement en regard d'un ou plusieurs critères quantitatifs. Dans le domaine des performances, il s'agit d'améliorer des indices mesurables tels que le temps de réponse, le débit de requêtes traités par seconde ou le nombre de clients simultanés. L'optimisation consiste alors à maximiser certaines métriques et à en minimiser d'autres.

L'optimisation des performances nécessite de modifier le système, afin d'évaluer différentes configurations ou différents paramétrages. Ces modifications s'intègrent dans un processus de *génération/évaluation* et de parcours d'espace de solutions, classique en informatique. Il est typiquement conduit par l'équipe de test, qui utilise son expertise en partie empirique afin de ne pas se retrouver submergée par la combinatoire énorme entre les différentes valeurs possibles des différents paramètres du système. Pour chaque paramétrage ou configuration jugé pertinent, un déploiement du système testé et une exécution du benchmark sont réalisés.

1.1.3 Problématique

L'optimisation des systèmes informatiques et le benchmarking sont des disciplines étroitement liées. Leur pratique implique l'expertise d'une équipe de test et des tâches répétitives et fastidieuses visant à chercher le paramétrage optimal du système testé. La problématique abordée dans cette thèse est celle de l'*automatisation* de l'optimisation et du benchmarking, que nous nommerons respectivement auto-optimisation et *self-benchmarking*. Ce dernier terme désigne l'exécution automatique d'une campagne de benchmark qui détermine de manière autonome le paramétrage donnant la meilleure performance.

Nous souhaitons fournir un cadre logiciel complet permettant de générer et d'appliquer des paramétrages à un système à optimiser, et de conduire des tests de performance entièrement automatisés. Pour ce faire, nous devons prendre en compte deux difficultés majeures :

- l'indépendance et l'adaptabilité vis-à-vis du système à optimiser ;
- le temps d'exécution "raisonnable", étant données la combinatoire énorme entre les valeurs des paramètres, et la durée d'exécution de chaque test.

1.2 Approche

1.2.1 Principes

Notre approche comporte deux volets. Le premier volet concerne l'architecture logicielle, afin de pouvoir assurer l'indépendance et l'adaptabilité aux différents systèmes à optimiser. Le second volet consiste à proposer un algorithme d'optimisation adapté aux contraintes d'utilisation réelles, notamment en terme de temps d'exécution global.

1.2.2 Composants et boucles de contrôle autonome

Nous proposons une approche à base de composants, héritée de travaux dans le domaine de l'*autonomic computing*. En tout état de cause, l'auto-optimisation est l'une des propriétés clés des systèmes auto-administrables, et il est naturel de s'y référer. De plus, l'essence de l'*autonomic computing* est justement la méthodologie de construction des propriétés autonomes, notamment avec le principe de boucle de rétroaction observation-analyse-réaction.

Techniquement, nous partons d'un canevas logiciel à composants dédié au test de performance (CLIF [Dil09]), et d'une extension permettant de piloter de manière autonome

le niveau de charge injecté jusqu'à atteinte de critères de saturation (Selfbench). Ces éléments logiciels nous fournissent la base pour obtenir le benchmark du système, à laquelle nous intégrons nos composants de paramétrage (configurateurs, démarreurs) et de contrôle de l'optimisation. Pour réaliser cette intégration, nous avons été amenés à traiter la question de la coordination entre plusieurs boucles de contrôle, par une solution faiblement couplée à base de communication asynchrone de type publication-souscription.

1.2.3 Un algorithme adapté

Nous proposons un algorithme d'optimisation basé sur un principe de recherche dichotomique et de parcours « en croix » de l'espace des solutions. Sous des hypothèse s que nous justifions, l'algorithme permet de diminuer de manière très significative le nombre de solutions à tester, et donc le temps total d'exécution de l'auto-optimisation. De plus, une heuristique sur l'ordre des paramètres est proposée afin de contribuer à réduire encore ce temps.

Afin de réduire la durée du processus de *self-benchmarking*, le nombre de paramètres et le nombre de leurs valeurs doivent être aussi réduits que possible. En complément, une granularité de valeur est fixée pour chaque paramètre, afin de réduire le nombre de tests, moyennant une approximation de la valeur optimale des paramètres jugée satisfaisante. L'expertise initiale reste donc très importante afin d'éviter une durée totale d'exécution rédhibitoire.

Par ailleurs, grâce à une représentation abstraite du problème d'optimisation, cet algorithme est indépendant du système à optimiser.

1.2.4 Validation

Pour la validation de notre travail, nous avons réalisé deux campagnes de test et d'optimisation sur deux applications WEB qui diffèrent significativement quant à leur fonctionnement, même si elles sont portées par des serveurs d'applications identiques. La première est une application exemple de type achat en ligne, déployée sur le serveur d'applications Java EE JOnAS. La seconde est le benchmark RUBIS, une application de type enchères en ligne, également déployée sur JOnAS.

1.3 Organisation du document

Le manuscrit se divise en trois parties : état de l'art en trois chapitres, contribution en deux chapitres et validation en un seul chapitre, plus une introduction générale et une

conclusion générale.

Le chapitre 1 est la présente introduction du mémoire.

Le chapitre 2 fait un rappel des différentes notions liées à l'évaluation de performances en informatique ainsi que les différentes méthodes employées dans ce domaine. Ce chapitre se concentre également sur la pratique du *testing* et son application dans le domaine du *(benchmarking)*.

Le chapitre 3 est dédié à l'optimisation, ses notions et principes, les approches principales et méthodes de résolution. Il aborde aussi l'optimisation en informatique du point de vue des niveaux et de paramètres d'optimisation. Ce même chapitre décrit quelques approches métiers phares d'optimisation des systèmes informatiques.

Le chapitre 4 fait un rappel des différentes notions et principes du Calcul Autonome *(Autonomic Computing)*. Il introduit les approches autonomiques de l'auto-optimisation, nous y avons consacrons une section au modèle architectural de composants que l'on a utilisé pour la mise en place de notre dispositif d'auto-optimisation. Un exemple des systèmes autonomes est donné dans ce chapitre, il s'agit d'un système d'injection de charge autorégulée que nous étendons justement dans le cadre de ce travail par l'ajout des aspects d'auto-optimisation.

Le chapitre 5 présente l'aspect algorithmique de notre contribution concernant l'auto-optimisation qui se traduit dans notre contexte par la recherche du paramétrage optimal du système sous test. Notre algorithme se décompose en deux parties. La première concerne le niveau global, c'est à dire le contrôle de l'évolution de la valeur de l'indicateur de performance, en fonction des valeurs des paramètres de réglage du système. La deuxième partie est relative à la recherche de l'optimum lorsqu'un seul paramètre est modifié. Nous détaillons dans ce chapitre également les réglages à réaliser sur l'algorithme et nous indiquons comment cet algorithme est mis en œuvre dans l'architecture de notre application d'auto-optimisation.

Le chapitre 6 décrit l'architecture logicielle de la plate-forme de Self-benchmarking que nous proposons, et détaille les différents composants Fractal qui la composent : l'optimiseur, les configurateurs et les démarreurs. Ce même chapitre aborde également la problématique de la communication entre les composants contrôleurs, les différentes approches possibles, et la mise en place de la solution adoptée.

Le chapitre 7 présente deux séries d'expérimentations réalisées pour la validation de notre dispositif d'auto-optimisation. La première expérimentation sert à trouver le meilleur paramétrage d'une application WEB de type achat en ligne simplifiée (application exemple `MyStore`) s'exécutant sur le serveur d'application `JOnAS`. La deuxième expérimentation concerne une application Java EE dont les trois tiers sont répartis sur des machines dis-

tinctes. Nous présentons et analysons dans ce chapitre les résultats obtenus ainsi que les gains de performances obtenus par rapport à des configurations par défaut.

Le chapitre 8 dresse à la fois un bilan des résultats obtenus à l'occasion de ce travail, une évaluation de ces derniers et aussi une projection dans le futur sous forme de quelques perspectives d'améliorations et d'extensions possibles de notre travail.

Première partie

État de l'art

Chapitre 2

Évaluation de performances et Benchmarking

Nous introduisons dans ce chapitre les différentes notions liées à l'évaluation de performances en informatique. Nous rappelons les principales méthodes employées dans ce domaine. Nous nous concentrons ensuite sur la démarche de test et son application dans les approches par comparaison (benchmarking).

Sommaire

La croissance rapide des systèmes informatiques actuels, également de nature de plus en plus complexe, entraîne des problèmes des performances parfois critiques, qui peuvent aller d'une simple dégradation de performance sous des charges plus ou moins importantes ou une diminution de la qualité de service, jusqu'à un effondrement total du système. L'évaluation des performances, bien qu'elle soit une discipline universelle, est devenue en informatique une discipline à part entière avec un ensemble de méthodes spécifiques. La figure 2.1 résume les différentes approches utilisées en évaluation de performance.

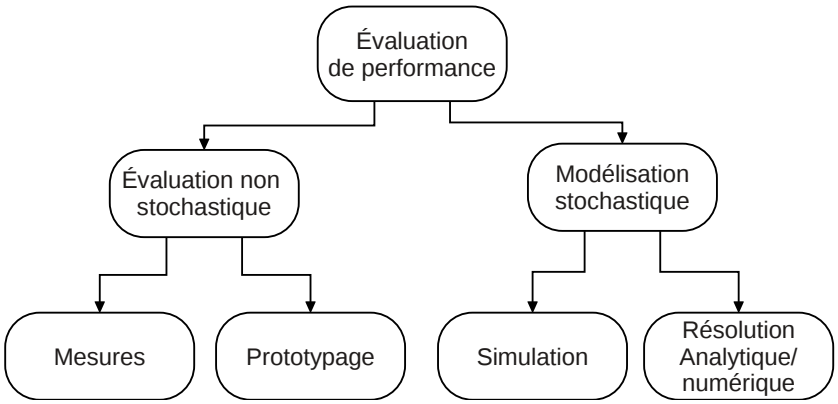

FIGURE 2.1: Les méthodes d'évaluation de performance

L'évaluation de performances s'applique à la fois avant la constitution du système et sur des systèmes existants. Dans le premier cas, on dispose d'une description du système à construire pour lequel on élabore un modèle mathématisé (voir section 2.2) à partir duquel on peut calculer des indicateurs de comportement et par conséquent prévoir les performances du système concerné. Dans le deuxième cas, on distingue trois cas de figures :

- On peut abstraire le système par un modèle mathématisé. on est alors ramené au premier cas.
- La modélisation du système est trop complexe ou impossible parce qu'on ne dispose pas d'assez d'informations. Dans ce cas, on est réduit à des mesures sur le système. On parle alors de *test de performances*.
- Il est également possible de construire un système simplifié, un prototype ou un simulateur, pour le faire fonctionner en lieu et place du système final. On fournit alors à ce système des stimuli qui représentent les données d'entrée du système et l'on observe le comportement du prototype.

2.1 Évaluation de performances des systèmes informatiques

Dans ce mémoire, nous nous cantonnons aux aspects scientifiques et techniques de la notion de performance, en écartant par exemple les problèmes de coût ou de consommation énergétique. En informatique, les performances d'un système, d'une application ou d'un service sont les caractéristiques de fonctionnement de ce dernier relatives à un ensemble de critères donnés.

Ces critères de performances parfois appelés indices de performances sont de natures

diverses. On distingue en particulier les indices suivants :

1. le débit souvent noté χ : nombre de tâches, requêtes, traitements, réalisés pendant une période de temps donnée ;
2. le temps de réponse souvent noté R : temps de traitement d'une tâche. Par exemple dans les réseaux de communication, le temps de réponse de bout en bout est le délai d'acheminement des paquets à travers le réseau.
3. le nombre de « clients » souvent noté Q ou N : nombre de tâches présentes dans le système ;
4. le taux d'utilisation d'une ressource r souvent noté $U(r)$: le taux d'utilisation $U(r)$ est soit la proportion de temps pendant lequel r est utilisée sur une période de temps donnée (par exemple : taux d'utilisation d'une uité centrale de traitement (CPU), soit la proportion de ressources r utilisées dans un ensemble R de telles ressources (par exemple le taux d'utilisation d'un groupe de processus légers (`Threads`) d'un serveur WEB).

Pour chacun de ces paramètres, on peut s'intéresser à sa valeur à un instant de temps donné t, à sa valeur moyenne sur une durée donnée (ainsi qu'à d'autres statistiques comme la variance), ou, lorsqu'elle existe, à leur valeur limite au cours du temps, aussi appelée valeur à l'équilibre.

Chacun de ces paramètres possède une signification dans pratiquement tous les domaines de l'informatique. On peut par exemple s'intéresser aux performances d'un serveur en matière de débit, au temps de réponse d'une application de réservation en ligne, ou encore à la comparaison de plusieurs compilateurs d'une machine multiprocesseurs.

L'évaluation de performances a donné lieu à de nombreux travaux d'ingénierie dont l'ouvrage de Raj Jain [Jai91] est une des références du domaine. Jain identifie dix étapes importantes pour mener à bien toute étude de performances d'un système informatique. Nous avons résumé ces étapes dans les points suivants :

1. Fixer les objectifs de l'évaluation et définir les contours du système étudié. Par exemple, pour deux CPUs (*Central Processing Unit*), si l'objectif est d'étudier l'impact de leurs performances sur le temps d'exécution de plusieurs programmes, le système pourrait consister en le système à temps partagé et les résultats de l'étude pourraient dépendre significativement d'autres composants que les CPUs. Par contre si les deux CPUs sont identiques à l'ALU (*Arithmetic and Logical Unit*) près, le CPU est considéré comme système et les composants au sein du CPU sont considérés comme parties du système.
2. Identifier la liste des fonctionnalités ou services offerts par le système ainsi que les

résultats qu'ils fournissent. Par exemple, un réseau informatique permet la transmission de paquets entre différents destinataires ; un système de gestion de base de données (SGBD) répond à des requêtes ; un processeur traite différentes instructions.

3. Choisir les indicateurs de performances que l'on veut obtenir en fonction des objectifs de l'étude.

4. Identifier la liste des paramètres du système qui affectent les performances étudiées. On distingue ici les paramètres liés au système (CPU, mémoire, réseau, ...) et les paramètres liés à la charge de travail demandée au système (nombre ou vitesse d'arrivée des requêtes, ...). Ce point est important. Nous verrons dans le reste de ce mémoire qu'une connaissance approfondie du système testé est nécessaire pour identifier l'ensemble de ces points. Ces paramètres sont appelés paramètres d'optimisation (*tunning parameters*).

5. Sélectionner parmi les paramètres, ceux, en nombre réduit, sur lesquels on va jouer au cours de l'évaluation (Jain les nomme « facteurs d'évaluation »).

6. Choisir la méthode d'évaluation. Il existe trois méthodes : analyse et simulation qui se basent sur une modélisation du système étudié, test qui se base sur des mesures directes sur le système testé.

7. Définir les caractéristiques de la charge sous laquelle le système va être étudié. Ce point est important du fait que si la charge n'est pas adaptée au système (charge de référence, montée de la charge dans le temps, etc.) on ne peut obtenir que des résultats peu significatifs et parfois inutiles.

8. Calculer les indicateurs de performances. Il s'agit donc de calculer les différents indices de performances à partir du modèle défini (analyse ou simulation) ou bien de mesurer ces métriques directement sur le système testé via des sondes mises sur les éléments concernés (JVM, CPU, réseaux, .etc.),

9. Analyser et interpréter les résultats. Pour les méthodes de simulation et de mesures, l'analyse des résultats doit être précédée par une phase de traitements statistiques,

10. Exprimer de manière synthétique les résultats et les analyses. Cette étape est nécessaire lorsqu'il s'agit d'exploiter les résultats de l'étude en vue de prendre des décisions.

2.2 Évaluation de performance basée sur les modèles stochastiques

Comme pour les autres catégories de méthodes (voir figure 2.1), la démarche consiste à décrire le système étudié, choisir et définir des indicateurs de performance, obtenir leurs valeurs et analyser les résultats obtenus. Mais, dans les approches par modélisation stochastique, les indices de performance sont obtenus comme résultats de calculs.

Au niveau des modèles, on distingue deux catégories. D'une part, ce qu'on appelle des modèles fondamentaux ou de bas niveau, qui sont des systèmes stochastiques à évènements discrets (SSEV). Ils décrivent les systèmes avec des états et des transitions entre ces états. Les durées de séjour dans les états suivent des distributions de probabilité dont les lois sont définies dans le modèle. Parmi ces modèles, les chaînes de Markov sont le modèle le plus employé. Dans une chaîne de Markov, les durées de séjour dans les états suivent des lois exponentielles négatives (temps continu) ou géométriques (temps discret). Cette propriété est effectivement vérifiée ou représente une approximation satisfaisante dans de nombreux cas et les chaînes de Markov permettent d'obtenir des résultats pertinents depuis bientôt un siècle, pas uniquement dans le domaine de l'informatique.

Les modèles SSEV sont très complexes à définir en partant d'un système technologique comme un système informatique ou un atelier de production. Le nombre d'états et de transitions peut être en effet extrêmement grand (10^{12} états) et la description très laborieuse. On emploie donc souvent des modèles de plus haut niveau sémantique, *i.e.* à plus fort pouvoir d'expression. Parmi ces modèles, nous citerons les files d'attente, les réseaux de files d'attente [Bay00], les réseaux de Petri stochastiques [ABC+95, HM01] les algèbres de processus stochastiques. Dans chaque cas, on définit des algorithmes de traduction automatique de ces modèles vers des SSEV. Les risques d'erreur de modélisation sont ainsi réduits et les concepteurs de systèmes sont plus à même de définir les éléments du modèle (structure et paramètres).

Lorsque le modèle est défini, on met en œuvre des méthodes de résolution du problème mathématisé, on parle de « résolution du modèle », pour calculer les indices de performance que l'on a défini.

Notons enfin qu'il est possible d'employer à la fois une méthode fondée sur un modèle stochastique tout en utilisant des mesures sur système effectif. Dans ce cas, les mesures permettent de paramétrer, caractériser, le modèle avant de calculer les indices de performance.

2.3 Évaluation de performances basée sur les mesures

La mesure de performances est l'une des méthodes les plus utilisées, si ce n'est la plus utilisée, pour évaluer les performances des systèmes existants. La procédure qui permet d'obtenir ces mesures s'appelle un *test de performances* [Bar04]. On distingue plusieurs catégories de test. Les plus pratiqués sont indiqués dans le tableau 2.1.

2.3.1 Activités de base d'un test de performances

La tâche principale de toute campagne de test, quel qu'en soit le type, est de fournir un ensemble d'informations concernant le fonctionnement du système testé afin d'être en mesure d'aider les testeurs ou les chefs de projets de validation des tests à prendre des décisions éclairées relatives à la qualité globale de l'application testée. Les tests de performances ont ainsi tendance à se concentrer sur l'identification des goulets d'étranglement dans un système, l'amélioration de son réglage, ou encore l'aide à établir une base de référence pour les essais futurs En outre, les résultats des tests et leur analyse permettront d'estimer la configuration matérielle requise pour un usage de l'application en production.

Il existe de nombreux outils de test (voir section 2.3.3) qui couvrent à degrés divers, l'ensemble des catégories de tests présentées ci-dessus. Tous ces outils présentent de nombreux points communs. Inspirés sans doute par l'approche de Jain et *al.* [MFB+07] listent un ensemble d'étapes pour la bonne pratique des tests des performances dans le domaine des applications web. Ces étapes sont résumées dans les points suivants :

1. Identification de l'environnement du test : une connaissance approfondie de l'environnement physique du test (configurations matérielles, logicielles et réseau) et de l'environnement de production permet la conception des tests plus efficaces et une identification plus rapide des objectifs du test dès le début du projet,
2. Identification des critères de performances. Il s'agit de définir les objectifs ainsi que les contraintes concernant le temps de réponse, le débit et l'usage des ressources. En général, le temps de réponse est une préoccupation de l'utilisateur, le débit est une préoccupation de l'entreprise et l'usage de ressources est une préoccupation du gestionnaire du système.
3. Plans et scénarios de test. Il s'agit de définir les principaux scénarios, déterminer les comportements des utilisateurs et simuler la variabilité parmi ces utilisateurs, de définir les données de test, et de définir les mesures qui doivent êtres collectées.

Type de test	Objectif	Remarques
Test de performances	S'assurer du bon fonctionnement de l'application en utilisation réelle. Obtenir un référentiel de temps de réponse de l'application.	On ne cherche pas à atteindre les limites de l'application ou de l'infrastructure mais uniquement à mesurer les temps de réponse moyens lors d'une utilisation normale (fonctions utilisées, nombre d'utilisateurs).
Test de stress	Ce test permet de simuler un pic de charge réel et de détecter certaines failles (fuite de mémoire, interblocage, ...) en poussant les conditions de tests au-delà de l'utilisation normale de l'application.	La détection de ces failles n'est possible que dans un contexte de stress de l'application. En utilisation normale ces failles sont souvent invisibles.
Test d'endurance	S'assurer du bon fonctionnement de l'application en utilisation réelle et sur la durée. Utile lorsqu'il y a un engagement sur le taux de disponibilité de l'application.	Consiste à faire fonctionner l'application à un rythme normal, pendant une période de temps relativement longue pour voir si une dégradation des temps de réponse, une fuite mémoire ou un blocage n'apparaît pas au bout d'un certain temps.
Test de robustesse	S'assurer du bon fonctionnement de l'application avec une charge importante sur la durée.	Nécessite qu'il y ait eu au préalable un test d'endurance complet
Test de montée en charge	Observer les limites que peut supporter l'application ou un de ses composants.	On augmente le nombre d'utilisateurs simulés de manière régulière et on recherche le « point de rupture », c'est-à-dire le moment où l'application ne répond plus dans le temps maximum accepté par l'utilisateur.

TABLE 2.1: Différents types de test et leurs objectifs

4. Configuration de l'environnement du test. Il s'agit de préparer l'environnement de test, les outils et les ressources nécessaires pour exécuter chaque plan de test, et de s'assurer que l'environnement de test est instrumenté pour la surveillance des ressources *monitoring*.
5. Mettre en œuvre la conception du test. Il faut produire les artefacts du test de performance conformément à la conception du test.
6. Exécution et suivi du test : Valider le test, les données de test, et collecter les résultats. Après validation, exécuter les tests en les surveillant ainsi que l'environnement de test.
7. Analyse des résultats, rédaction des rapports et re-test. Il s'agit de la consolidation et du partage des résultats du test, de l'analyse des données individuellement et avec l'équipe inter-fonctionnelle, de la re-définition des priorités des tests et si nécessaire de leur re-exécution. Lorsque toutes les métriques sont dans les limites acceptées, qu'aucun des seuils fixés n'est violé, et que toutes les informations souhaitées ont été recueillies, la fin du test est atteinte.

2.3.2 Plan et scénarios de test

Toute campagne de test doit commencer par l'élaboration d'un plan de test. On y trouve :

- un résumé du projet ;
- l'architecture technique et logicielle de l'application à tester ;
- les objectifs du test ;
- le type de test ;
- le modèle de charge ;
- les scénarios du test ;
- le plan d'exécution des tests.

Quelque soit le type de test à effectuer, la sélection de la charge est une partie cruciale de toute étude de performances. Un mauvais choix de la charge peut mener à des résultats non significatifs, incohérents ou erronées. Le niveau de la charge est un facteur important. Selon l'objectif du test, une charge doit mener le système à ses capacités complètes (dans le meilleur cas) ou au-delà de ces capacités (mauvais cas) ou enfin au meilleur niveau de charge observé dans un cas d'usage réel du système (cas typique). Enfin, une bonne charge doit permettre de reproduire facilement les résultats sans écart significatifs. Les charges qui font des demandes de ressources fortement aléatoires sont généralement moins souhaitables car elles nécessitent beaucoup plus de tests et par conséquent plus de temps

pour avoir des résultats significatifs.

Dans de nombreux outils de test, on génère la charge sous la forme d'*utilisateurs virtuels* (*Virtual User*, VUsers). Un VUser est la reproduction du comportement d'un utilisateur réel de l'application (login, ouverture d'une session, connexion à une BD, traitement métier, etc.). Le comportement d'un VUser est défini dans un scénario (*Behavior*). Le nombre initial de VUsers et la stratégie de montée en charge sont définis dans le profil de charge.

Un bon plan est celui qui représente le plus possible le comportement d'un utilisateur réel du système. Une définition de cette représentativité peut être rédigée dans des scripts sous forme d'un ou plusieurs scénrios. Un scénario reflète le plus fidèlement possible le comportement d'un utilisateur réel de l'application ; il doit tenir en compte les points suivant :

1. le taux d'arrivée : le taux d'arrivée des requêtes doit être le même ou proportionnel à celui de l'application réelle.
2. la demande de ressources : la totalité des demandes en ressources (i.e. requêtes des clients) doit être similaire ou proportionnelle à celle de l'application réelle.
3. le profil d'usage de ressources : le séquencement et la fréquence avec lesquels les différentes ressources sont utilisées doivent être du même ordre que ceux des requêtes réelles.

Le comportement de l'utilisateur en fonction du temps est un autre facteur crucial dans la définition la charge. Les clients changent leurs comportement dans le temps et tout changement dans les performances du système les obligent à changer leur comportement d'usage de ressources du système. Un bon plan de test doit accorder une importance particulière à la définition du comportement des clients du système en fonction du temps.

L'impact d'éléments externes du système sous test peut aussi être plus ou moins fort ; par exemple la comparaison entre la vitesse de deux processeurs peut être complètement masquée à cause de l'impact des composants d'E/S.

Il est très important d'identifier les scénarios pertinents et de les pondérer. Pour définir ces scénarios on peut :

- récupérer des statistiques de l'utilisation de l'application si elles existent (par exemple le nombre quotidien ou mensuel de transactions métier, l'analyse des statistiques WEB (fichiers de connexions, *acces log*)).
- définir avec les utilisateurs les scénarios principaux d'utilisation (les fonctions les plus utilisées) de leur application.

Enfin, toute plate-forme de test de performances comporte généralement un générateur de charges et un système de sondes. Un système de génération de charge, appelé aussi

moteur de charge est un logiciel qui est capable de simuler les actions des utilisateurs et de ce fait générer la charge sur l'application. Ces actions sont définies dans des scripts automatisant les scénarios et valorisés sur un lot de données particulier. On peut donc distinguer deux catégories d'outils :

- Les simulateurs d'utilisateurs qui, essentiellement, font exécuter les scripts de test pour un grand nombre d'utilisateurs virtuels ;
- Les générateurs de données qui vont permettre de valoriser les scripts utilisés et ainsi assurer que chaque utilisateur virtuel n'effectue pas exactement la même opération ce qui n'aurait pas beaucoup de sens du point de vue des performances mesurées ensuite.

Un système de sondes est placé au niveau du système sous test. Ces sondes permettent de remonter des mesures dont l'objet est de savoir comment réagissent individuellement des composantes du système (mémoire, CPU, réseau, disques .etc.).

2.3.3 Outils de test

Il est évident qu'une démarche de test ne peut être conduite qu'avec l'aide d'outils logiciels adaptés. Il existe de nombreux outils de test. Citons parmi les plus connus, sinon les plus employés :

- Outils commerciaux :
  - LoadRunner (HP), leader du marché ;
  - Silk-performer (Micro Focus) ;
  - Load Testing (Oracle) ;
  - IBM Rational Performance Tester (IBM) ;

  Outils open source :
  - CLIF (France Télécom, `http://clif.ow2.org/`) ;
  - JMeter (Apache fundation, `http://jmeter.apache.org/`) ;
  - Grinder (`http://grinder.sourceforge.net/`)

Nous renvoyons le lecteur aux descriptions détaillées de chaque outil pour de plus amples informations.

On peut d'une part distinguer les outils « open source » et les outils commerciaux, en général très coûteux. Les outils open source sont fréquemment des solutions *ad hoc* à des problèmes spécifiques d'évaluation de performance. CLIF [Dil09] est une exception notable à cette situation. C'est l'outil que nous utiliserons dans notre étude et nous le présentons en détails dans le chapitre 4.

Les outils commerciaux proposent des palettes de fonctions plus étendues. Cependant, peu d'entre-eux disposent, à la fois, des fonctionnalités d'injection de charge, de

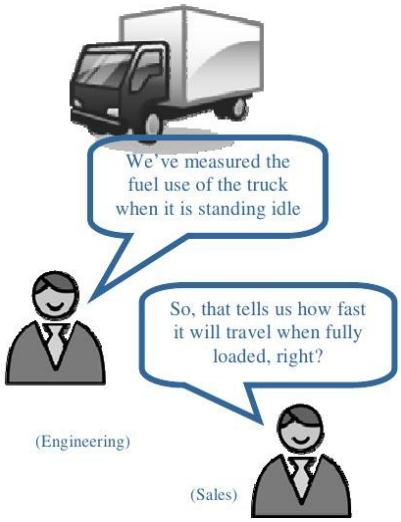

FIGURE 2.2: Un benchmark doit être approprié (reproduit de [Hup09])

surveillance d'usage des ressources et de profilage de l'exécution.

2.4 Benchmarking

2.4.1 Définition

Le terme même de *benchmarking* a donné lieu à de multiples définitions, dans la sphère commerciale comme dans la sphère technique et scientifique. Par exemple, David Kerns, directeur exécutif de Xerox, donne la définition suivante du Benchmarking : « *the continuous process of measuring products, services, and practices against the toughest competitors or those recognized as industry leaders* » (le process continu de la mesure des produits, services et pratiques par rapport aux concurrents les plus directs ou aux *leaders* du marché).

Plus techniquement, le benchmarking est la pratique de tests de performances en vue d'une *comparaison*

- entre plusieurs alternatives de conception du même système ;
- ou entre différentes configurations d'un système donné ;
- ou encore entre deux ou plusieurs systèmes de même type ou avec les mêmes fonctionnalités (Serveurs d'applications, systèmes de disques, systèmes de gestion de bases de données, etc.).

Le benchmarking consiste consiste donc à *comparer* plusieurs systèmes ou étalonner un système pour un fonctionnement donné vis à vis d'un système de référence, selon des

critères quantifiés. Le mot anglais benchmarking, que l'on peut traduire par évaluation, étalonnage, est en général conservé tel quel dans la communauté francophone de l'évaluation et du test de performance, du test et des systèmes informatiques. Nous suivrons cet usage.

En pratique, concevoir puis pratiquer un benchmark pertinent, relève, comme le souligne [Hup09], de l'art de l'ingénieur comme de la science. La figure 2.2 tirée d'une présentation de Huppler résume deux écueils classiques du benchmarking : mesurer des grandeurs non significatives par rapport au problème posé, tirer des conclusions fausses de résultats de benchmarks.

Notons que pour comparer différents systèmes de manière cohérente, il est nécessaire de *régler* chacun d'entre-eux de sorte qu'il fonctionne avec les meilleures performances relativement aux critères retenus pour le benchmark. Ainsi, toute procédure de benchmarking implique l'*optimisation* des systèmes concernés, qui se traduit par un *réglage des configurations et de leurs paramètres*.

2.4.2 Exemples de benchmarks

Il existe de très nombreux benchmarks dans le domaine de l'informatique. Deux organismes élaborent des séries de benchmarks, TPC et SPEC. Par ailleurs, plusieurs communautés ont défini des benchmarks spécifiques qui font en quelque sorte référence dans leur domaine, sans pour autant être décrits de manière totalement formelle.

2.4.2.1 TPC

Le Transaction Processing Performance Council (`http://www.tpc.org/`) est un organisme a but non lucratif dont la mission est de définir des benchmakrs de performances de systèmes informatiques transactionnels et de publier les résultats de ces benchmarks. Les origines de TPC remontent au début des années 80, où les constructeurs avaient chacun leur propre système de mesure de puissance de machine, ce qui ne permettait pas la comparaison entre ces derniers [Gra85]. Le TPC en tant qu'organisme a été créé en 1988 par huit compagnies. Les premiers résultats TPC-A ont été annoncés en juillet 1990. Par la suite d'autres benchmarks ont été mis au point pour mesurer d'autres points de vue, comme par exemple les performances des systèmes client-serveur. L'originalité des classements du TPC est non seulement de fournir des tableaux de puissance relative, mais aussi des tableaux faisant intervenir le ratio performance/coût [Cou]. Les acteurs du marché publient régulièrement les records atteints lors des passations des benchmarks TPC, en particulier pour le plus utilisé d'entre-eux, TPC-C.

2.4.2.2 Spec

La *Standard Performance Evaluation Corporation* (`http://www.spec.org/`), est une organisation à but non lucratif, dont la vocation est de produire, créer, maintenir et approuver un ensemble standard de benchmarks. SPEC até crée en 1988 et ses benchmarks sont largement utilisés pour l'évaluation des systèmes informatiques. Les benchmarks de SPEC concernent différents systèmes ou sous-systèmes informatiques tel que : Benchmarks de CPU, Benchmarks Cient/Serveur, Benchmarks Serveurs de mail, Benchmarks SIP, Benchmark SOAP et Benchmark énergie. La description des différentes benchmarks, de leurs plateformes ainsi que les résultats sont publiées sur le site web de la corporation [SPE].

2.4.2.3 Benchmarks de type cas d'étude

Il s'agit de systèmes ou applications ou ensemble d'applications, dans un domaine donné, qui, sans être complètement définis, correspondent à des cas d'étude (*use case*) fréquemment rencontrés dans ce secteur. Dans le contexte des serveurs WEB par exemple, nous citerons les architectures Apache-PHP-MySQL (LAMP ou WAMP). Dans le domaine des serveurs d'application, on associe en général un serveur à une application déployée d'un type donné (e-commerce, consultation d'horaires, réservation en ligne, etc). Deux exemples caractéristiques et largement utilisés, MyStore (e-commerce) et RUBIS (enchères en ligne) toutes deux déployées sur le serveur Java EE JOnAS seront utilisés dans nos expérimentations (voir le chapitre 7).

2.5 Conclusion

Dans ce chapitre, nous avons réalisé un état de l'art relatif à l'évaluation de performance. Nous avons vu qu'il y a trois méthodes possibles pour évaluer les performances des systèmes informatiques. Nous avons présenté succinctement ce que c'est l'évaluation basée sur la modélisation. Nous avons vu en détails les principes de l'évaluation basée sur des mesures réalisées sur le système en fonctionnement. C'est cette approche que nous suivons dans ce mémoire. Nous avons également introduit le concept de benchmarking. Dans le chapitre suivant, nous présentons une brève synthèse sur les problèmes d'optimisation.

Chapitre 3

Optimisation des performances

Nous rappelons dans ce chapitre les notions et principes de l'optimisation. nous introduisons les principales approches et méthodes de résolution. Nous abordons ensuite l'optimisation en informatique et nous dressons une liste des paramètres et des niveaux d'optimisation. Nous présentons également quelques approches métiers d'optimisation des systèmes informatiques.

Sommaire

Le mot optimisation est d'origine latine, dérivé du mot *Optimum* qui signifie le meilleur. L'optimisation est originalement une branche des mathématiques qui sert à analyser et à résoudre analytiquement ou numériquement les problèmes qui consistent à déterminer le meilleur élément d'un ensemble, au sens d'un critère quantitatif donné.

De nos jours, les applications de l'optimisation sont innombrables. Chaque processus qu'il soit en industrie, en économie en sciences ou en ingénierie est potentiellement

optimisable, et peut donc tirer profit des méthodes d'optimisation.

Aujourd'hui il n'y a pas d'entreprise qui ne soit impliquée dans la résolution de problèmes d'optimisation [Tal09]. En effet, de nombreuses applications difficiles dans les sciences et l'industrie peuvent être formulées comme des problèmes d'optimisation. Dans les réseaux de communication, et afin d'optimiser les coûts et la qualité de service il faut déterminer les configurations ou les topologies. Dans les systèmes de production il existe plusieurs façons d'ordonnancer les tâches afin d'optimiser le temps de fabrication. En sciences biologiques, il existe plusieurs façons de définir la forme 3D d'un protéine pour optimiser l'énergie.

En sciences de l'ordinateur, et depuis l'apparition des premières machines *calculateur* l'optimisation est présente partout et sur tout le niveau : optimisation du matériel (i.e. circuits, cartes mère, mémoire, processeur, interfaces réseaux, .etc) ou optimisation logicielle (i.e. algorithmes, langages, *frame-works*, .etc).

L'accroissement de la puissance des machines de dernière génération (depuis pratiquement le début des années 70) a été accompagnée par une évolution dans le monde de conception et de développement des systèmes informatiques. Ces systèmes sont devenus distribués avec plus de capacité de traitement. Le *World Wide Web* est un des exemples de cette *(r)*évolution.

Dans de tels systèmes de plus en plus complexes (plates-forme, architectures et protocoles plus complexes qu'auparavant), l'ouverture partielle ou totale de ces systèmes et applications sur les réseaux, et une capacité assez limitée d'administration, de configuration et de gestion de ces application dans un contexte en mouvement et en évolution permanents et soumis à des charges non contrôlables et non maitrisée.

L'optimisation d'un système informatique quelconque concerne essentiellement son fonctionnement, elle peut signifie l'assurance *maximale* de la disponibilité du service censé être offert par le système surtout dans des situations critiques, elle peut signifie aussi l'amélioration du temps de réponse du système, comme elle peut signifie la maximisation de la capacité du traitement parallèle du système en question.

Dans la pratique, quand on parle d'optimisation d'un système d'information, on entend par cela l'amélioration de ses performances. Les performances d'un systèmes sont mesurées (ou calculées) en terme d'un indice de l'état actuel du système (i.e. débit, temps de réponses, nombre de clients, nombre de requêtes traitées , .etc) 2.1.

La complexité des systèmes informatiques rend difficile et complexe la tâche d'amélioration de performances de ces systèmes. De toutes manières l'optimisation de performances est un travail quotidien concerne pratiquement tout les métiers de l'informatique(développeurs, intégrateurs, responsables de mise en production, administrateurs

système ou réseaux, .etc) et touche plusieurs niveaux de l'application en question et cela tout au long de son cycle de vie. Nous consacrons une section à l'optimisation en informatique 3.2 et présentons trois approches métier d'optimisation de systèmes informatique dans la section 3.3 avant de conclure ce chapitre par le concept d'auto-optimisation des systèmes informatiques.

3.1 Principes et méthodes

3.1.1 Principes

Définition 3.1.1 *Optimiser la fonction f sur $\mathbb{A}$, consiste à trouver l'élément x_0 de $\mathbb{A}$ tel que pour tout élément x de $\mathbb{A}$, $f(x)$ est majorée par $f(x_0)$ (s'il s'agit d'une maximisation) et minorée par $f(x_0)$ (s'il s'agit d'une minimisation). De manière formelle, nous avons donc pour un problème de maximisation :*

$$f : A \to \mathbb{R} \; \exists ? x_0 \in \mathbb{A}, \; \forall x \in A, \; f(x_0) \geq f(x) \tag{3.1}$$

et pour un problème de minimisation :

$$f : A \to \mathbb{R} \; \exists ? x_0 \in \mathbb{A}, \; \forall x \in A, \; f(x_0) \leq f(x) \tag{3.2}$$

La fonction f est appelée fonction objectif *et les éléments de l'ensemble $\mathbb{A}$ sont appelés* solutions admissibles. *L'élément $f(x_0)$ est appelé* Optimum *ou solution optimale. Lorsque l'ensemble $\mathbb{A} \subset \mathbb{R}^n$, on parle d'optimisation* multivariables.

Nota : Un problème d'optimisation combinatoire est susceptible d'avoir plusieurs solutions optimales.

Définition 3.1.2 (Optimisation multiobjectifs) *Lorsque la fonction objectif n'associe pas une valeur numérique, mais un point d'un ensemble c.à.d. elle est associée à un vecteur, dans ce cas on parle de l'optimisation multiobjectifs. Le but est alors d'optimiser simultanément l'ensemble des composantes de ce vecteur.*

On peut aussi voir l'optimisation multiobjectif comme un ensemble de problèmes d'optimisation portant sur les mêmes paramètres, ayant des objectifs éventuellement contradictoires, et que l'on cherche à résoudre au mieux.

Nous n'abordons pas le sujet de l'optimisation multiobjectifs dans le présent manuscrit, nous nous contentons donc dans le reste des sections par l'étude des modèles et méthodes

d'optimisation ayant un seul objectif d'optimisation. La section suivante montre les deux méthodes de résolution possibles pour tout problème d'optimisation.

3.1.2 Méthodes de résolution des problèmes d'optimisation

3.1.2.1 Résolution exacte

C'est des méthodes *déterministes* permettant de résoudre certains problèmes d'une manière exacte en un temps fini. Ces méthodes d'optimisation explorent de façon systématique l'espace de recherche. Souvent on parle des méthodes arborescentes car elles se basent sur un parcours d'arbre afin de trouver la solution optimale, les nœuds de l'arbre sont les affectations des valeurs des variables du problème.

Ces méthodes nécessitent généralement un certain nombre de caractéristiques de la fonction objectif, comme la stricte convexité, la continuité ou encore la dérivabilité. On peut citer comme exemple de méthodes de résolution de ce type de fonction : la programmation linéaire, quadratique ou dynamique, la méthode du gradient et la méthode de Newton.

Parmi les méthodes exactes, on trouve la plupart des méthodes traditionnelles (développées depuis une trentaine d'années) telles les techniques de séparation et évaluation *(branch and bound)*, ou les algorithmes avec retour en arrière *(backtracking)*. Les méthodes de résolution à base de solveurs de contraintes font partie de cette catégorie de méthodes, car la majorité des solveurs de contraintes implémente des algorithmes de type branch and bound et de backtracking lors de la construction de l'arbre de solutions. Les méthodes exactes permettent de trouver des solutions optimales pour des problèmes de taille raisonnable. Dès que la taille du problème soit plus grande le temps de calcul nécessaire pour trouver une solution risque d'augmenter exponentiellement et donc ces méthodes montrent des difficultés avec les applications de taille importante.

Notons en fin, qu'il existe des logiciels génériques tel que : AMPL [FGK02] qui est un langage de modélisation pour la programmation mathématique qui simplifie la modélisation et la formulation des problèmes d'optimisation compliqués, la résolution de ces modèles peut être faite à l'aide de plusieurs logiciels tel que : CPLEX [IBMa] qui est un solveur d'optimisation d'IBM pour la programmation linéaire mixte et quadratique, LINDO [LIN] qui est un logiciel d'optimisation pour la programmation en nombres entiers, linéaire, non-linéaire, stochastique et globale, MPL [Max] et XPRESS [FIC] qui est l'un des logiciels privés de la programmation linéaire et en nombres entiers les plus utilisés sur le marché.

3.1.2.2 Résolution approchée

La modélisation des problèmes réels n'est pas toujours une tâche facile et parfois impossible. D'autres problèmes appelés difficiles par exemple dans le cas de la non convexité stricte (multimodalité), l'existence de discontinuités, la non dérivabilité de la fonction ou la présence de bruit par exemple.Ou des problème ayant une taille de solutions énorme, et donc leur résolution par une méthodes déterministe prend énormément de temps. Dans certains problèmes l'optimalité n'est pas primordial du moment où on obtient une solution *satisfaisante*. Dans ces cas de problèmes une classe de méthodes a vu le jours appelée *heuristiques*.

Définition 3.1.3 (Heuristique) *Une heuristique est une méthode qui permet d'identifier au moins une solution réalisable à un problème d'optimisation, mais sans garantir que cette solution soit optimale.*

Le mot heuristiques vient du verbe grec *heuriskein*, qui signifie : trouver. L'inconvénient est de ne disposer en retour d'aucune information sur la qualité des solutions obtenues.

Feignebaum et Feldman [FF63] définissent une heuristique comme une règle d'estimation, une stratégie, une astuce, une simplification, ou tout autre sorte de système qui limite drastiquement la recherche des solutions dans l'espace des configurations possibles. Newell, Shaw et Simon [NSS58] précisent qu'un processus heuristique peut résoudre un problème donné, mais n'offre pas la garantie de le faire. Dans la pratique, certaines heuristiques sont connues et ciblées sur un problème particulier.

La métaheuristique, elle, se place à un niveau plus général encore, et intervient dans toutes les situations où l'ingénieur ne connait pas d'heuristique efficace pour résoudre un problème donné, ou lorsqu'il estime qu'il ne dispose pas du temps nécessaire pour en déterminer une.

En 1996, I.H. Osman et G. Laport [OL96] définissent la métaheuristique comme *un processus itératif qui subordonne et qui guide une heuristique, en combinant intelligemment plusieurs concepts pour explorer et exploiter tout l'espace de recherche.* Des stratégies d'apprentissage sont utilisées pour structurer l'information afin de trouver efficacement des solutions optimales, ou presque-optimales.

En 2006, le réseau Metaheuristics [Met12] définit les métaheuristiques comme suivant :

Définition 3.1.4 (Métaheuristique) *un ensemble de concepts utilisés pour définir des méthodes heuristiques, pouvant être appliqués à une grande variété de problèmes. On peut voir la métaheuristique comme une boîte à outils algorithmique, utilisable pour résoudre*

différents problèmes d'optimisation, et ne nécessitant que peu de modifications pour qu'elle puisse s'adapter à un problème particulier .

Comme l'heuristique, la métaheuristique n'offre généralement pas de garantie d'optimalité, bien qu'on ait pu démontrer la convergence de certaines d'entre elles.

3.1.3 Quelques méthodes d'optimisation

Cette section présente les principales méthodes d'optimisation (déterministes ou approchées).

3.1.3.1 Programmation mathématique

La programmation mathématique est la branche des mathématiques appliquées pour la modélisation et la résolution des problèmes d'optimisation que l'on peut modéliser sous forme de fonctions (purement) analytique : linéaires ou non-linéaires de variables entières ou continues voire mixtes.

De nombreux problèmes d'optimisation, où la fonction objectif et les contraintes peuvent être énoncées d'une façon analytique, peuvent être modélisés et résolus avec ce type de modèles.

Que se soit linéaire (en nombre entières ou non) ou non linéaire (convexe ou quadratique) ce type de problème peut bien être *NP-complet*, par conséquent dans la plus part des cas, sa résolution avec des méthodes exactes n'est pas toujours la façon la plus appréciée.

Le problème de voyageur de commerce qui peut être formulé à l'aide d'un modèle d'optimisation linéaire en nombre entier et résolu par une des méthodes de résolution tel que la *Simplexe* par exemple.

Par ailleurs, ce type de modèles est souvent le modèle -si ce n'est le principal- qu'on utilise pour résoudre des problèmes de planification, de décision et de stratégie dans plusieurs domaines (économie, industrie, etc.). Un exemple de domaine d'application de ce type de modèles est l'industrie pétrolière, où on y utilise pour déterminer la production optimale des puits et de raffineries de pétrole à moyen ou à long terme, en la présence d'un nombre de contraintes.

3.1.3.2 Optimisation combinatoire

En Informatique et en mathématiques appliquées, *l'optimisation combinatoire* également appelée **optimisation discrète** consiste à trouver la meilleure solution dans un ensemble discret dit ensemble des solutions réalisables. En général, cet ensemble est fini

mais compte un très grand nombre d'éléments, et il est décrit de manière implicite, c'est-à-dire par une liste, relativement courte et de contraintes que doivent satisfaire les solutions réalisables, dans ce cas le problème d'optimisation est dit problème d'optimisation sous contraintes.

Le problème d'optimisation combinatoire consiste donc, à trouver l'élément optimal dans cet ensemble discret et fini (maximum ou minimum) pour lequel la valeur de la fonction objectif soit optimale (maximale ou minimale).

En théorie, un problème d'optimisation combinatoire se définit par l'ensemble de ses instances, souvent infiniment nombreuses. Dans la pratique, le problème se ramène à résoudre numériquement l'une de ces instances, par un procédé algorithmique.

Formellement, une instance d'un problème d'optimisation combinatoire est définie par un couple $< S, f >$ tel que :

1. S est l'ensemble discret de configurations (espace de recherche)
2. $f : A \to \mathbb{R}$ est la fonction objectif (ou de coût)

La résolution consiste donc à déterminer s^*, pour un problème de maximisation :

$$s^* \in S,\ \forall s \in X \subseteq S, f(s^*) \geq f(s) \tag{3.3}$$

Et pour un problème de minimisation :

$$s^* \in S,\ \forall s \in X \subseteq S, f(s^*) \geq f(s) \tag{3.4}$$

X est l'ensemble des solutions réalisables (admissibles) et s^* est la solution optimale de $< S, f >$.

Bien que les problèmes d'optimisation combinatoire soient souvent faciles à définir, ils sont généralement difficiles à résoudre. En effet, la plupart de ces problèmes appartiennent à la classe des problèmes *NP-difficiles* et ne possèdent pas à ce jour de solution algorithmique efficace valable pour toutes les données.

Trouver une solution optimale dans un ensemble discret et fini est un problème facile en théorie : il suffit d'essayer toute les solutions réalisables, et de comparer les valeurs de la fonction objectif correspondantes pour désigner laquelle est la meilleure. Cependant, le parcours de toutes les solution peut prendre énormément de temps, et justement c'est à cause du temps de la recherche de la solution optimale que ce type de problème est réputé si difficile.

L'exemple le plus classique de ce type de modèle est le problème du plus court chemin entre deux sommets s et t d'un graphe. L'ensemble des solutions réalisables est l'ensemble

des chemins entre s et t tandis que la fonction objectif est la longueur du chemin. Ce modèle trouve son application dans plusieurs domaines, notamment dans le transport en commun dans la détermination du plus court chemin en présence de contraintes temporelles par exemple.

Quelques problèmes d'optimisation peuvent êtres résolus d'une façon exacte, par un algorithme glouton par exemple, ou par programmation linéaire si on prouve que le problème est modélisable de façon linéaire.

Dans la plus part des cas, et en la présence des contraintes, le problème est *NP-difficile*, et en pratique on se contente très souvent d'avoir une solution approchée obtenue à l'aide d'une heuristique ou une métaheuristique (voir section 3.1.3.3).

3.1.3.3 Méthodes heuristiques

Certains problèmes d'optimisation 3.1.2.2 demeurent hors de portée des méthodes exactes.

Les heuristiques s'appliquent dans des cas où le problème d'optimisation est n'est pas modélisable et donc impossible de le résoudre analytiquement, ou quand le temps nécessaire pour la résolution n'est pas raisonnable. Parfois on fait appel aux heuristiques tout simplement parce que l'exactitude de la solution n'est pas importante (par rapport au gain de temps) et qu'une solution approchée est acceptable.

Les métaheuristiques n'étant pas, a priori, spécifiques à la résolution de tel ou tel type de problème, leur classification reste assez arbitraire. On peut cependant distinguer :

1. Approches trajectoires : Le terme de recherche locale est souvent utilisé pour qualifier ces méthodes. Ces algorithmes partent d'une solution initiale (obtenue de façon exacte, ou par tirage aléatoire) et s'en éloignent progressivement, pour réaliser une trajectoire, un parcours progressif dans l'espace des solutions. Dans cette catégorie, se rangent :
   - La méthode de descente [Cea68]
   - Le recuit simulé [KGV83]
   - La méthode Tabou [CS00]
   - La recherche par voisinage variable [AL03]
2. Approches populations ou évolutionnaires : Elles consistent travailler avec un ensemble de solutions simultanément, que l'on fait évoluer graduellement. L'utilisation de plusieurs solutions simultanément permet naturellement d'améliorer l'exploration de l'espace des configurations. Dans cette seconde catégorie, on recense :
   - Les algorithmes génétiques [KCS06]
   - Les algorithmes par colonies de fourmi [DBS06]

- L'optimisation par essaim particulaire [PKB07]
- Les algorithmes à estimation de distribution [ZSTF04]
- Le *path relinking* (ou chemin de liaison) [DMG11]

Notons que ces métaheuristiques évolutionnaires seront probablement plus gourmandes en calculs, mais on peut supposer aussi qu'elles se prêteront bien à leur parallélisation.

Certains algorithmes peuvent se ranger dans les deux catégories à la fois, comme la méthode GRASP (Greedy randomized adaptive search procedure) [Hir06], qui construit un ensemble de solutions, qu'elle améliore ensuite avec une recherche locale.

Une autre manière, plus intuitive, de classifier les métaheuristiques consiste à séparer celles qui sont inspirées d'un phénomène naturel, de celles qui ne le sont pas. Les algorithmes génétiques ou les algorithmes par colonies de fourmi entrent clairement dans la première catégorie, tandis que la méthode de descente, ou la recherche Tabou, vont dans la seconde.

On peut également raisonner par rapport à l'usage que font les métaheuristiques de la fonction objectif. Certaines la laissent telle quelle d'un bout à l'autre du processus de calcul, tandis que d'autres la modifient en fonction des informations collectées au cours de l'exploration – l'idée étant toujours de s'échapper d'un minimum local, pour avoir davantage de chance de trouver l'optimal. La recherche locale guidée est un exemple de métaheuristique qui modifie la fonction objectif.

3.2 Optimisation des systèmes informatiques

3.2.1 Optimisation en informatique

En informatique, l'optimisation des performances désigne un processus dont l'objectif est de modifier un système existant afin de rendre l'une ou plusieurs de ses caractéristiques optimales. L'optimalité d'une caractéristique du système se mesure par un indice de performance à optimiser, c'est-à-dire à maximiser ou minimiser et cela en fonction de l'objectif de l'optimisation, on peut s'intéresser dans un système informatique (application web, base de données ou un serveur d'applications) à minimiser le temps de réponse, maximiser le nombre de clients ou encore le débit. Les indices de performances que l'on peut considérer sont nombreux et variés, mais aussi les niveaux d'optimisation. En effet les performances globales d'un système informatique quelconque dépend de différentes configurations matérielles et logiciels dans son entourage. Une application fut elle bien conçue et optimisée ne peut être complètement performante que si elle est déployée dans

un environnement bien configuré (matériel performant, système d'exploitation bien configuré, accès optimisé au ressources et au données, serveur d'applications bien paramétré, .etc).

3.2.2 Paramètres et niveaux d'optimisation

Dans une plateforme distribuée, les performances d'une application dépendent en partie du paramétrage des différents composants logiciels qui la constituent. Un réglage (*tuning*) s'avère nécessaire afin de tirer un meilleur parti de l'infrastructure matérielle et logicielle sur laquelle repose le système à tester. Il existe plusieurs niveaux de réglage pour optimiser un système tel qu'une application web dans un serveur d'applications. Pour gérer ces paramètres, on peut définir quatre classes de paramètres :

- Paramètres de niveaux matériel et système : à ce niveau d'optimisation on trouve les paramètres liés au choix du serveur (marque, nombre d'unités centrales (CPU), nombre de cœurs, vitesse de processeurs, mémoire vive (RAM), .etc.).
  Si le serveur est virtuel, en outre le nombre de CPU virtuels et la mémoire attribuée au serveur virtuel, il existe d'autres paramètres à régler dérivant de la technologie de virtualisation utilisée (pour VMWare on en trouve par exemple : CPU ShareS, Memory Reservation, Memory Shares). Quant au réglage du système, il y a peu de gain de performance à obtenir en réalisant un réglage du système d'exploitation (SE). Toutefois, le principal problème au niveau SE pouvant impacter les performances est la mauvaise configuration de l'interface réseau (par exemple : débit permis par le réseau, mode half-duplex/flux-duplex).
- Paramètres de la JVM : le choix de la JVM et son réglage est souvent lié à l'application à optimiser, et est préconisé par le constructeur (par exemple JRockit pour WebLogic et JVM de SUN pour JOnAS). On peut définir la taille min et max du tas Java et choisir l'algorithme du ramasse miettes (*garbage collector*). On peut aussi intervenir sur d'autres paramètres d'optimisation avancés de la JVM (compactage du heap, optimisation de l'allocation de la mémoire par les threads (TLA : Thread Local Area), optimisation du JIT (compilation Juste à temps), politique de gestion de verrous (locks), .etc.
- paramètres du serveur d'applications : premièrement, il faut s'assurer de l'optimisation des paramètres liés au serveur Apache mis en frontal du serveur d'applications. Dans la pratique on trouve souvent un serveur Apache en frontal, qui sert entre autres à servir le contenu statique (images, feuilles de style CSS, fichiers HTML, fichier JavaScript. . . .) sans solliciter le serveur d'applications, mais également à répartir la charge entre une instance Apache et plusieurs instance du serveur d'ap-

plications (notamment le réglage du module mod_jk pour JOnAS et mod_wl pour WebLogic). Il existe aussi d'autres paramètres d'Apache à régler (e.g. les workers et les pools de connexions). Quant au serveur d'application en lui-même, on trouve les points suivant :

1. Le connecteur web : c'est le composant du SA qui traite les requêtes des clients (AJP si Apache est délégué, sinon http ou https). Il faut paramétrer méticuleusement entre autres le pool de threads, le temps d'attente du connecteur pour recevoir une requête après avoir accepter une connexion, le nombre max de requêtes à mettre en attente quand aucun thread n'est disponible, nombre max de requêtes pouvant être traitées au sein de la même connexion TCP, .etc.

2. Le gestionnaire de sessions(Manager) : il gère les sessions http d'utilisateurs (informations stockées par l'application sur les objets HttpSession) ; on y trouve entre autres le nombre max des sessions simultanées autorisées,

3. La liste des différents services du serveur d'applications à activer(et à configurer, exemple : les workthreads du service EJB : le dimensionnement du pool des threads pour le traitement parallèle des messages, tous composant EJB de type MDB confondus). Le même principe est applicable pour le service des adaptateurs de ressources (e.g. JDBC RA) ou sinon pour une ressource de donnée (e.g. Datasource) : le paramétrage du pool de connexions.

Une autre classification des paramètres, d'ordre technique, peut être proposée. Elle se base sur la possibilité de modifier le paramètre :

- Paramètres modifiables avant le démarrage du serveur ;
- Paramètres modifiables avant le déploiement de l'application ;
- Paramètres modifiables pendant la marche de l'application.

Il faut noter que pour modifier la valeur des paramètres, plusieurs techniques sont applicables en fonction du type de paramètre (e.g. via des fichiers de configuration, via JMX par accès au MBean correspondant).

Une autre classification des paramètres est liée à la variation de la métrique de performance considérée m(exemple : nombre de clients simultanés) en fonction d'un paramètre p_i donné . Pour appliquer des algorithmes d'optimisation adéquats, il est nécessaire de voir la nature de variation de m en fonction de la variation d'un paramètre p_i pour des valeurs fixes pour le reste des paramètres.

3.3 Approches « métier » d'optimisation

En principe tout logiciel doit être délivré avec un guide d'utilisation. La tendance aujourd'hui est de livrer des guides d'optimisation avec toute nouvelle version du produit.

Cette section présente deux approches d'optimisation de deux produits informatiques connus sur le marché : la base de données Oracle et le serveur d'applications WebSphere d'IBM.

Une troisième approche générique [Hai06] pour l'optimisation des performances des serveurs d'applications Java EE a été étudiée et nous a aidée dans nos expérimentation 7 notamment dans le choix des paramètres à optimiser dans nos deux cas d'études.

3.3.1 Oracle et l'optimisation des Systèmes de gestion de bases de données

La méthode d'optimisation des performances d'Oracle repose sur l'identification et l'élimination des goulets d'étranglement dans la base de données, et l'optimisation des requêtes SQL. L'identification et la résolution d'un problème du premier goulet d'étranglement pourrait ne pas conduire à une amélioration des performances immédiatement, car un autre goulet d'étranglement peut se produire avec a un impact encore plus grand sur les performances. Pour cette raison, la méthode des performances d'Oracle est itérative. Le diagnostic précis du problème de performance est la première étape pour s'assurer que les modifications qu'on apporte au système se traduiront par des performances améliorées.

La pratique d'Oracle peut être appliqué jusqu'à ce que les objectifs de performances sont atteints ou jugés impraticable. L'Automatic Database Diagnostic Monitor (ADDM) implémente la méthode des performances d'Oracle décrite dans les points qui suivent, et des analyses statistiques afin de fournir un diagnostic automatique des problèmes de performances les plus importants.

Le réglage d'une BD Oracle est réalisé en deux phases : proactive (monitoring en temps réel, analyse des résultats, réponses aux alertes, etc.) et réactive(répondre aux problèmes signalés par les utilisateurs tels que la chute momentanée des performances). Plus un processus itératif pour s'assurer de l'efficacité des instructions SQL (surtout celles à forte charge).

La méthode de tuning d'oracle se base donc sur une succession de tâches sous forme d'une boucle de rétro-actions comportant ce que suit :

1. Préparation de la BD pour le réglage :
   - Déterminer la portée de l'optimisation et ses objectifs dans le futur. Ce processus est essentiel pour une future planification des capacités ;

- S'assurer de la validité et le bon fonctionnement de tout les éléments (logiciels ou matériels) pouvant êtres impliqués dans les performances (e.g. serveurs, OS, réseaux, SA, .etc.) ;
- Activer la collecte des statistiques de la BD, et les propriétés de réglage automatique de la BD, y compris AWR *(Automatic Workload Repository)* et ADDM *(Automatic Database Diagnostics Monitor).*

2. Réglage proactif :
   - ADDM détecte et publie automatiquement, les problèmes de perofrmance au sein de la BD (e.g. goulets d'étranglement à cause de CPU, problèmes de capacité d'E/S, problèmes de configuration de la BD ou encore de la dégradation des performance dans le temps). Grâce à ces résultats on peut identifier rapidement la source de tels problèmes ;
   - Appliquer les recommendations de ADDM, suivant le type du problème rencontré ;
   - Suivre l'évolution des performances et appliquer des réglages sur la BD en temps réel, grâce à Oracle Entreprise Manager. Cet outils permet aussi de répondre aux alertes de performances générées au fil du temps ;
   - Vérifier si les changements faites assurent le niveau d'amélioration de performnaces souhaités ;
   - Répéter ces étapes jusqu'à ce que les objectifs de performance soient atteints ou deviennent impossibles à réaliser en raison d'autres contraintes.
3. Réglage réactif : considéré comme réactif parce qu'il doit être effectué périodiquement lorsque des problèmes de performance sont rapportés par les utilisateurs.
   - ADMM doit être lancé manuellement ; et ce dans le but de ne pas attendre la prochaine analyse pour intervenir et effectuer le réglage nécessaire. Ceci est utile pour analyser les performances actuelles, ou pour analyser l'historique des performances à défaut d'une surveillance proactive ;
   - Les rapports du module ASH (Active Session History) permettent d'analyser les problèmes dites transitoires- de courtes durées et qui ne sont couverts par l'analyse de l'ADDM ;
   - L'AWR permet également de résoudre des problèmes de dégradation de performances, à l'aide d'une comparaison des performances entre deux périodes de temps différentes.
   - Vérifier si les changements faites assurent le niveau d'amélioration de performances souhaités ;
   - Répéter ces étapes jusqu'à ce que les objectifs de performance soient atteints ou deviennent impossibles à réaliser en raison d'autres contraintes.

4. Identification et optimisation des instructions à forte charge, sources possibles de conflits d'accès à la BD.

3.3.2 IBM et l'optimisation des serveurs d'application

Le serveur d'applications WebSphere d'IBM est un serveur d'entreprise robuste, il fournit un ensemble de composants, de ressources et de services que les développeurs peuvent s'en servir dans leurs applications. Chaque application possède ses propres exigences et utilise les ressources du SA souvent de façons différentes. Afin de fournir un degré élevé de flexibilité et de support pour ces applications, WebSphere offre une longue liste de paramètres de réglage *(knobs)* qui peuvent êtres utilisés afin d'améliorer les performances de l'application. Les valeurs par défaut des paramètres les plus couramment utilisés dan un SA, sont définie pour assurer une performance adéquate, sans modification, pour la plus large gamme d'applications. Cependant, et parce que chaque application est spécifique, il n'y a aucune garantie que cet ensemble de paramètres de réglage sera parfaitement adapté. Cette réalité met en évidence combien il est important d'effectuer des tests de performances et de réglage intensif pour l'application en question [Tea09].

La méthode d'optimisation d'IBM distingue entre trois niveau d'optimisation : réglage de base (points de réglage ayant un grand impact sur les performances) , réglage avancé (un moins d'impact sur les performances) et réglage de la messagerie asynchrone (optimisation fine).

1. Réglage de base :
   - Taille du tas *(JVM heap)* : élément important d'amélioration de performances. Par défaut, la valeur initiale est à 50MB, et la valeur maximale est à 256MB. En fonction de l'application en question, il est, en général, recommandé de régler les deux paramètres du tas avec la même valeur, pour des performances maximales. Un autre élément important dans le réglage de la JVM, c'est la politique du GC. Par défaut, WebSphere à partir de la version 8 utilise *gencon* comme politique de GC (WebSphere 7 utilisait optthruput par défaut). Si l'application n'utilise pas beaucoup d'objets de longue durée, généralement, elle fonctionnerait mieux avec la politique optthruput. Toutefois, comme chaque application est différente, un travail d'évaluation de chaque politique de GC doit être mené pour trouver le meilleur ajustement pour l'application concernée.
   - Taille du pool de threads : un pool de threads permet à des composants du serveur de réutiliser des threads afin de ne pas avoir à recréer d'autres threads lors de l'exécution. Les trois pools de threads les plus couramment utilisés (et optimisés) dans un serveur d'application sont les suivants :

  - Conteneur web : utilisé lorsque les demandes proviennent à travers de HTTP. Généralement, on a la plupart du trafic vient à travers le pool de threads du conteneur Web. Ce groupe a trois paramètres importants à optimiser : la taille min du pool, la taille max, et le délai d'inactivité du thread.
  - Default : utilisé si la requête concerne un MDB (Message Driven Bean) ou dans le cas où aucune chaine de transport particulière n'est indiquée pour un type précis de pool.
  - ORB *(Object Request Broker)* : utilisé lorsque des demandes proviennent à distance via RMI-IIOP pour un bean à partir d'un client EJB, une interface EJB distante, ou à travers d'un autre serveur d'applications.

  Le tableau indique les tailles par défaut de pool de threads et les valeurs de temps d'attente en inactivité pour les trois pools de threads les plus utilisés.
- Taille du pool de connexions : chaque fois qu'une application tente d'accéder à un dépôt (comme une base de données par exemple), il faut des ressources à créer, les maintenir et réaliser une connexion à ce dépôt de données. Par exmple, lors de l'accès à une base de données depuis une Servlet, l'établissement de la connexion peut prendre quelques secondes. Ce délai augmente le temps de réponse de la Servlet. Un SA permet via le mécanisme du pooling, de réduire ce temps de traitement. Dans WebSphere, les paramètres importants réglables pour un pool de connexions sont : la taille min et max du pool, et le délai d'inactivité d'un thread.
- taille du cache pour les objets des requêtes (Data source statement cache size) : pour les applications qui utilisent beaucoup de requêtes pré-compilées, le réglage de ce paramètre peut améliorer les performances de l'application.
- passage par référence d'ORB : la création d'une copie de chaque objet de paramètre et le faire passer (par valeur) à la méthode invoquée de l'EJB est plus coûteux que si l'on fait par une simple référence à cet objet.

2. Réglage avancé :
   - Cache des Servlets : activer et configurer le service du cache dynamique peut améliorer les performances de l'application, en mettant en cache la sortie des Servlets (mais aussi des services Web, du fragment de *Portlet* ou des commandes WebSphere).
   - Persistance des connexions HTTP : dans de nombreux cas, un gain de performance peut être obtenu en augmentant le nombre maximal de connexions autorisées pour une connexion HTTP.
   - Désactiver les services non utilisés

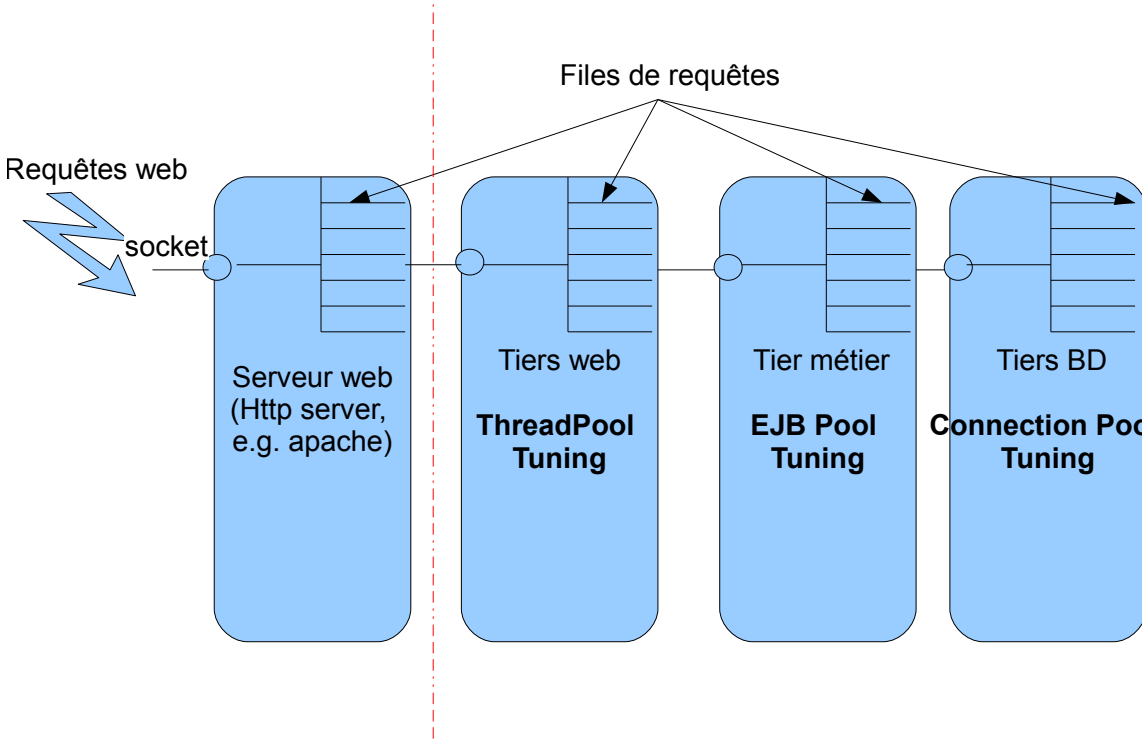

FIGURE 3.1: Chemin typique d'une requête le long d'une pile Java EE

– Localisation de Web server : pratiquement on se sert d'un serveur web dédié en frontal du SA pour s'occuper du traitement du contenu dynamique ou pour des raisons de gestion de charge. Mettre un IBM HTTP Server dans la même machine physique que WebSphere représente une consommation en CPU qui est équivalent à l'un des 4 cœurs du processeur (25%), par contre, et à partir de la version 5 de WebSphere, mettre le serveur web dans une machine dédiée distante peut économiser la consommation de ressources en CPU, mais également les performances de l'application (le temps de réponse par exemple), et pas le cas pour le débit par exemple.

3. Réglage de la messagerie asynchrone : Il y a deux options de réglage élémentaire associées à la configuration de la messagerie et qui a un impact significatif sur les performances : le type de stockage des messages, et le niveau de fiabilité des messages. Avec ces derniers, une technique mise au point plus avancée qui pourrait offrir des gains de performance supplémentaires, il s'agit de déplacer les logs de transactions et les stocker sur un disque rapide (quand si applicable).

3.3.3 Optimisation à base de points d'attente

Le concept d'optimisation à base de points d'attente dans un environnement Java Entreprise, est introduit par Steven Hains [Hai06]. Hains part du paradigme d'optimisation d'IBM de WebSphere et le concept de points d'attente d'Oracle. Après une étude-fouille dans la nature des requêtes des applications traversant les différents étages d'un SA, il s'est posé la question : Où est-ce que dans cette pile de technologie un blocage pourrait se produire ?

La figure 3.1 montre un exemple du chemin typique qu'une requête parcoure, le long d'une pile Java EE.

Concrètement, cette méthode se base sur un réglage des différents niveaux du serveur d'applications afin de s'assurer à la fois qu'il n'ait pas de points de blocage à un tiers donné et que ce réglage apporte un gain de performances considérable. Vraisemblablement, inspiré de la méthode d'IBM, Hains s'appuie sur une longue expérience dans le domaine des performances et d'optimisation dans le domaine du Java EE, il distingue deux catégories de points de réglage : les paramètres permettant un gain équivalent à 80% d'amélioration de performances, et des paramètres permettant le gain des 20% restants. Hains, affirme que l'ennemi de l'optimisation des serveurs d'applications Java EE est les éventuels goulets d'étranglement qui peuvent se produire à un étage quelconque du SA, de ce fait un réglage méticuleux des différents paramètres dans chaque étage pourrait éviter cet étranglement et apporter une souplesse et une fluidité dans le traitement des requêtes des clients à chaque étage et par conséquent, améliorer considérablement les performances de l'application en production. Hains, concentre précisément sur les pools de threads. En effet ce mécanisme du pooling peut bien ne pas être un élément de gain de performances s'il ne soit pas exploité bien comme il se doit en fonction de l'application déployée, notamment en matière de réglage des valeurs min et max des différents pool de chaque étage. Hains, considère l'acheminement de requêtes entre les différents tiers d'un SA comme un flux d'exécution au sein d'un réseau de files d'attente, et de ce fait chaque tiers est vu comme une file d'attente, et pour que le réseau soit stable il faut éviter le cas où l'une des files soit une source de blocage à un moment donné. Sans aller jusqu'à modéliser le SA sous forme d'un RFA quelconque, L'ensemble des points de réglage identifiés par Hains, pour l'optimisation d'un serveur d'applications Java EE, sont les suivants :

1. Principaux points d'optimisation : sont un ensemble de paramètres qui permettent d'améliorer les performances de 80% :
   - Optimisation du tas de la JVM
   - Optimisation de pools de threads
   - Optimisation de pools de connexions

   - Optimisation des caches
2. Points d'optimisation fine : sont des point qui ont moins d'impact sur les performances, et qui apportent un gain de 20% restants :
   - Optimisation des pools EJB
   - Optimisation des JSP pré-compilées
   - Optimisation de JMS
   - Optimisation des caches des requêtes pré-compilées
   - Optimisation avancée du JDBC

En résumé, La méthode d'optimisation à base de points d'attente, peut être résumée dans les points suivants :

1. Étudier minutieusement l'architecture de l'application et identifier les points où une requête pourrait potentiellement attendre ;
2. Ouvrir tout les points d'attente ;
3. Générer une charge équilibrée et représentative dans l'environnement ;
4. Identifier les points limites de saturation des points d'attente ;
5. Serrer tout les points d'attente afin de faciliter uniquement la charge maximale du point limite d'attente ;
6. Forcer tout les requêtes en attente d'attendre dans le serveur web ;
7. Si la charge est trop élevée, alors établir un point de coupure où on redirige les requêtes entrantes vers une page de type : réessayez votre demande ultérieurement. Sinon, ajouter plus de ressources.

3.4 Conclusion

Si le mot optimisation en informatique, désigne la procédure de recherche de la meilleure configuration pour laquelle un système informatique trouve ses performances optimales. L'auto-optimisation est donc, la capacité de rendre cette procédure de recherche de la meilleure configuration complètement automatique (i.e. sans intervention humaine).

Plusieurs travaux portent sur l'auto-optimisation dans des domaines variés en informatique ont été proposés. Par exemple, le domaine des réseaux de communication cellulaire était un domaine fertile pour l'auto-optimisation, par exemple les paramètres de configuration du réseau doivent être automatiquement et continuellement adaptés au profil du trafic dans le réseau et à l'environnement du réseau (topologie, propagation, interférences, .etc.). L'auto-optimisation de la consommation en énergie a toujours été un domaine d'actualité.

Dans les systèmes embarqués ou contraints tel que les téléphones ou ordinateurs portables, la consommation en mémoire et la consommation en énergie sont deux facteurs critiques. Des travaux d'auto-optimisation ont été menés afin de réduire l'empreinte énergétique et optimiser la consommation en mémoire dans de tels systèmes.

Plusieurs approches et algorithmes ont été étudiés afin de mettre en œuvre cette optimisation automatiques dans ces domaines. La grande partie des travaux s'inspire de la théorie de contrôle, du principe de la boucle de rétro-action, mais aussi de beaucoup d'autres branches de sciences (voir la section 4.1).

L'optimisation du fonctionnement d'applications informatiques est une tâche indispensable dans le travail d'administrateurs des différents couches applicatives d'un système informatique. Cette optimisation englobe différents aspects d'amélioration de fonctionnement de ces applications, partant de la garantie du service, jusqu'à l'amélioration des performances. Ce travail quotidien nécessite une connaissance approfondie de l'application et de son environnement, plus une attention et un suivi méticuleux par le biais d'outils de monitoring de diagnostique et d'analyse, qui aboutissent par une prise de décisions adéquates.

Une automatisation complète de ces tâches d'optimisation, s'avère hors de portée au jour d'aujourd'hui, pour plusieurs raisons d'ordre techniques liés à l'architecture des applications, le non-déterminisme du comportement de certaines applications et parfois pour des raisons sécuritaire (temps réel et applications critiques par exemple). Néanmoins, et malgré la non possibilité d'automatiser complètement une application du fait qu'on ne peut pas automatiser la pratique plusieurs outils d'automatisation de quelques tâches d'optimisation ont vu le jour grâce à des constructeurs de différents types de systèmes informatiques notamment IBM et Oracle.

Nous verrons dans le chapitre suivant, que l'apparition du concept de l' *autonomic computing* et de ses différentes applications, notamment celle réalisées à l'aide des modèles de composants a grandement ouvert la porte pour pouvoir espérer un jours concevoir des systèmes auto-optimisables.

Chapitre 4

L'auto-optimisation dans le contexte de l'autonomic computing

Nous rappelons dans ce chapitre de la notion du calcul autonome *(Autonomic Computing). Nous introduisons les approches autonomiques de l'auto-optimisation en justifiant les choix adoptés notamment concernant l'architecture. Nous consacrons une section au modèle architectural de composants que nous utilisons. Nous présentons enfin un exemple de système autonome -un système d'injection de charge auto-régulée- que nous étendrons dans le cadre de ce travail par l'ajout des aspects d'auto-optimisation.*

Sommaire

Les composants logiciels offrent des éléments structurants pour la conception et le développement de logiciels, afin de mieux faire face à la complexité croissante des systèmes et applications. L'autonomic computing vise à rendre les systèmes informatiques auto-administrables, c'est-à-dire capables de détecter d'éventuels changements ou problèmes et d'y réagir afin de préserver et adapter leur fonctionnement. L'amélioration des performances des systèmes fait partie des quatre grands thèmes mis en avant par IBM dans sa vision de l'autonomic computing, à travers la propriété d'auto-optimisation.

A l'instar des trois autres propriétés autonomiques identifiées par IBM, l'auto-optimisation repose sur des mécanismes de reconfiguration des éléments constituant le système à optimiser, avec le but particulier d'obtenir une configuration du système plus performante. Pour ce faire, les composants offrent un modèle et des outils intéressants, tels que la gestion du cycle de vie, des paramètres de configuration, des dépendances et de l'assemblage des composants, ou de l'état des composants via des mécanismes de réflexion (introspection, intercession).

4.1 Autonomic computing

Nous nous positionnons dans cette section par rapport à l'état de l'art concernant le domaine du *Calcul Autonomique* parfois appelé `Informatique Autonomique`. Nous présentons la vision d'IBM comme étant la fondatrice de ce concept. Nous présentons la boucle de contrôle autonome appelée MAPE-K expliquant la vision d'IBM pour la construction des systèmes autonomiques. Nous citons quelques travaux phares ayants repris cette approche pour le développement des systèmes autonomiques. Nous mettons l'accent sur les travaux qui ont abordé les aspects architecturaux (autonomicité globale), et donc nous nous contentons de juste citer quand'il faut les travaux portants sur d'autres éléments d'autonomicité (e.g. outils d'administration, gestion de ressources, support réseaux). Une autre section est consacrée à d'autres travaux initiés par des académiques et industriels pour construire des systèmes autonomiques ou même d'introduire des fonctionnalités d'administration autonome dans des systèmes patrimoniaux existants. Nous nous positionnons là aussi par rapport à ces travaux en mettant l'accent sur les travaux sur la catégorie de systèmes multi-tiers qui est dans l'axe de notre contribution, et ceux qui ont abordés l'aspect de coopération entre plusieurs boucles de contrôle autonome.

4.1.1 Bref historique

Les termes automatique, autonome et autonomique peuvent se référer à la même idée : restreindre au strict minimum l'intervention humaine. Mais il y en a quand même une différence importante entre ces concepts qui va être utile pour mieux comprendre pourquoi et comment une nouvelle discipline a vue le jour il y a maintenant dix ans à savoir l'informatique autonomique *(Autonomic Computing)*.

Une tâche ou un processus est dit automatique lorsqu'on remplace une routine manuelle par une routine codée. Par contre on dit qu'un processus est autonome s'il émule vraiment un *comportement* humain dans le sens où il réalise un ensemble de tâches automatiques pour atteindre le résultat final voulu sans intervention humaine entre ces tâches. Donc un système autonome peut être considéré comme une *imbrication* de plusieurs système automatiques *communicants* ensemble. Un système autonomique quant à lui peut être vu comme un système autonome où la responsabilité et le *prise de décision* relative à son fonctionnement sont automatisées. De cette façon on peut imaginer en fonction du degré de l'automatisation qu'un système autonome peut changer ses objectifs intermédiaires voire créer ses propres buts !

Le concept de l' *Autonomic Computing* a été abordé pour la première fois dans un manifesto paru en 2001 [IBM01]. Le papier part du constat que la complexité croissante des systèmes a rendu leur tâche d'administration de ces derniers de plus en plus complexe et risque d'atteindre les limites des capacités humaines. Le papier fait référence à plusieurs domaines scientifiques pouvant servir de sources d'inspiration pour la construction de systèmes auto-administrés sans pour autant donner une façon de faire pour y parvenir. Il cite par exemple : la théorie du contrôle, la cybernétique, et surtout le système nerveux humain, qui est à l'origine du terme *autonomic*.

Outre qu'il définisse 5 degrés d'autonomicité dans la construction des systèmes autonomes (le 5ème degré correspond à un système complètement autonomique où les systèmes et composants sont dynamiquement gérés par des politiques et règles métiers) IBM proposent une classification des fonctionnalités autonomes selon quatre catégories :

- *Autoconfiguration.* Le système s'adapte automatiquement et dynamiquement aux caractéristiques de son environnement d'exécution. Les ressources logicielles et matérielles doivent pouvoir s'insérer dynamiquement dans une infrastructure existante sans perturber les services en cours d'exploitation. L'autoconfiguration n'implique pas seulement la configuration individuelle des ressources mais aussi la configuration automatique de l'infrastructure qui les intègre.
- *Autoréparation.* Le système détecte, diagnostique et répare l'infrastructure face à une panne. Un système autoréparable isole la ressource défaillante et la répare, par

exemple, en re-déployant une ressource à l'identique ou en choisissant des ressources alternatives disponibles dans l'infrastructure. Par exemple, une infrastructure de commerce électronique peut être indisponible si les machines hébergeant les serveurs web sont en pannes. Une infrastructure autoréparable est capable de détecter la panne et de redéployer automatiquement les serveurs web sur de nouvelles machines, et de les intégrer dans le reste de l'infrastructure.

- *Auto-optimisation.* Le système autonome surveille ses propres performances et cherche à les optimiser. Il est capable de se dimensionner automatiquement et dynamiquement en fonction de la charge à un instant donné. Il est capable d'appliquer des politiques de gestion de performance de haut niveau définies par les administrateurs.
- *Autoprotection.* Le système anticipe, détecte et se protège des attaques internes ou externes. Ces systèmes ont la capacité de gérer les accès utilisateurs à l'ensemble des ressources de l'infrastructure. Le système peut, par exemple, restreindre temporairement les droits d'un utilisateur perturbant l'intégrité du système.

En l'an 2003 Kephart dans [OM03] considéré comme l'article fondateur du concept s'inspire du système nerveux humain qui permet par exemple à chacun de faire des efforts physiques sans avoir à se soucier du calcul du rythme cardiaque, de la pression sanguine ou de la quantité d'adrénaline, et de prendre des décisions nécessaire dans certain cas s'il le faut. Kephart affirme ainsi qu'un futur système autonome agirait de même, permettant de surpasser sa complexité et offrant aux administrateurs le temps pour se focaliser sur des problèmes de haut niveau.

Un système autonomique est donc un système avec des capacités autonomes (i.e. tâches automatisées pour atteindre un objectif précis) mais avec en plus la capacité de prendre des décisions. Reprenons l'exemple du système nerveux humain qui est naturellement autonomique parce-que à tout moment on peut prendre la décision d'arrêter l'effort physique si on sent qu'on est fatigué ou qu'il y a un problème de rythme cardiaque par exemple. En transposant cette vision vers le domaine de l'informatique autonomique, un système autonome doit donc pouvoir prendre de même des décisions quant à son fonctionnement global.

Bien qu'il décrive les considérations architecturales pour construire des systèmes autonomiques représentées par ce qu'on appelle la boucle MAPE-K et les notions de *Autonomic Manager* et *Managed Resource* et que l'on détaille dans la section suivante, ainsi que les chalenges possibles pour y parvenir, l'article ne se propose aucune approche technique pour le faire.

4.1.2 Travaux apparentés

4.1.2.1 Projets de recherche industriels

En 2002 IBM propose ABLE [BSP+02] un système multi-agents qui a servi à l'auto-paramétrage des serveurs web Apache. En 2003 IBM propose LEO [MLR03] un outils d'optimisation des requêtes pour DB2. Il s'agit d'un composant autonome basé sur l'observation des requêtes en exécution et un module d'apprentissage afin d'optimiser les requêtes et ainsi les performances. En 2004 IBM propose un outil basé sur le frame-work eclipse pour aider les développeurs à ajouter des éléments autonomes dans leurs applications.

En 2005 Sun propose un logiciel [N1S05]destiné principalement à la gestion autonome des ressources dans les fermes et clusters.

4.1.2.2 Projets de recherche universitaires

À l'initiative d'IBM une communauté scientifique s'est fédérée sur les recherches en système autonome. En 2004, la première conférence internationale sur les systèmes autonomes (ICAC) a été organisée [ica04]. Le programme de cette conférence montre que l'attention a principalement été portée sur l'optimisation et les performances, et en second plan sur la tolérance aux pannes, les aspects sécurité étant plutôt délaissés. La recherche en système autonome est désormais reconnue et est identifiée comme constituant un grand défi et un grand enjeu pour le futur. Parmi les travaux relevant de la construction de systèmes sous administration autonome on peut citer notamment :

Le projet *Recovery-Oriented Computing (ROC)* [Ber] et le projet le projet OceanStore [Uni] de l'université de *Berkeley of California* qui portent sur essentiellement sur la gestion autonomique et la haute disponibilité. Le projet Oceano [IBMb] d'IBM qui est centré sur l'auto-optimisation de la gestion d'allocation de ressources. le projet onCall [NCFC04] de l'université de Stanford qui propose un système permettant à des applications en grappes de répondre de manière autonome à des pics de charge. Le projet *Autonomizing Legacy Systems* de l'université de Columbia [Laba] propose d'adjoindre *après coup* des capacités autonomiques à des systèmes patrimoniaux, dont on ne dispose pas du code source. L'approche repose sur l'utilisation de *meta-architectures* qui abstraient la structure et le comportement d'un système cible. Des capteurs observent l'évolution de l'environnement du système cible, à travers les interfaces offertes par la meta-architecture. Une boucle de décision/commande analyse les observations remontées par les capteurs et les compare au comportement attendu du système. Enfin des effecteurs réalisent des actions correctives sur le système, également à travers les interfaces de la meta-architecture.

Le projet produit une implantation appelée Kinesthetics eXtreme (KX). Le projet *Darwin* [Col] de l' *Imperial College* de Londres a pour objectif le développement d'un langage de description d'architecture, d'une infrastructure et de langages de description de politiques d'administration. L'infrastructure d'administration proposée repose sur une représentation explicite de l'architecture logicielle du système, décrite dans le langage Darwin. Enfin, le projet *Smartfrog* [Labb] développe un langage et une infrastructure pour la gestion automatique de configurations logicielles réparties à grande échelle.

4.1.2.3 Échantillon de travaux dans le domaines des serveurs d'applications, systèmes distribués et intergiciels

Dans le domaine des systèmes multi-tiers et serveurs d'applications, des travaux sur le serveur web Apache ont été réalisés par [RBM05]. [RAC+02] ont travaillé sur l'auto-optimisation des serveurs d'applications en ajoutant des fonctionnalités de reconfiguration à chaud dans le composant des EJBs du serveur. Quant à [CKZ+03] ils ajoutent des fonctionnalités d'auto-récupération dans le serveur d'applications JBoss. [BPHT06, PBB+08] ont proposés `Jade` un intergiciel de gestion autonome d'applications en clusters. [BHS+08] propose `Tune`, un système de gestion autonome permettant d'envelopper les modules des systèmes patrimoniaux par des composants logiciels afin d'administrer l'architecture de ces systèmes sous forme d'architecture de composants. [HLM+09] propose `Entropy` un outil de gestion autonome des machines virtuelles dans les clusters. [GHBP09] propose une approche à base d'une architecture d'adaptation dynamique pour la gestion autonome de l'énergie dans les clusters multi-tiers.

4.1.3 Boucle de contrôle MAPE-K

C'est en 2003 et après avoir été inspiré par le système nerveux humain pour introduire le paradigme de l'informatique autonomique, IBM a suggéré un modèle de référence pour la boucle de contrôle autonomique en informatique(c'est le contrôle qui est autonome) **??** appelé : la boucle **MAPE-K** *(Monitor-Analyze-Plan-Execute-Knowledge)* en s'inspirant des boucles de contrôles déjà connues en théorie de contrôle dans l'automatique : où les paramètres des objets contrôlés sont ajustés (i.e. réglés) en continu en fonction des mesures de retour. Ce modèle est utilisé de plus en plus pour construire des architectures de systèmes autonomiques.

Comme nous pouvons le constater dans la figure 4.1 MAPE-K est constituée de cinq fonctionnalités :

– **Monitor** : responsable de l'observation de l'élément autonome (i.e. Autonomic Ma-

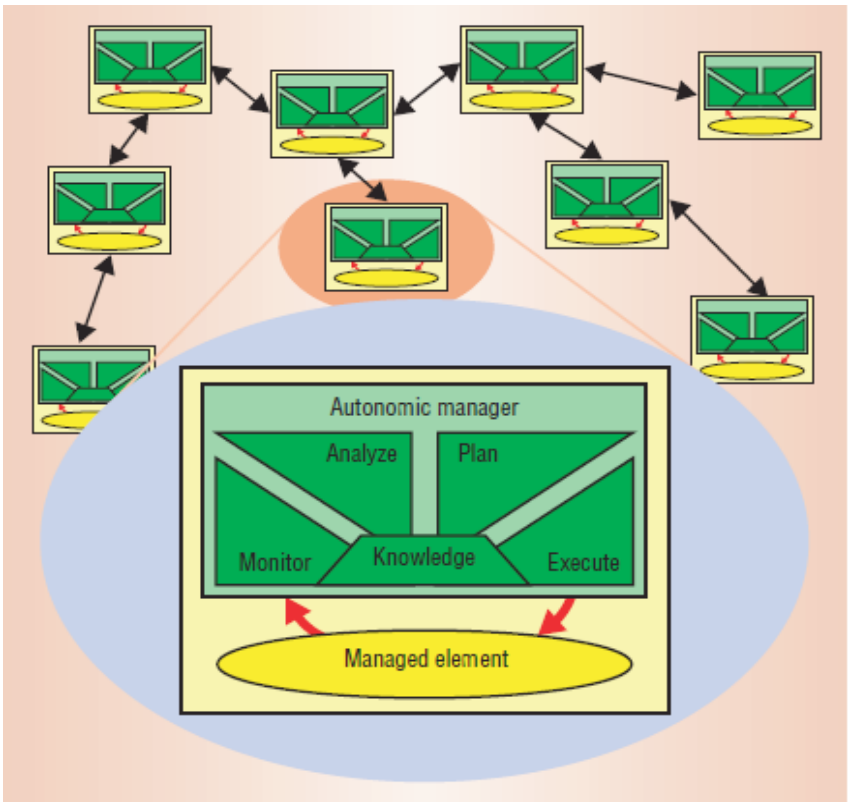

FIGURE 4.1: Un système autonome est un ensemble d'éléments autonome qui coopèrent pour atteindre un objectif commun, Boucle MAPE-K, Source : J.O. Kephart, D. Chess, "The Vision of Autonomic Computing."

nager + Managed resource(s)),

- **Analyze** : responsable de l'analyse d'observations,
- **Plan** : responsable de la planification d'une éventuelle réaction appropriée,
- **Execute** : responsable de l'exécution du plan de réaction,
- **Knowledge** : chargé d'alimenter les autres fonctionnalités en connaissance sur l'élément autonome.

La flèche allant du ME vers le *Monitor* de l'AM est une sonde (ou jauge) permettant de récolter de l'information sur le ME. La inverse est un actionneur qui permet d'appliquer le plan d'exécution sur le ME.

Un système autonomique est donc un ensemble d'éléments autonomiques (voir figure 4.1). Un élément autonomique est constitué de un ou plusieurs éléments administrés *Managed Elements, MA* couplés et contrôlés par un *Autonomic Manager, AM* qui les représente. C'est l'Autonomic Manager qui doit implémenter -en principe- les cinq fonctionnalités de la boucle MAPE-K. Ces éléments ont pour rôle de délivrer des services ou contenir des ressources pour être exploiter par soit le client final du système soit par un

autre élément autonomique.

Un MA peut être un service ou une application comme il peut bien être une ressource physique. Il peut être un système ordinaire (non autonomique) mais qui doit être adapté pour permettre au AM de lui surveiller et contrôler. Il peut donc être un serveur web, une base de donnée, un composant logiciel dans une application (e.g. l'optimiseur de requêtes dans une BD ou le pool de threads d'un connecteur http), un système d'exploitation, un cluster de machines dans un environnement en grappes, un réseau sans fils, etc.

L'AM est un composant logiciel qui idéalement peut être configuré par les administrateurs humains en utilisant des objectifs de haut niveau. L'AM exploite les données recueillies par les sondes et capteurs sur le(s) ME et la connaissance interne du système afin de planifier et d'exécuter, en se basant sur les objectifs de haut niveau, les actions de bas niveau nécessaires pour atteindre ces objectifs. La connaissance interne est souvent un modèle architectural du ME qui n'a pas comme rôle la description d'une configuration précise des composants et connecteurs dont l'élément administré doit respecter. Le modèle architectural fixe un ensemble de contraintes et propriétés pour les composants et connecteurs à partir desquelles le système peut déterminer quand l'élément administré viole le modèle et il est besoin d'être adapté (voir la section **??**. Les objectifs sont souvent exprimés en utilisant des politiques de type :

- **évènement-condition-action** : e.g. lorsque 95% du temps de réponse du serveur web dépasse 2secondes, et il y a assez de ressources, alors augmenter le nombre de serveurs web actifs.
- **objectif** : e.g. on voudrait que le temps de réponse du serveur web soit inférieur de 2secondes lorsque celui du serveur d'application est inférieur à 1seconde. Le AM décide, basé sur la connaissance interne, d'ajouter ou de supprimer de ressources pour atteindre la situation désirée.
- **utilité** : e.g. pour l'exemple précédent, la fonction d'utilité prendra en entrée le temps de réponse du serveur web et celui du serveur d'applications et en sortie le niveau désiré correspond à ces entrées. Ainsi la fonction d'utilité donne le niveau désiré pour chaque combinaison.

4.1.4 Coopération entre éléments autonomes

Comme nous l'avons cité auparavant un système autonomique se construit à l'aide de plusieurs éléments autonomes. Ces éléments autonomes doivent communiquer entre eux afin d'atteindre un objectif commun. Prenons l'exemple d'un cluster de serveurs pour l'optimisation de l'allocation de ressources aux applications afin de minimiser le temps de réponse global ou l temps d'exécution des applications. Ainsi, les AMs doivent veiller non

seulement par rapport au conditions de leurs propores MEs mais aussi doivent avoir une connaissance (perception) de leur environnement et particulièrement d'autres MEs dans leur réseau. Cette notion de coopération entre éléments individuel pour atteindre un objectif commun est un aspect fondamental dans le domaine de la recherche dans les systèmes multi-agents. Un nombre considérable de travaux de recherche étudie l'application des systèmes multi-agents pour mettre en oeuvre la coopération des éléments autonomiques, en étudiant particulièrement les difficultés rencontrées dans les systèmes multi-agents afin de garantir que le comportement des objectifs individuels de chaque agent va vraiment aboutir à l'objectif commun [Jen00, LPGG00, KC00]

Une alternative de la coopération à base de multi-agents est la structuration hiérarchique des éléments autonomes [WCL+00]. Nous allons voir dans la partie contribution dans le chapitre 6 la solution que nous proposons pour implémenter la coopération entre nos boucles autonomes.

4.2 Approches autonomiques de l'auto-optimisation

D'un point de vue général, l'optimisation désigne l'acte, la procédure ou encore la méthode la plus efficace pour faire quelque chose [Mer]. Les intergiciels complexes, tels que les serveurs d'application Java EE (e.g. WebSphere, Jboss, JOnAS) ou les systèmes de gestion de bases de données (e.g. Oracle, DB2), ont des centaines de paramètres de réglage, dont certains ont une influence plus ou moins marquée sur les performances, et ce, en fonction de l'application. L'optimisation du fonctionnement de tels systèmes consiste à régler leurs paramètres dans un objectif d'amélioration de la performance globale, dans un contexte applicatif déterminé.

Dans les environnements complexes et répartis, où toute reconfiguration de tout élément est susceptible d'influer sur les performances du système global, il est très difficile de trouver le réglage exact du système permettant d'obtenir les meilleures performances. Cette tâche d'optimisation est typiquement empirique, basée sur un processus itératif d'expérimentations.

L'auto-optimisation est l'aptitude du système à chercher continuellement à identifier et saisir les occasions et les moyens de se rendre plus efficace en matière de performances. Un système autonome - tout comme un cerveau humain, qui modifie ses circuits lors de l'apprentissage - va surveiller, expérimenter et régler ses propres paramètres, et apprendre à faire les choix appropriés.

Dans nos travaux, nous reprenons le principe de boucle de contrôle pour construire des systèmes autonomes, en s'inspirant de la structuration MAPE-K. Cette approche a

été reprise dans plusieurs travaux de recherche académiques et industriels dans différents domaines, notamment dans les systèmes répartis et les modèles à composants. Dans le domaine de l'auto-optimisation, on peut citer à titre d'exemple [SBP08, Sha10, TBPH08, Kan11].

Nous abordons en détail dans la section 4.3.3 un exemple d'adaptation et de mise en œuvre de la boucle MAPE-K au sein d'une plate-forme d'exécution répartie offrant des mécanismes d'administration et de déploiement autonomes d'applications et de services. Nous décrivons plus particulièrement l'architecture de la plate-forme et les fonctionnalités qui relèvent de l'auto-optimisation.

4.3 Approche architecturale à composants

4.3.1 Les modèles à composants

La notion de composant logiciel est apparue pour la première fois dans [McI68] où l'auteur décrit sa vision d'un *marché* de composants logiciels réutilisables et facilement combinables afin de construire rapidement des systèmes complexes. Cependant, il a fallu attendre les années 90 pour que ce terme se popularise, notamment grâce à des modèles industriels tels que COM *Component Object Model* de Microsoft [Mic], CCM *CORBA Component Model* [GGM03], et les EJB *Enterprise Java Beans* de Sun [Bri].

Alors que les modèles à objets, déjà largement répandus, abordent les problèmes de la réutilisation et de l'adaptation du logiciel au niveau du développeur, les modèles à composants permettent de prendre en compte d'autres rôles confrontés à ces problèmes, tels que l'administrateur, l'architecte ou l'intégrateur, en leur proposant des outils adaptés. Les principes fondateurs du génie logiciel basé sur les composants sont détaillés dans [Szy02]. De nombreuses définitions de la notion de composant logiciel ont étés proposées. Nous citons celle retenue lors de l'édition 1996 de la conférence européenne sur les programmes orientés objets (ECOOP) [Coi96] :

Définition 4.3.1 *Un composant logiciel est une unité de composition dotée d'interfaces spécifiées. Un composant logiciel peut être déployé indépendamment et sujet à une composition par une tierce entité [BSW97].*

Nous citons également cette autre définition de [Szy97], très couramment acceptée :

Définition 4.3.2 *Un composant est une unité de composition dont les interfaces et les dépendances contextuelles sont spécifiées sous forme de contrats explicites.*

L'article de synthèse sur les langages de description d'architecture de Medvidovic et Taylor [MT00] donne un ensemble très détaillé de concepts pour caractériser les architectures à composants. Pour notre part, nous présentons les principaux concepts à travers un modèle à composants particulier - le modèle Fractal - que nous utilisons pour construire notre plate-forme de Self-benchmarking. Ce choix du modèle Fractal est justifié par des critères objectifs d'adéquation à la problématique de l'architecture des systèmes d'autonomic computing, ainsi que par la poursuite de l'approche proposée par le projet Selfware 4.3.3.

4.3.2 Le modèle Fractal

Issu de travaux conjoints entre France Télécom R&D et l'INRIA, Fractal est un projet du consortium de logiciel libre OW2 (anciennement ObjectWeb). Le projet spécifie un modèle à composants indépendamment de tout langage d'implantation et de toute cible technique [BCS04]. Il existe ainsi des implantations en différents langages (Java [BCL+06], C, .NET, Smalltalk, Python), utilisées pour la construction de noyaux de systèmes d'exploitation embarqués (THINK [FSLM02]) ou d'applications réparties telles que CLIF [Dil09]. Cette grande polyvalence découle :

- des concepts et fonctionnalités avancés qui sont proposés (e.g. hiérarchie et partage de composant, introspection et intercession),
- mais qui ne sont pas obligatoires.

De manière classique, un composant Fractal est une unité de traitement avec un état, fortement encapsulée. Comme le montre la figure 4.2, il interagit avec l'extérieur via des *interfaces client* (services requis) et des *interfaces serveur* (services offerts). Une interface ne désigne pas une définition abstraite de type comme dans les langages à objets, mais un point d'accès (typé et désigné par un nom) sur un composant. Ce qui est moins classique, ce sont les *interfaces internes* qui permettent de relier une interface externe à une interface d'un sous-composant. Ceci correspond à la possibilité de définir des composants dits *composites*, qui ne sont pas des implantations (appelées composants *primitifs*) mais des assemblages de sous-composants. Fractal offre donc un modèle de composant récursif, qui ne se limite pas à une simple hiérarchie, car un sous-composant peut appartenir à plusieurs composants parents. Ce *partage* de composants a été tout particulièrement conçu pour représenter le partage de ressource, par exemple dans les systèmes d'exploitation.

Le contenu du composant est isolé par une *membrane*, dont la fonction est de contrôler les interactions au niveau des interfaces externes (interception) et de fournir des capacités d'observation et de reconfiguration du composant (introspection et intercession). Cette membrane est contrôlée par des *interfaces de contrôle*. La spécification du modèle Fractal mentionne quelques interfaces de contrôle, ni obligatoires, ni exhaustives : nommage du

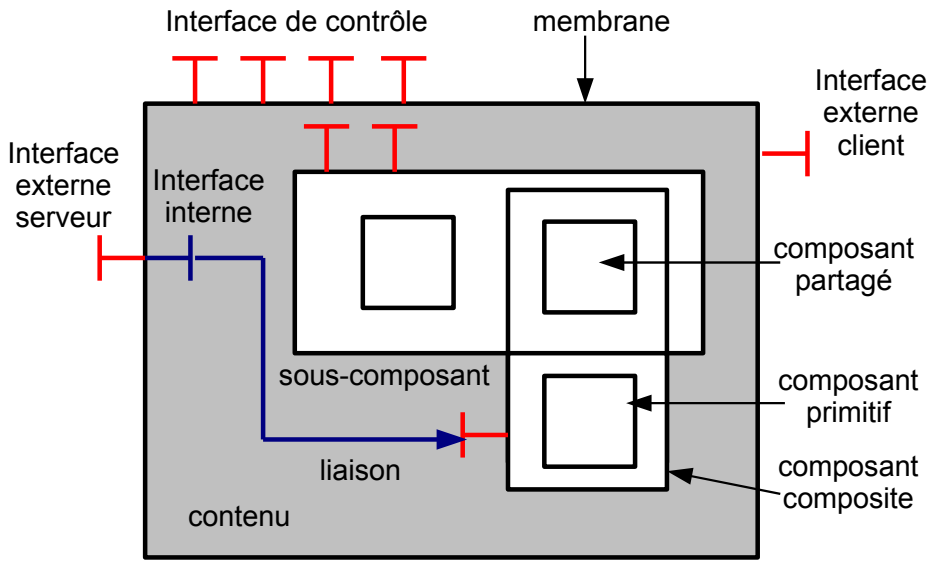

FIGURE 4.2: Éléments de terminologie du modèle Fractal

composant, établissement ou suppression d'une *liaison* d'une interface client du composant avec une interface serveur (injection de dépendances), accès au contenu d'un composite, gestion du cycle de vie (démarrage, arrêt), et accès à l'état du composant (attributs).

Ainsi, le modèle Fractal a clairement été spécifié en vue de permettre l'observation et la reconfiguration à chaud d'un système, incluant l'ajout, le retrait ou le remplacement d'un composant. D'ailleurs, un composant Fractal n'est pas un simple élément de conception, mais un élément existant à l'exécution. Ces caractéristiques justifient l'utilisation du modèle Fractal dans le projet Selfware, dont l'objectif était d'offrir un cadre architectural pour la construction de systèmes relevant de l'autonomic computing.

4.3.3 Selfware : une architecture à composants pour l'Autonomic Computing

Selfware est un projet RNTL pour le déploiement, la configuration et l'administration autonome de systèmes répartis. L'objectif principal du projet Selfware est le développement d'une plate-forme logicielle pour la construction de systèmes informatiques répartis sous administration autonome, et son application à deux domaines particuliers : d'une part, l'administration de serveurs d'applications J2EE de grande taille (serveurs en grappes et fermes de serveurs), et, d'autre part, l'administration d'un bus d'information d'entreprise [Pro07]. La majorité de ces logiciels sont disponibles sous licence libre dans

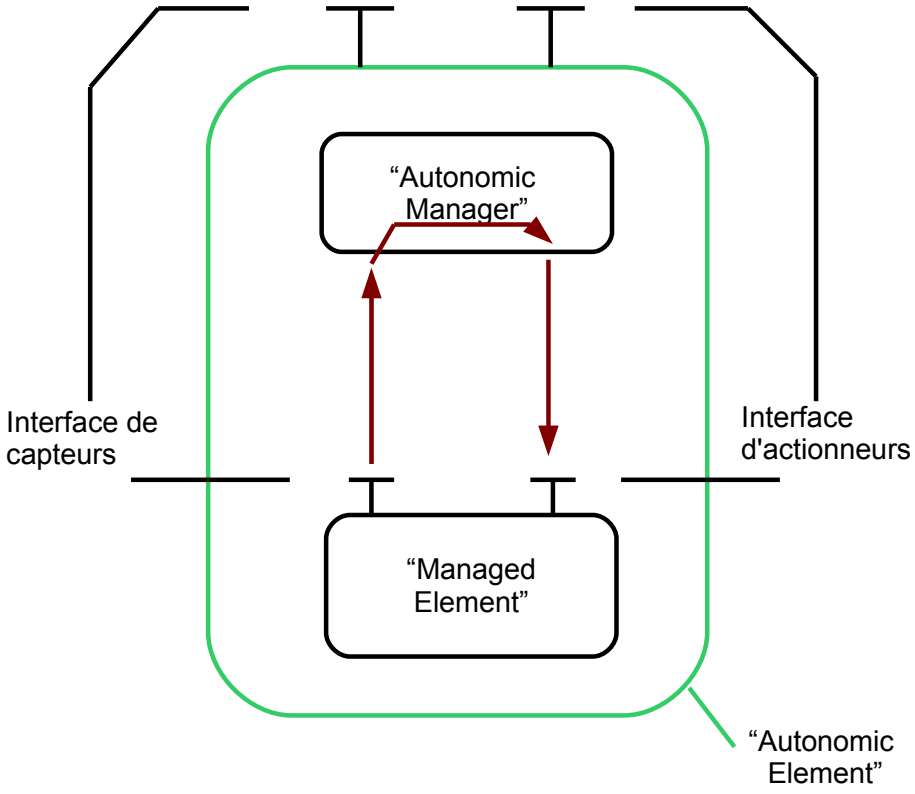

FIGURE 4.3: L'élément autonomique est la combinaison d'un Managed Element et d'un Autonomic Manager

le cadre du consortium OW2, en particulier au sein du projet JASMINe [OW2].

Conformément à l'approche de la boucle MAPE-K proposée par IBM, Selfware adopte le principe de la boucle de contrôle observation-analyse-décision-action. Afin de pouvoir ajouter des propriétés autonomiques à un système patrimonial quelconque, Selfware propose de donner accès aux fonctionnalités d'observation et de contrôle (reconfiguration, redémarrage) des éléments du système au travers d'une couche de composants Fractal d'encapsulation (*wrappers*). Ces composants sont du type ME (*Managed Element*), sur lesquels des composants du type AM (*Autonomic Manager*) viennent réaliser des comportements autonomiques (voir 4.3 tels que l'auto-réparation ou l'auto-dimensionnement d'architectures trois-tiers. Les AM pouvant également être des ME, les fonctionnalités autonomiques peuvent s'appliquer aux composants qui les implantent. Ainsi, la boucle autonomique d'autoréparation est capable de s'autoréparer.

Cette approche architecturale vise à se prémunir d'une explosion de la complexité que pourrait engendrer l'introduction ad hoc de boucles de contrôle. En effet, si le système à rendre autonomique est déjà complexe au départ, que dire du niveau de complexité du

système résultant de l'ajout de fonctionnalités autonomiques sans une vision uniforme et cohérente du système et des boucles de contrôle ?

Quant au modèle à composants Fractal, il a montré qu'il offrait tous les mécanismes nécessaires à la construction de tels systèmes, pour en gérer la complexité à travers une vue uniforme réflexive et récursive, permettant l'observation et la reconfiguration.

4.4 Injection de Charge Auto-Régulée (ICAR)

4.4.1 Introduction

Une motivation classique des tests en charge est le besoin de connaître la charge maximale admissible par un système avant qu'il ne sature. Par saturation, nous entendons l'atteinte de certains critères pouvant relever de la qualité de service (e.g. temps de réponse trop long, rejet d'appel) ou d'un taux d'utilisation critique des ressources du système. Pour connaître le nombre maximal d'utilisateurs virtuels ou de requêtes par unité de temps que le système est capable de servir dans les conditions nominales définies, l'équipe de test doit procéder de manière itérative et empirique, pour trouver le niveau de charge critique.

Le principe de l'injection de charge autorégulée est illustré par la figure 4.4. Il consiste à introduire une boucle de contrôle capable d'ajuster le niveau de charge produit par un injecteur de requêtes, en fonction de l'observation des métriques associées aux critères de saturation. Un tel système est décrit dans [HSDV10]. Le système est basé sur le canevas logiciel d'injection de charge CLIF, auquel une boucle de contrôle d'injection de charge est ajoutée par le biais d'un composant "contrôleur" tout à fait similaire à un AM Selfware. Le différence ici est que CLIF est conçu selon le modèle Fractal et qu'il est inutile de l'encapsuler. Ce système ICAR étant à la base de nos travaux, nous présentons le canevas CLIF et l'architecture d'ICAR.

4.4.2 CLIF, un canevas logiciel de test de performance

CLIF est un logiciel à composants Fractal en Java pour l'injection de charge et la mesure de performance. De par son approche canevas logiciel, il est notamment prévu pour s'adapter à toute sorte de système sous test, en termes de protocoles d'invocation et de ressources à observer. Il offre aussi différentes interfaces utilisateurs : interface graphique indépendante, plug-ins Eclipse, ligne de commande, serveur d'intégration continue Jenkins...

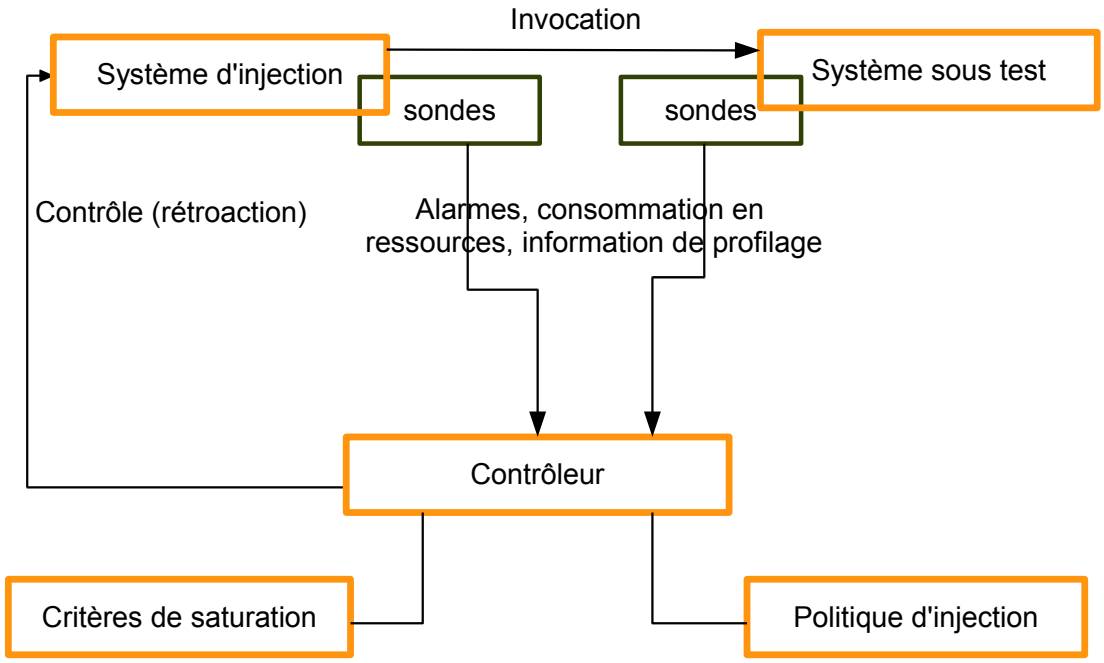

FIGURE 4.4: Schéma de principe de test de charge avec contrôle automatique d'injection

4.4.2.1 Principes de fonctionnement

La figure 4.5 montre les principes d'exécution d'un test CLIF. CLIF est une application Fractal répartie qui permet de déployer et superviser des composants d'injection de charge (*injecteurs*) et de mesure de consommation de ressource (*sondes*) à travers le réseau. Les *serveurs CLIF* sont des composants de type usine à injecteur ou sonde qui permettent ces instanciations de composants à distance.

Lorsque le test est démarré, les injecteurs envoient des requêtes au système sous test et produisent des rapports de requêtes indiquant le temps de réponse, le type de requête et des éléments de la réponse du serveur. En parallèle, les sondes mesurent la consommation de certaines ressources (processeur, mémoire, réseau. . .) et produisent également des rapports de mesure. Tous ces rapports représentent les mesures complètes, qui seront stockées dans un composant de *stockage*, pour mise à disposition à un composant d'*analyse* qui permettra de produire des rapport de test.

Toutefois, ces mesures complètes ne seront disponibles qu'à la fin de l'exécution du test, après une opération de *collecte*. En effet, afin de ne pas perturber le fonctionnement du réseau au cours du test, les mesures sont mises en tampon localement sur chaque injecteur et sur chaque sonde. En outre, CLIF étant conçu pour admettre de très hauts trafics (millions d'utilisateurs virtuels, millier d'injecteurs et sondes), le composant de stockage centralisé pourrait se trouver engorgé s'il devait stocker toutes les mesures en

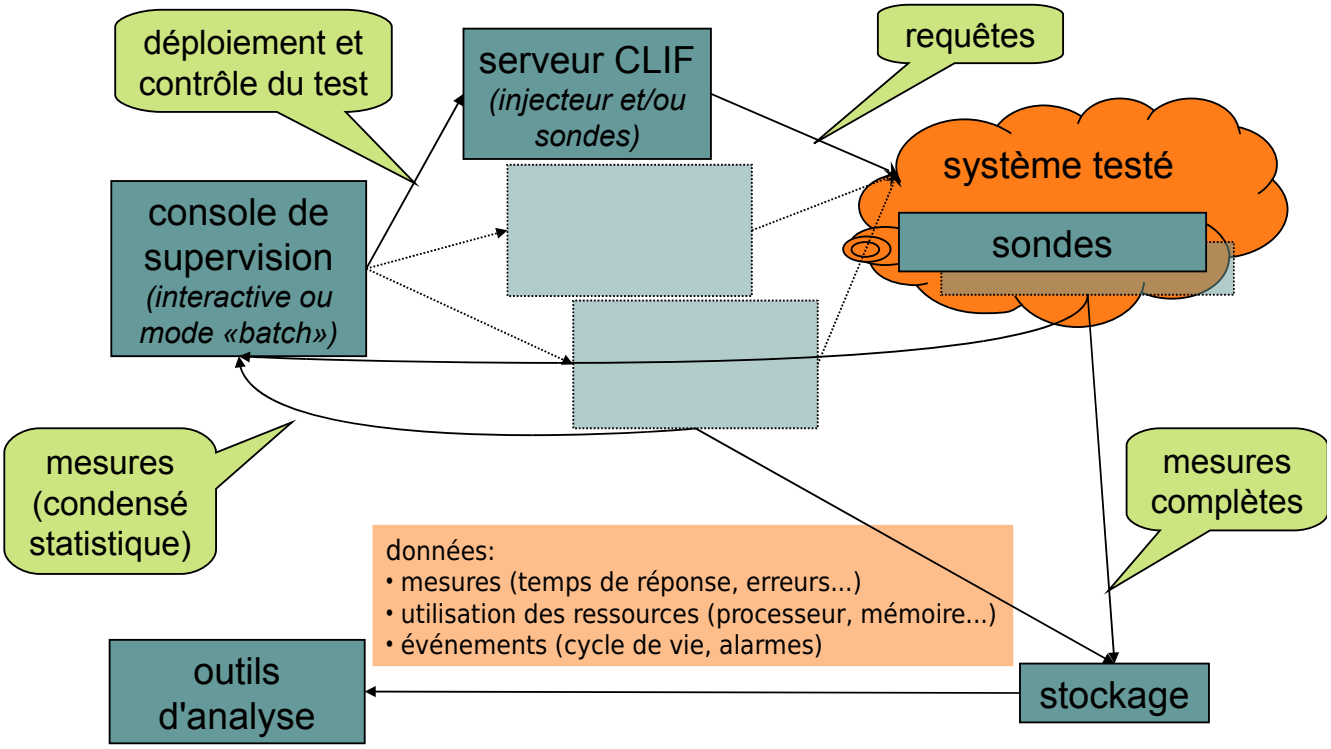

FIGURE 4.5: Principes fonctionnels de CLIF

temps réel.

Pourtant, les besoins de supervision des tests de performance en cours d'exécution (e.g. évolution des temps de réponse et des débits, occurrences d'erreur, évolution du taux d'utilisation du processeur) sont incontournables. Pour cette raison, les injecteurs et les sondes CLIF offrent des statistiques mobiles sur l'ensemble des métriques qu'elles mesurent (e.g. temps de réponse et débit moyens, utilisation moyenne du processeur ou de la mémoire). Cette fonctionnalité permet de suivre en temps réel l'évolution des différentes métriques, et de les afficher, par exemple, dans des graphes de supervision.

4.4.2.2 Architecture à composants

La figure 4.6 décrit l'architecture à composants de CLIF. On y retrouve les composants de stockage (*storage*) et d'analyse (*analyzer*) mentionnés, ainsi que les injecteurs et les sondes représentés par le type lame (*blades*). Les composants sondes, injecteurs et stockage sont tous liés à un unique composant de supervision (*supervisor*). Ce composant permet de contrôler et superviser chaque injecteur et chaque sonde de manière centralisée. Il est typiquement embarqué dans l'interface utilisateur.

Nous ne décrivons pas davantage cette architecture et le rôle de toutes les interfaces

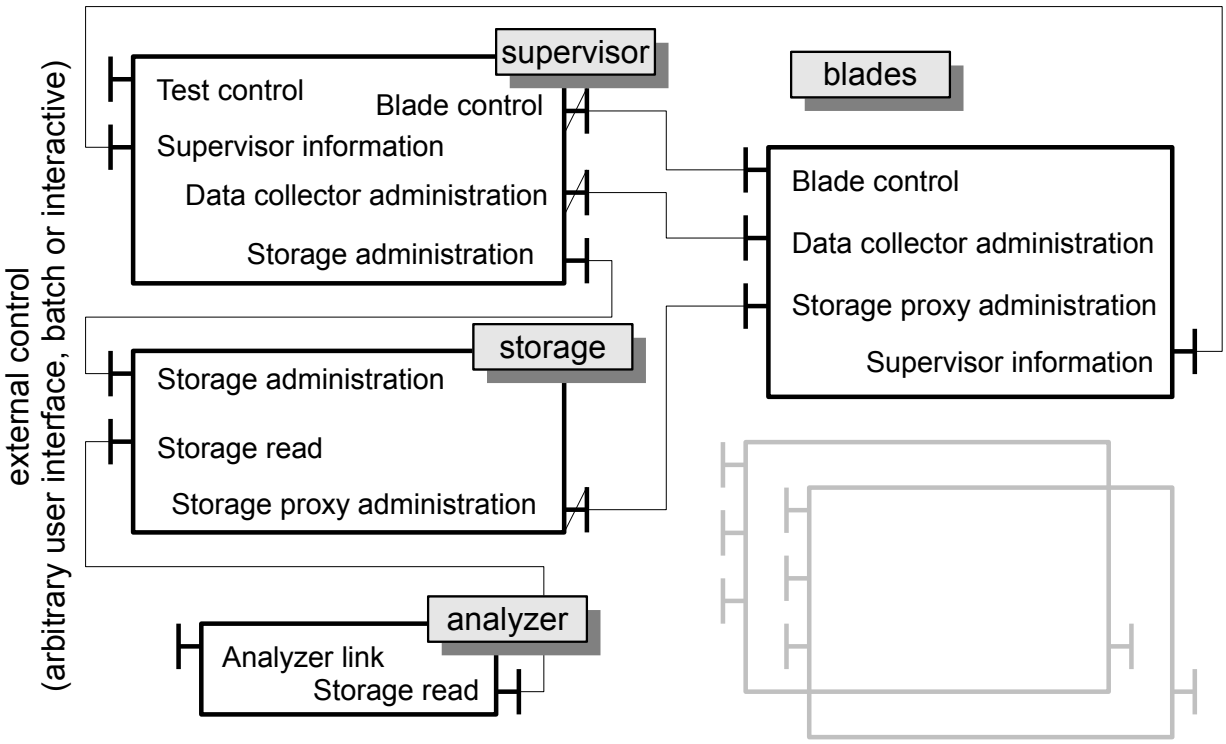

FIGURE 4.6: Architecture d'un plan de test CLIF déployé

mentionnées par la figure. On se référera à l'article [Dil09] pour obtenir toute explication supplémentaire.

4.4.3 Régulation automatique de la charge

CLIF permet de définir un scénario de test combinant un ou plusieurs comportements types d'utilisateur virtuel. Pour chaque comportement, un *profil de charge* spécifie l'évolution du nombre d'utilisateurs virtuels actifs en fonction du temps d'exécution. Les profile de charge sont dont définis a priori, avant le déploiement et l'exécution du test. On peut y définir autant de paliers et de rampes de montée en charge que l'on souhaite, mais il demeure statique.

Toutefois, CLIF permet de modifier le nombre d'utilisateurs virtuels à la volée, à travers l'interface `Test control` du composant de supervision, ou de l'interface `Blade control` du composant injecteur. Cette interface offre donc le point de rétroaction pour ajuster automatiquement la charge. Par ailleurs, nous avons indiqué que les sondes et les injecteurs offrent des statistiques mobiles sur leurs mesures, ce qui donne le point d'observation pour notre boucle de contrôle d'injection.

Les travaux décrits dans [HSDV10] donne l'architecture complète du système d'injec-

tion de charge auto-régulé et de l'algorithmique de contrôle de la montée en charge, basée sur la théorie des files d'attente. En résumé, le système est composé d'un composant de contrôle de la saturation et d'un composant de contrôle de l'injection. Le premier possède un pouvoir de contrôle sur le second, car il stoppe la montée en charge lorsqu'un critère de saturation est atteint. On peut parler de coordination hiérarchique entre les deux boucles de contrôle. Comme nous le verrons dans la partie contribution de cette thèse, chapitre 6, nous adopterons un autre principe de coordination permettant de construire des boucles de contrôle de façon totalement indépendante.

4.5 Conclusion

Dans ce chapitre, nous avons placé la problématique de l'auto-optimisation dans le contexte de l'autonomic computing. Au-delà des aspects algorithmiques présentés dans le chapitre précédent, nous avons voulu ainsi nous intéresser à la question de la construction de systèmes auto-administrés. Conformément à l'approche de la boucle de contrôle MAPE-K popularisée par IBM, nous envisageons la construction de notre plate-forme d'auto-optimisation par l'utilisation d'un système de test de performance offrant sondes et injecteurs de charge, en l'occurrence le canevas logiciel CLIF, et l'ajout de boucles de contrôle.

Plus spécifiquement, notre démarche hérite d'une approche architecturale à base de composants, à travers le projet Selfware et l'utilisation du modèle Fractal, pour la construction des boucles de contrôle. Enfin, nous avons vu qu'un système d'injection de charge auto-régulée avait été construit selon cette approche à l'aide de CLIF. Ce système peut nous permettre d'obtenir une mesure des capacités maximales d'un système sous test en terme de nombre maximal d'utilisateurs virtuels qu'il peut servir en parallèle.

Dans la partie contribution qui suit, nous verrons l'algorithme d'optimisation que nous proposons, dans le chapitre 5, puis l'architecture logicielle associée dans le chapitre 6.

Deuxième partie

Contribution

Chapitre 5

Politique d'optimisation

Nous présentons dans ce chapitre le cœur algorithmique de notre travail concernant l'auto-optimisation qui se décompose en deux parties. Nous montrons en quoi les hypothèses que nous faisons pour la validité de cet algorithme sont recevables. Nous détaillons également les réglages à réaliser sur l'algorithme et nous indiquons comment cet algorithme est mis en œuvre dans l'architecture de notre application d'auto-optimisation.

Sommaire

L'objectif général de ce travail étant la définition et la réalisation de méthodes d'auto-optimisation, il est indispensable de développer des méthodes d'optimisation adaptées au contexte. Rappelons que notre travail s'inscrit dans le cadre de l'outillage logiciel

CLIF qui dispose déjà de fonctionnalités d'injection de charge auto-régulée (nommées ICAR) ICAR permet, pour un système S donné et réglé selon un ensemble E fixé de valeurs de paramètres fixées, de soumettre S à une charge croissante jusqu'à atteindre les niveaux maximaux d'indicateurs de fonctionnement (charge CPU, mémoire, utilisée, etc.) appelés objectifs de niveaux de service (*Service Level Objectives*, SLOs). La charge est constituée de multiples instances d'une même charge unitaire dont le profil est défini par l'utilisateur de CLIF et qui dépend du système à tester. ICAR fonctionnant dans le contexte de l'outil de benchmarking CLIF, l'utilisateur peut déterminer la valeurs des mesures collectées par CLIF et calculer par exemple un indice de performances, que nous appellerons *métrique* (notée m) dans la suite. Si le profil de charge unitaire est adapté à la recherche des variations de m, l'utilisation d'ICAR permet d'obtenir à son retour la valeur extrémale de m en respectant des SLOs donnés.

Un premier problème lié à l'auto-optimisation concerne l'articulation des deux activités, injection de charge auto-régulée et recherche de l'optimal de m selon les réglages du système. *A priori* deux choix sont possibles :

1. exécuter l'injection auto-régulée ICAR en modifiant les paramètres de fonctionnement pendant l'exécution d'ICAR ;
2. fixer les paramètres de S, exécuter ICAR et recommencer jusqu'à obtenir l'optimum de m.

En réalité le choix 1 n'est pas acceptable pour deux raisons. D'une part, nous avons vu 3.2.2 que certains paramètres de réglages ne peuvent être modifiés qu'avant le démarrage du système. Si l'un de ces paramètres est utilisé pour la recherche de l'optimum de m, le choix 1 est impossible. Si nous ne sommes pas dans ce cas, le phénomène de stabilisation reste une difficulté importante du choix 1. Lorsqu'on modifie les paramètres d'un système S en fonctionnement, on génère un changement, qui peut être rapide, dans l'exécution de S. Par exemple, accroître « fortement » et en un temps « court », la charge de S, provoque fréquemment un accroissement de la charge CPU de la machine d'exécution au delà de ce qu'elle sera lorsque le système sera stabilisé. Il faudrait donc gérer ce retour à la stabilité qu'il est difficile de quantifier à l'avance lors de chaque changement de valeur de paramètre. Nous avons donc décidé d'organiser les deux activités selon le schéma 2.

Il reste plusieurs problèmes à traiter concernant la recherche d'un réglage optimal. Le premier problème concerne le choix de l'indicateur m de qualité du réglage. Ce peut-être, de manière générale, un indicateur élémentaire de performance comme le temps de réponse ou le débit de requêtes servies ou un indicateur composite calculé à partir des précédents. Dans le contexte des applications multi-tiers que nous abordons dans ce travail, nous avons choisi le nombre d'utilisateurs servis simultanément par le serveur d'application.

Cependant, la solution que nous élaborons est tout à fait générique et tout indicateur mesurable peut être choisi.

Le second problème à étudier est relatif aux paramètres de réglage sur lesquels nous choisissons d'agir. Nous avons vu 3.2.2 que les applications que nous souhaitons régler dispose de plusieurs dizaines de paramètres de configuration/réglage. À partir de cet ensemble, nous définissons un ensemble, significativement réduit (de l'ordre de 20 ou 30 fois) de paramètres. C'est cet ensemble réduit que nous considérons dans le présent chapitre.

Même lorsque l'on réduit le nombre de paramètres sur lesquels nous intervenons pour optimiser l'application, nous restons confrontés au problème du nombre de tuples de valeurs possibles. De plus, pour un tuple potentiel de valeurs, nous devons nous assurer que le système peut être réglé avec ces valeurs, ce qui n'est pas systématique.

Enfin, le dernier problème est le choix de l'algorithme d'optimisation que nous utilisons pour obtenir la valeur maximale de l'indicateur m.

5.1 Caractéristiques du problème

5.1.1 Formalisation du problème d'optimisation

Nous donnons ici l'expression mathématisée du problème d'optimisation que nous traitons. Comme rappelé dans le chapitre 2, nous pouvons exprimer sans perte de généralité, notre problème sous forme de recherche de maximum, la recherche d'un minimum s'en déduisant à une ré-écriture près des comparaisons dans les algorithmes.

Nous supposons que nous avons un ensemble de n paramètres p_i, chacun prenant ses valeurs dans un ensemble P_i. Puisque chaque paramètre correspond à une valeur utilisée dans le système informatique de l'application sous test, nous supposons que P_i est fini pour tout i. Nous notons $P = \prod_{i=1}^{n} P_i$ et $\bar{p} = (p_1, \ldots, p_n)$ un tuple de P.

Nous utilisons comme indicateur de qualité de réglage, une mesure m qui peut être considérée comme une fonction de P dans $\mathbb{R}^+$ sans perte de généralité.

Le problème d'optimisation consiste donc à chercher la valeur $\bar{p}^{\text{opt}} \in P$ telle que :

$$\forall \bar{p} \in P_m, \;\; m(\bar{p}) \leq m(\bar{p}^{\text{opt}})$$

Rappelons que m est une *fonction* sur P. Son ensemble de définition, noté P_m, peut être strictement inclus dans P, ce qui correspond à des valeurs de paramètres incompatibles avec le fonctionnement de l'application.

Parmi les caractéristiques du problème d'optimisation défini ci-dessus, nous notons

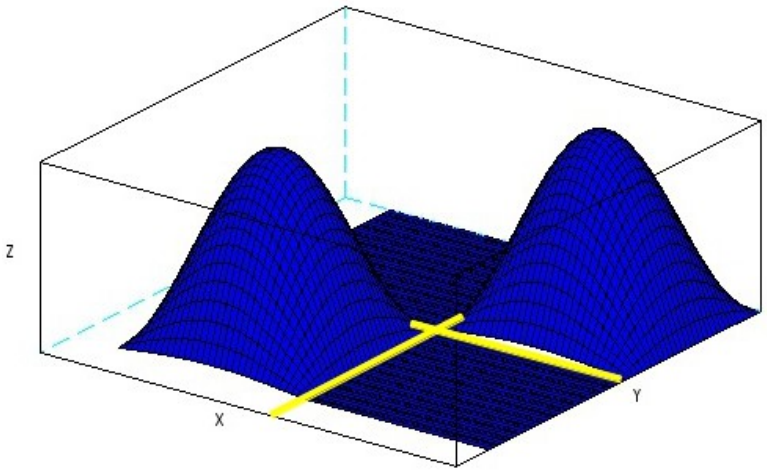

FIGURE 5.1: Exemple de fonction à plusieurs maxima locaux sous l'hypothèse 2

deux points importants à considérer : la nature de la fonction m et la complexité calculatoire du problème lui même.

5.1.2 Nature de la fonction à optimiser

Nous supposons dans la suite que m est uniquement fonction des paramètres ($\bar{p}$) de configuration du système testé, c'est à dire que l'environnement d'exécution du système sous test (voir 3.2.2) a été optimisé au préalable (équipements physiques, système d'exploitation, réseau, .etc).

Puisque m est obtenu par mesure sur le système sous test et que le comportement du système est fortement dépendant de la charge qui lui est soumis, il est très difficile de faire des hypothèses sur les propriétés mathématiques de m en tant que fonction de $\bar{p}$ pour un système quelconque.

Remarquons que notre fonction étant définie sur un ensemble P_m fini, elle possède toujours au moins un maximum sur cet ensemble. Une *première hypothèse* serait de supposer que *la fonction m admet un unique maximum local, qui est donc le maximum global, sur l'ensemble P_m.* Cette hypothèse est en fait très difficile à garantir. Ainsi, nous posons dans la suite l'hypothèse suivante (dénommée deuxième hypothèse). Nous supposons que *pour tout i, la fonction projection $m_{\bar{p}|i}$ définie ci-dessous possède un unique maximum sur son intervalle de définition.*

Définition 5.1.1 (Fonction projection) *Si m est une fonction de $P = \prod_{i=1}^{n} P_i$ dans*

B, on appelle projection de m sur P_i, et on note $m_{\bar{p}|i}$ la fonction de P_i dans B définie par

$$m_{\bar{p}|i}(x) = m(p_1, \ldots, p_{i-1}, x, p_{i+1}, \ldots, p_n)$$

Cette deuxième hypothèse implique que les fonctions $m_{\bar{p}|i}$ sont croissantes au sens large sur un premier intervalle $[\min_i, p_i^*]$, atteignent leur maximum en p_i^* puis sont décroissantes au sens large sur $[p_i^*$,
$athrmmax_i]$

Nous observons qu'une fonction m qui satisfait cette hypothèse peut posséder plusieurs maxima. Mais, grâce à la deuxième hypothèse, une fonction m ne peut admettre deux (ou plus) maxima locaux que si les antécédents des voisinages de ces maxima sont situés dans des polytopes d'hyperplans d'intersection vide. Ainsi, avec deux paramètres, la surface de la fonction m doit avoir une forme analogue à celle de la figure 5.1 : les points (p_1, p_2) autour des maxima sont dans des rectangles disjoints (séparés par les segments jaunes).

Si nous utilisons une fonction m qui possède plusieurs maxima locaux, tout en vérifiant notre hypothèse, nous avons un indicateur de réglage m qui définit deux (au moins) zones de fonctionnement du système sous test, chacune autour des réglages donnant le maximum local de m. Face à cette situation, il faut alors mettre en œuvre des stratégies permettant de passer d'un voisinage du maximum local courant à celui d'un autre. Diverse méthodes sont envisageables mais n'ont pas été explorées dans ce mémoire.

Remarquons enfin que si nous pouvons assurer l'unicité du maximum de m sur P_m, l'hypothèse sur les fonctions projections est vérifiée, la réciproque étant fausse (voir l'exemple ci-dessus).

5.1.3 Complexité du problème

Tout système informatique exécutant une application, aussi élémentaire soit-il, possède un nombre important de paramètres de configuration. Pour tout problème d'optimisation de performances, il est nécessaire d'identifier dans cet ensemble de paramètres de configuration, un sous ensemble qu'on appelle paramètres de réglage (*tuning parameters*), qui sont les paramètres ayant le plus grand impact sur les performances du système considéré, pour le problème d'optimisation donné. Nous avons ainsi deux problèmes à résoudre.

5.1.3.1 Détermination des paramètres de réglage

Il s'agit d'une part de choisir les paramètres significatifs intervenant dans le réglage du système. Ce choix, qui ne peut être complètement formalisé, est fondé sur une connaissance experte du système et de l'application qu'il exécute, du point de vue de leurs architectures

et de leurs comportements. Bien entendu, il est également nécessaire de connaître les méthodes d'évaluation de performances qui devront être utilisées. D'autre part, il faut, pour chaque paramètre de réglage p_i retenu, déterminer la plage de valeurs qu'il peut prendre. Nous supposons, comme c'est le cas dans les applications que nous avons pu analyser, que chaque ensemble P_i est un intervalle $[\mathrm{MIN}_i, \mathrm{MAX}_i]$ d'entiers. Notons que même si chaque P_i est un intervalle, l'ensemble des tuples de paramètres *admissibles* peut être un sous-ensemble strict (noté P_m) de $P = \prod_{i=1}^{n} P_i$. L'expression des conditions d'admissibilité d'un tuple peut être simple, par exemple $p_i < 2p_j$, ou complexe car issue d'une expertise humaine. Dans ce dernier cas, on est réduit à exclure explicitement les tuples « interdits » alors que dans le cas de relations du premier type, on peut envisager l'emploi de tout système algorithmique de vérification, par exemple un système solveur de contraintes.

5.1.3.2 Taille de l'espace de tuples

Même lorsque l'on a réduit le nombre de paramètres à un ensemble de l'ordre de 10 paramètres de réglage, le nombre de tuples possibles reste important (plusieurs centaines) ce qui, couplé au caractère expérimental et long (plusieurs minutes) de l'obtention de la valeur $m(\bar{p})$, mène à des temps d'exécution prohibitifs. Pour fixer les idées, donnons un exemple d'optimisation manuelle. Le processus complet se compose d'une succession de tests et de paramétrages du système, d'une durée moyenne de 10 minutes environ.

Admettons que nous n'ayons que 2 paramètres p_1 et p_2 de réglage du système, qui prennent leurs valeurs entières dans un domaine de 1 à 500, avec une contrainte simple ($p_1 \leq p_2$). Le nombre de solutions candidates pour ce problème est de 125000. Si l'on procède à la mesure de m pour chaque tuple de réglage, il faut 125000 x 10 = 1250000 minutes, soit plus de 868 jours, pour être sûr d'avoir la meilleure configuration. Or, dans un contexte d'optimisation d'un système effectif avant mise en production, il est indispensable que le réglage du système soit réalisé dans un temps compatible avec les contraintes de fonctionnement d'un service commercial. Ainsi, si des durées de réglage de l'ordre du jour sont acceptables, ce n'est clairement pas le cas pour des durées de plusieurs semaines.

Afin de tenir compte de l'ensemble de ces propriétés, nous proposons de rechercher l'optimum de la mesure m, à l'aide d'un algorithme constitué de deux phases imbriquées. La première phase, que nous appelons algorithme général, contrôle le balayage de l'ensemble des tuples de P_m, en lançant la recherche, pour chaque composante, de la meilleure valeur de m. La seconde phase, appelée algorithme élémentaire, recherche cette meilleure valeur en configurant le système sous test et en lançant l'opération ICAR d'auto-optimisation contrôlée.

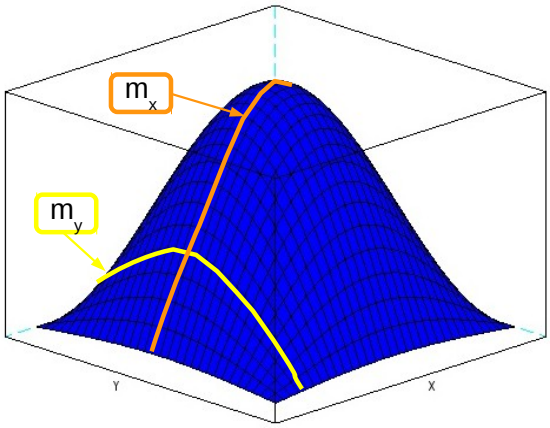

FIGURE 5.2: Illustration de l'algorithme de recherche en croix avec deux paramètres

5.2 Algorithme général (en croix)

5.2.1 Principe

L'algorithme général 1, que nous appelons aussi *algorithme de recherche en croix*, s'inspire de la méthode de recherche de personnes lors d'avalanches. Il suppose que l'on se situe dans un « voisinage » du maximum de la fonction m. Cet algorithme peut être considéré comme appartenant aux méthodes d'ascension de colline (*hill climbing*, [RN04] pages 111 et suiv.). Le principe consiste à s'intéresser successivement à chacun des paramètres $p_1, \ldots, p_n$ de réglage, les valeurs des autres paramètres étant fixées. Pour chaque paramètre, on effectue une recherche de maximum présentée dans la section suivante (algorithme élémentaire).

Partant d'une valeur du tuple $(p_1^{(k)}, \ldots, p_n^{(k)})$, nous mettons à jour la $i$ème composante de ce tuple à chaque recherche de maximum sur p_i. Par exemple, sur la figure 5.2, la projection m_y (courbe en jaune) permet de déterminer un premier extremum et la valeur de y correspondante. Pour *cette* valeur de y, nous cherchons le maximum de la projection m_x (courbe en orange). À l'issue des n recherches élémentaires, nous obtenons un nouveau tuple $(p_1^{(k+1)}, \ldots, p_n^{(k+1)})$ qui fournit une valeur $m^{(k+1)}$ plus grande que $m^{(k)}$. On reprend alors la recherche des maxima pour les fonctions projections en partant de $(p_1^{(k+1)}, \ldots, p_n^{(k+1)})$. L'exemple de la figure 5.2 est un cas simple et nous obtenons le maximum de m en une seule passe sur x et y. Dans le cas général, une passe peut conduire à un sorte de « palier » de m et donc nécessiter de recommencer la recherche. À l'issue de l'exécution de l'algorithme, nous obtenons un tuple de paramètres de réglage, que nous notons $\bar{p}^{opt}$, donnant la valeur maximale de la métrique m.

Algorithm 1: Algorithme général : recherche en croix

$\bar{p}_p^{\text{opt}} := (\text{MIN}_1, \text{MIN}_2, \ldots, \text{MAX}_n)$; 1

$\bar{p}_c^{\text{opt}} := (\frac{\text{MIN}_1+\text{MAX}_1}{2}, \text{MIN}_2, \ldots, \text{MIN}_n)$; 2

$m_p := \text{ICAR}(\bar{p}_p^{\text{opt}})$; 3

$m_c := \text{ICAR}(\bar{p}_c^{\text{opt}})$; 4

$\text{varM} := \frac{|m_c - m_p|}{m_p}$; 5

$\text{varP} := \frac{\|\bar{p}_c^{\text{opt}} - \bar{p}_p^{\text{opt}}\|}{\|\bar{p}_p^{\text{opt}}\|}$; 6

$k := 1$; 7

while $((k = 1)$ *or* $(\text{varP} > \epsilon_1)$ *or* $(\text{varM} > \epsilon_2))$ *and* $(k \leq k_{\max})$ **do** 8

  $m_p := m_c$; 9

  $\bar{p}_p^{opt} := \bar{p}_c^{opt}$; 10

  for $i := 1$ ***to*** n **do** 11

    $(\bar{p}_{c,i}^{opt}, m_c) := \text{findbest}(i, \bar{p}_c^{opt}, \text{MIN}_i, \text{MAX}_i, g_i)$; 12

  end 13

  $\text{varM} := \frac{|m_c - m_p|}{m_p}$; 14

  $\text{varP} := \frac{\|\bar{p}_c^{\text{opt}} - \bar{p}_p^{\text{opt}}\|}{\|\bar{p}_p^{\text{opt}}\|}$; 15

  $k := k + 1$; 16

end 17

return $\bar{p}_c^{\text{opt}}$; 18

5.2.2 Réglage de l'algorithme

Les éléments de réglages de l'algorithme sont d'une part le choix de l'ordre des calculs d'optimum des fonctions projection, c'est à dire l'ordre d'indexation des paramètres, et d'autre part les valeurs des contrôles de la boucle tant que.

5.2.2.1 Ordre de paramètres

Il est connu que l'ordre de parcours des éléments x d'un ensemble fini X lors de la recherche d'une valeur particulière d'une fonction f sur X par énumération de X peut modifier considérablement le comportement de la recherche. Dans notre cas, l'ensemble des paramètres de réglages a déjà été choisi comme pertinent pour l'optimisation du système sous test. Il semble donc délicat d'établir un ordre (total) entre les influences de ces paramètres. Nous avons donc choisi l'ordre suivant.

Nous appelons *Domaine de p*, noté D_p, l'ensemble des valeurs de p à examiner *a priori*. Nous détaillons ci-dessous comment est déterminé D_p.

Nous posons alors qu'un paramètre p a un indice inférieur à un paramètre p' si et seulement si le cardinal de D_p (noté $|D_p|$) est inférieur à celui de $D_{p'}$. Autrement dit, p est avant p' si et seulement si le nombre de valeurs qui seront examinées *a priori* pour p est plus petit que celui pour p'. La détermination de $|D_p|$ est fondée sur la granularité de p présentée ci-dessous. Nous avons :

$$|D_{p_i}| = \frac{\mathrm{MAX}_i - \mathrm{MIN}_i}{g_i}$$

5.2.2.2 Finesse de parcours

Les valeurs de chacun des paramètres p_i constituent l'ensemble $P_i = [\min_i, \max_i]$. Potentiellement, chaque valeur de cet intervalle peut être choisie pour p_i. Par expérience, les variations de m selon p_i ne sont pas sensibles en dessous d'une valeur de l'ordre de 5 à 10 % de $\mathrm{MAX}_i - \mathrm{MIN}_i$. On définit alors le pas de variation, ou *granularité*, g_i de p_i par

$$g_i = \frac{\mathrm{MAX}_i - \mathrm{MIN}_i}{\delta_i}$$

où δ_i est le taux de sensibilité de m pour p_i. Par exemple, pour un paramètre dans $[100, 1300]$, en prenant $\delta_i = 10$, on obtient $g_i = 120$.

5.2.2.3 Critères d'arrêt

La boucle tant que de l'algorithme 1 est contrôlée par les conditions de la ligne 8. Nous avons deux conditions à respecter : une relative aux évolutions des valeurs obtenues dans le corps de boucle, l'autre limitant le nombre de corps de boucle exécutés ($k \leq k_{\max}$).

La condition principale porte sur les variations de $\bar{p}_p^{opt}$ et m. Chacune de ces grandeurs doit évoluer significativement lors de l'exécution d'un corps de boucle. Une évolution est déclarée significative si elle est supérieure à un seuil donné ϵ_1 (ou ϵ_2). Pour le vecteur $\bar{p}_p^{opt}$, nous employons une distance fondée sur une norme de $\mathbb{R}^n$. Les valeurs de ϵ_1 et ϵ_2 ont été fixées après expérimentations (voir le chapitre 7). Notons qu'il est nécessaire de repérer les variations *des deux grandeurs*, et pas uniquement de m ou de $\bar{p}_p^{opt}$. En effet, si nous nous restreignons aux variations de m, nous risquons d'arrêter la recherche de maximum parce que m varie peu, alors que $\bar{p}_p^{opt}$ varie encore « fortement » laissant ouverte la possibilité de trouver une meilleure valeur de m : nous sommes sur un « plateau » de m. À l'inverse si $\bar{p}_p^{opt}$ varie peu, m peut, elle, varier fortement : nous sommes proche d'un « pente prononcée » de m.

La valeur de $k_{\max}$ dépend *a priori* de la variabilité de m. Nous exécutons plusieurs fois l'algorithme pour obtenir un ordre de grandeur de $k_{\max}$ (voir le chapitre 7).

5.3 Algorithme élémentaire

5.3.1 Principe

Cet algorithme (2) a pour fonction de retourner la valeur p_i^* du paramètre p_i qui correspond à l'unique maximum de la fonction projection $m_{\bar{p}|i}$ en parcourant un ensemble de valeurs entières dans l'intervalle $[\min_i, \max_i]$. Rappelons que les valeurs de tous les autres paramètres $j \neq i$, sont fixées à p_j. Puisque le temps de calcul de m est significatif, l'algorithme doit minimiser les appels à ce calcul, en évitant par exemple l'algorithme naïf qui calcule les valeurs $m_{\bar{p}|i}(x)$ pour tous les x de $[\min_i, \max_i]$.

Grâce à l'hypothèse 2 d'unique maximum de $m_{\bar{p}|i}$ sur $[\min_i, \max_i]$, l'algorithme récursif est construit comme un algorithme de recherche dichotomique de zéro d'une fonction continue f dans un intervalle $[a, b]$ avec $f(a) < 0$ et $f(b) > 0$. Le principe est le suivant. Étant donné un intervalle $[a, b]$ de recherche de la valeur p_i^*, on essaie de décider si p_i^* se situe dans $[a, \frac{a+b}{2}]$ ou dans $[\frac{a+b}{2}, b]$. Le choix de l'un de ces deux intervalles est détaillé ci-dessous. Nous distinguons 6 situations illustrées dans les figures 5.3, 5.4 et 5.5. Dans cette section, nous utilisons m en lieu et place de $m_{\bar{p}|i}$ pour simplifier la notation.

La figure 5.3 illustre les trois premiers cas fondamentaux, indiquant la position de

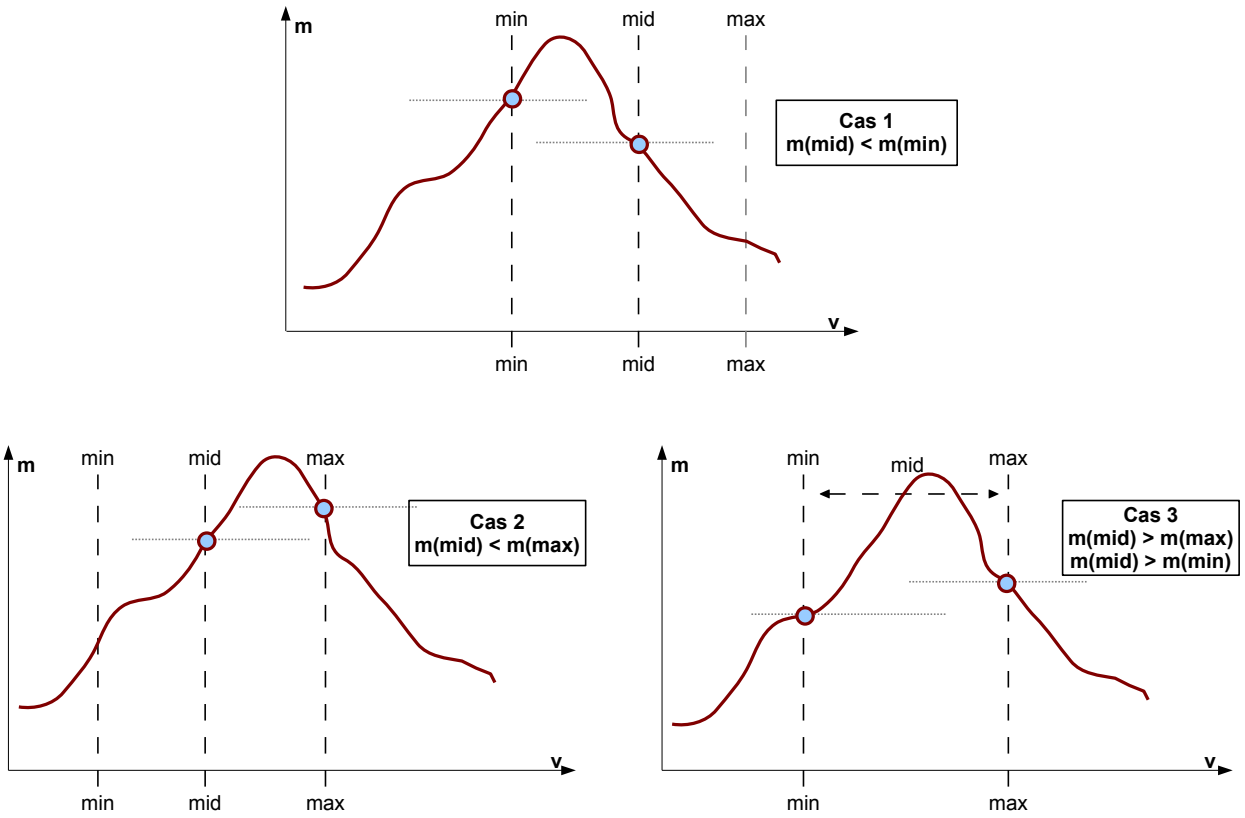

FIGURE 5.3: Algorithme élémentaire - cas 1, 2 et 3

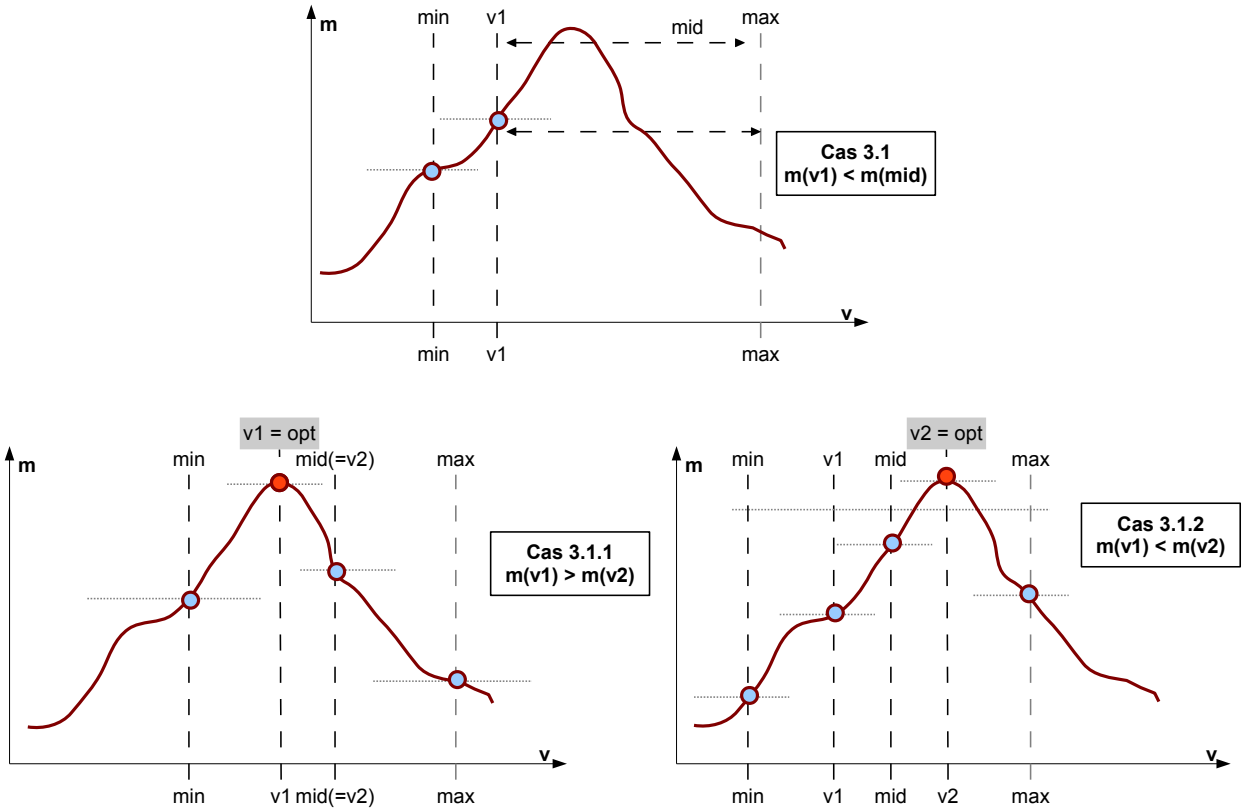

FIGURE 5.4: Algorithme élémentaire - cas 3.1, 3.1.1 et 3.1.2

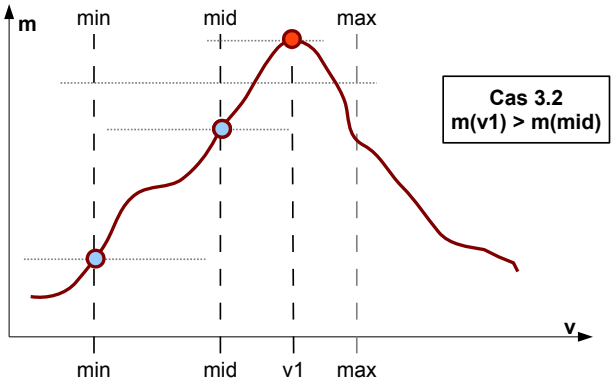

FIGURE 5.5: Algorithme élémentaire - cas 3.2

$m(\text{mid})$ par rapport à celles de $m(\text{min})$ et $m(\text{max})$.

Dans le cas 1, $m(\text{mid}) < m(\text{min})$ (ligne 7 de l'algorithme 2). L'hypothèse de maximum unique de m implique que p^* est nécessairement dans l'intervalle $[\text{min}, \text{mid}]$.

Dans le cas 2, nous avons $m(\text{mid}) < m(\text{max})$ (ligne 11 de l'algorithme 2). De plus nous sommes dans la partie sinon du cas 1, donc $m(\text{mid}) > m(\text{min})$. On est donc assuré que p^* se trouve entre mid et max.

Enfin le cas 3 (lignes 13 et suivantes de l'algorithme 2) correspond à $m(\text{mid}) > m(\text{min})$ et $m(\text{mid}) > m(\text{max})$. Cette situation nous donne le moins d'informations : il faut calculer de nouvelles valeurs de m par des appels récursifs conduisant aux cas 3.1 et 3.2 (figures 5.4 et .5.5).

La figure 5.4 illustre les deux cas de comparaisons correspondant au tests qu'il faut faire à gauche et à droite de *mid*, correspondants aux sous-cas du cas 3, afin de guider la suite de la recherche de p^*.

Le cas 3.1 correspond à la ligne 15 de l'algorithme 2. Après avoir calculé $v1 = findbest(i, \bar{p}, \text{min}, \text{mid}, m)$, si $m(v1) < m(mid)$, p^* n'est pas dans $[\text{min}, \text{mid}]$, il faut encore calculer $v2 = findbest(i, \bar{p}, \text{mid}, \text{max}, m)$. Nous obtenons alors les deux situations 3.1.1 et 3.1.2 qui correspondent aux lignes 17 et 19 respectivement.

Le cas 3.2 (figure 5.5, ligne 22 de l'algorithme 2) est la branche sinon du cas 3.1. Dans cette situation, on sait que $v1$ est l'optimum de m.

Enfin, la récursion est arrêtée par la condition $((max - min) \leq g)$ indiquant que nous passons en-dessous du degré de différentiation entre deux valeurs du paramètre p_i (ligne 3 de l'algorithme 2). Dans ce cas, nous retournons l'une des deux valeurs de m, à savoir celle dont on est sûr qu'elle a été calculée. Notons que puisque nous réduisons strictement la valeur entière $(\text{max} - \text{min})$ qui est strictement positive, nous sommes assurés que la condition d'arrêt finira pas être satisfaite.

5.3.2 Algorithme détaillé

L'algorithme 2 correspond aux cas présentés dans les figures ci-dessus. Le calcul de la valeur de $m_{\bar{p}|i}(x)$ est réalisé par l'appel à la fonction ICAR$(\bar{p}, i, x)$.

Nous avons constaté que plusieurs appels à ICAR$(\bar{p}, i, x)$ sont réalisés avec les mêmes valeurs du vecteur $\bar{p}$ et de x. Puisque le temps d'expérimentation pour obtenir $m(\bar{p})$ est significatif (plusieurs ordres de grandeurs plus long que l'exécution d'une instruction élémentaire), il est très efficace d'utiliser un mécanisme de cache pour éviter un recalcul de $m(\bar{p})$ si cette valeur a déjà été obtenue. L'implantation de ce mécanisme conduit à l'algorithme 3 pour la fonction ICAR. Le principe est simple. les couples $(\bar{p}, m(\bar{p}))$ sont

Algorithm 2: Algorithme élémentaire

```
Function (int, double) findbest(i, p̄, min, max, g);                          1
m_min := ICAR(p̄, i, min) ;                                                   2
if ((max − min) > g) then                                                    3
    mid := (min + max)/2 ;                                                   4
    m_mid := ICAR(p̄, i, mid);                                                5
    if (m_min > m_mid) then                                                  6
        return findbest(i, p̄, min, mid, g) ;                                 7
    else                                                                     8
        m_max := ICAR(p̄, i, max) ;                                           9
        if (m_mid < m_max) then                                              10
            return findbest(i, p̄, mid, max, g) ;                             11
        else                                                                 12
            (v1, m_v1) := findbest(i, p̄, min, mid, g) ;                      13
            if (m_v1 < m_mid) then                                           14
                (v2, m_v2) := findbest(i, p̄, mid, max, g) ;                  15
                if (m_v1 > m_v2) then                                        16
                    return (v1, m_v1)                                        17
                else                                                         18
                    return (v2, m_v2)                                        19
                end                                                          20
            else                                                             21
                return (v1, m_v1)                                            22
            end                                                              23
        end                                                                  24
    end                                                                      25
else                                                                         26
    return (min, m_min)                                                      27
end                                                                          28
```

Algorithm 3: Fonction d'appel à ICAR

```
Function int ICAR(p̄, i, v);                                                  1
p̄[i] := v;                                                                   2
if (p̄, m) is not in the ICARCache then                                       3
    ReconfigureSUT(p̄);                                                       4
    m = runICAR;                                                             5
    Put m in ICARCache with key p̄;                                           6
else                                                                         7
    Retrieve m from ICARCache for key p̄;                                     8
end                                                                          9
return m;                                                                    10
```

rangés dans un cache associatif dont les clés sont les vecteurs $\bar{p}$. À chaque demande de $m(\bar{p})$, on vérifie si cette valeur est dans le cache. Dans le cas où il faut effectivement calculer $m(\bar{p})$, on configure le système sous test avec les paramètres $\bar{p}$, puis on lance la boucle d'auto-optimisation ICAR. Nous verrons dans le chapitre 7, l'efficacité de ce mécanisme de cache.

5.4 Complexité de l'algorithme

Nous avons introduit l'algorithme en croix pour réduire le nombre d'appels au mécanisme ICAR de recherche des meilleurs performances avant saturation. Nous quantifions ici les gains réalisés par rapport à la méthode exhaustive de recherche de l'optimum de la fonction objectif m. Nous nous concentrons sur le nombre d'appels à ICAR pour cette étude de complexité temporelle puisque nous avons vu que l'ordre de grandeur du temps d'exécution de ICAR est de 5mn au moins.

Rappelons que pour chaque paramètre p_i, nous avons un intervalle de valeurs entières $[\mathrm{MIN}_i, \mathrm{MAX}_i]$ et une granularité g_i de sorte que le nombre effectif de valeurs utilisées par la recherche exhaustive pour p_i est au plus

$$U_i = \left\lfloor \frac{\mathrm{MAX}_i - \mathrm{MIN}_i}{g_i} \right\rfloor$$

à plus ou moins un près ($\lfloor a \rfloor$ désigne la partie entière de a). Il est possible que ce nombre d'appels soit inférieur à U_i si des tuples $\bar{p}$ correspondent à des valeurs interdites de réglage du système (voir la section suivante). Le nombre de tuples parcourus dans le cas exhaustif est donc au plus

$$U = \prod_{i=1}^{n} U_i$$

Dans notre algorithme en croix, examinons le nombre d'appels à ICAR. Pour un indice i donné, la fonction findbest doit au pire calculer trois valeurs : $m(\mathrm{min}_i), m(\mathrm{mid})$ et $m(\mathrm{max}_i)$, bien entendu $\mathrm{MIN}_i = \mathrm{min}_i$ et $\mathrm{MAX}_i = \mathrm{max}_i$ au premier appel de findbest uniquement. En fait, sauf au premier appel, la valeur $m(\mathrm{min}_i)$ a déjà été calculée. Nous avons donc au plus deux appels à ICAR par appel de findbest. La fonction findbest est récursive et la condition d'arrêt $\mathrm{max}_i - \mathrm{min}_i \leq g_i$ nous donne le nombre a_i d'appels récursifs à findbest :

$$\frac{\mathrm{MAX}_i - \mathrm{MIN}_i}{2^{a_i}} \leq g_i$$

Donc, a_i satisfait l'inégalité

$$a_i \leq 1 + 2\ln_2\left(\frac{\mathrm{MAX}_i - \mathrm{MIN}_i}{g_i}\right) = 1 + 2ln_2(U_i)$$

où $\ln_2$ désigne le logarithme en base 2. Nous avons une inégalité car, grâce au mécanisme de cache, il est possible que la valeur $m(\bar{p})$ soit présente dans le cache, évitant ainsi un appel effectif à ICAR.

Une exécution du corps de la boucle `while` de l'algorithme en croix comporte les n appels récursifs à findbest correspondant à chaque paramètre p_i. Le nombre d'appels à ICAR d'une telle exécution est donc au plus

$$a = \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (1 + 2ln_2 U_i)$$

Si l'algorithme en croix exécute K corps de boucle `while`, le nombre d'appels à ICAR est donc au plus

$$A = Ka$$

L'ordre de grandeur du gain G en nombre d'appels à ICAR en utilisant l'algorithme en croix est donc

$$G = \frac{U}{A} \approx \frac{U}{2K\ln_2 U} = \frac{1}{2K}\frac{U}{\ln_2 U}$$

Pour fixer les idées, si nous avons 5 paramètres dont chacun peut prendre 10 valeurs utilisées par les appels à ICAR, nous obtenons $U = 10^5$, $\ln_2 U = 13.29$ et $\frac{U}{\ln_2 U} = 752.57$ (arrondi à deux chiffres après la virgule). Donc $G \geq \frac{1}{2K}752$ reste strictement supérieur à 1 pour tout $K \leq 376$. Nous verrons également dans le chapitre 7 que les valeurs de K rencontrées sont de l'ordre de 10, et donc que le gain de notre algorithme est significatif (au moins plusieurs ordres de grandeur).

5.5 Prise en compte des contraintes

Nous avons vu que nous recherchons le tuple $\bar{p} \in P_m$ qui maximise la fonction m, où P_m est un sous-ensemble de l'ensemble $P = \prod_{i=1}^{n} P_i$ de tous les tuples de paramètres possibles. Les tuples de P P_m correspondent à des valeurs de réglage incohérentes ou qui sont interdites. Par exemple, dans le contexte des serveurs d'applications Java EE, considérons les tailles des groupes de threads (*Thread pools*). On distingue au moins trois groupes :

– le groupe de threads du connecteur WEB (WTG) ;

– le groupe des EJBs (ETG) ;
– le groupe des connexions au(x) serveur(s) de base de données (CTG).

On peut imaginer une application dans laquelle 60% des requêtes des clients produisent des requêtes serveur WEB, conteneur d'EJBs et accès aux bases de données et 40 % uniquement des requêtes sur le serveur WEB. Dans cette situation, il est souhaitable de dimensionner chaque groupe de threads de sorte que : $|\text{ETG}| \geq 0.4|\text{WTG}|$ et $|\text{ETG}| = |\text{CGT}|$.

L'existence de contraintes entre les paramètres incite à utiliser une méthode de type Problème de satisfaction de contraintes (*Constraint Satisfaction Problem,* CSP). Au plan algorithmique, après avoir défini un tuple de paramètres et avant l'appel au test à l'aide d'ICAR, il faut vérifier la validité de ce tuple. Ceci peut être réalisé à l'aide d'un moteur de règles autonome ou avec un solveur de contraintes comme Choco [CJLR06] pour lequel on utilise uniquement la vérification de la consistance du tuple. Notons que le temps de vérification (de l'ordre de quelques milli-secondes) n'a pas d'impact significatif sur l'ensemble de la méthode d'optimisation, le temps d'un test de charge ICAR étant de l'ordre de 10 minutes. Cette fonctionnalité de vérification n'est actuellement pas implantée dans notre logiciel.

La recherche de l'optimum d'une fonction d'un tuple de paramètres qui satisfont un ensemble de contraintes nous a conduits, au cours de nos travaux, à envisager l'emploi d'un système d'optimisation sous contraintes. En réalité, ce type de méthode n'est pas adaptée à notre contexte. En effet, les méthodes d'optimisation sous contraintes se fondent à la fois sur des parcours d'arbre comme les méthodes CSP, et sur les propriétés de la fonction à optimiser. Mais ces propriétés doivent s'énoncer sur des sous-arbres dont on ne connaît que le chemin depuis la racine. C'est à dire que l'on doit pouvoir énoncer des propriétés de la fonction, pour en ensemble de tuples dont on connait seulement la valeur de certaines composantes. Or, dans notre cas, nous n'avons aucune information de cette nature, la valeur de la fonction étant obtenue par mesure sur une expérimentation avec le système sous test. Nous avons donc renoncé à l'emploi des méthodes d'optimisation sous contraintes.

5.6 Intégration de l'algorithme à ICAR

La figure 5.6 donne la séquence des interactions entre l'algorithme et le système d'injection de charge auto-régulée. Tout d'abord, l'algorithme fait appliquer le paramétrage initial au système sous test (nous verrons dans le chapitre suivant comment cela est réalisé). Puis, l'algorithme demande une évaluation des performances du système au système

ICAR. Celui-ci commence par demander à la boucle de contrôle de la saturation de surveiller le système. Lorsque la surveillance est effective, un message d'accusé permet à la boucle de contrôle de l'injection de démarrer le test ICAR.

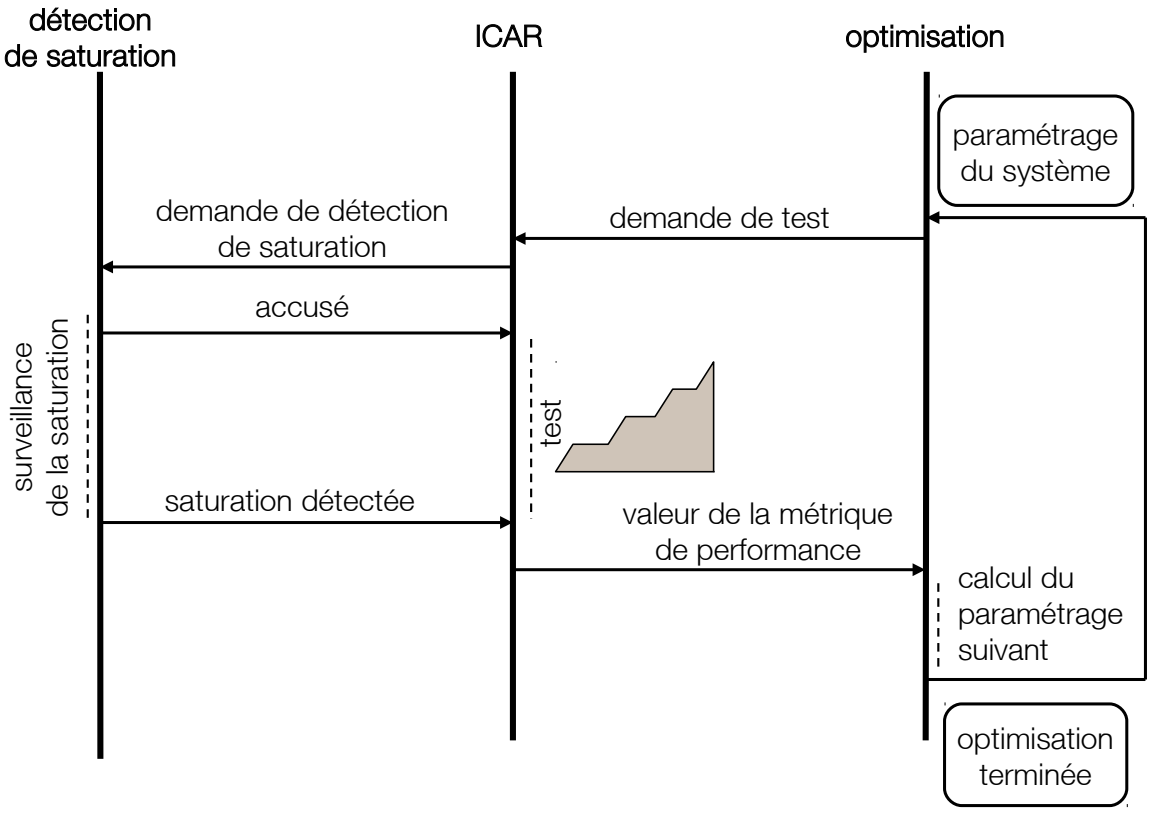

FIGURE 5.6: Principe de l'intégration du système d'injection de charge auto-régulé dans l'algorithme

La montée en charge s'effectue jusqu'à ce que la boucle de contrôle de saturation détecte la saturation du système. Lorsque cet événement survient, ICAR stoppe le test et renvoie à l'algorithme le nombre d'utilisateurs virtuels atteint par le palier précédent la saturation. Ensuite, en fonction de la valeur obtenue et conformément à l'algorithme 1, soit l'algorithme se termine, soit un nouveau paramétrage est calculé, appliqué puis soumis à évaluation par ICAR.

Dans le chapitre suivant, nous détaillons l'aspect architectural de notre plate-forme de Self-benchmarking et l'implémentation de la politique d'optimisation présentée ici.

Chapitre 6

Architecture logicielle de la plate-forme de Self-benchmarking

Nous présentons l'architecture logicielle de la plate-forme de Self-benchmarking que nous proposons, et nous détaillons les différents composants Fractal qui la composent. Nous abordons également la problématique de la communication entre les composants contrôleurs, nécessaire à la coordination des boucles de contrôle correspondantes.

Sommaire

Après avoir présenté la politique d'optimisation proposée pour résoudre notre problème d'auto-optimisation, nous détaillons à présent l'implémentation de cette politique d'optimisation et la mise en œuvre des différents composants servant à accomplir la tâche de benchmarking automatisé. Pour cela, nous présentons d'abord l'architecture globale de la plate-forme, puis nous détaillons les nouveaux composants introduits, ainsi que les aspects communication et coordination entre les boucles de contrôle.

6.1 Architecture de l'application d'auto-optimisation

6.1.1 Introduction

Globalement, notre contribution vise à compléter la plate-forme de test de performance par injection de charge existante, en introduisant un mécanisme de reparamétrage du système testé, et ce, dans le but d'améliorer ses performances. Le principe général est d'appliquer une méthode classique de génération-évaluation de paramétrages. L'algorithme d'optimisation 1 est utilisé pour générer les paramétrages possibles, et le processus de test ICAR (cf. section 4.4) sert à l'évaluation de chaque paramétrage du système sous test.

Sur le plan architectural, ce principe se traduit tout d'abord par l'introduction d'un composant `Optimizer` (*optimiseur)* dédié au calcul des paramétrages à tester. Il doit être capable d'appliquer un paramétrage au système à optimiser, puis de déclencher le processus de test ICAR et d'en attendre le résultat avant de calculer un autre paramétrage.

Afin de conférer un caractère générique à l'optimiseur, i.e. une indépendance totale vis-à-vis du système à optimiser, nous introduisons des composants `Configurator` (*configurateurs*) pour prendre en charge le changement de paramétrage. En fonction des modalités de paramétrage du système, il peut y avoir plusieurs sortes de configurateurs : modification de fichiers de configuration selon différents formats, utilisation de protocoles d'administration ou d'interface programmatique spécifique, etc. De plus, il peut s'avérer nécessaire de redémarrer tout ou partie du système à optimiser pour que la nouveau paramétrage soit effectivement pris en compte. Nous introduisons alors des composants `Starters` (*démarreurs*) afin de couvrir ce besoin de manière spécifique à chaque système.

6.1.2 Exploitation du type "lame" de CLIF

Une question importante se pose : devons-nous étendre CLIF en y introduisant de nouveaux types de composant afin de pouvoir introduire les configurateurs et les démar-

reurs ? Cela paraîtrait a priori naturel, mais aurait de sérieux inconvénients, car on ne pourrait plus bénéficier des mécanismes de gestion de cycle de vie standard (déploiement, monitoring, contrôle d'exécution) sans modification du cœur de CLIF, avec des impacts sur les divers outils et interfaces utilisateurs.

En définitive, aucune modification de CLIF n'est nécessaire. En effet, le cœur de CLIF est basé sur un type de composant générique `Blade` (*lame*) [Dil09], dont les interfaces offertes et requises vont entièrement répondre à nos besoins. Le type lame montre que CLIF est conçu comme un canevas logiciel pour le déploiement et le contrôle de composants de calcul parallèle, les lames étant considérées comme des éléments actifs que l'on peut déployer, et dont on peut observer et contrôler l'exécution, avant de récupérer les résultats par une opération de "collecte" en fin d'exécution. Ainsi, de la même façon que les sondes et les injecteurs, les configurateurs et les démarreurs seront de simples lames, avec des rôles particuliers se traduisant par une implantation spécifique de certaines méthodes.

Cette exploitation opportuniste du type lame nous permet de bénéficier de toutes les fonctionnalités de CLIF standard pour nos nouveaux composants. Les tests déployés sont donc tout simplement composés d'injecteurs, de sondes, de configurateurs et de démarreurs. Le cas du contrôleur d'optimisation est traité à part, de la même façon que les contrôleurs d'injection et de saturation du module CLIF/Selfbench.

6.1.3 Architecture globale

La figure 6.1 montre une vue globale des différents composants de l'architecture que nous proposons. Le composant *Supervisor* est le composant central du canevas logiciel CLIF, qui sert à la supervision de la plate-forme de test. Par le biais de ses interfaces `Data collector administration` et `Supervisor information`, il dispose en permanence d'informations sur l'avancement du test, l'état des injecteurs et des sondes, ainsi que sur leurs mesures par l'intermédiaire de statistiques glissantes disponibles à la demande. Son interface `Blade control` lui permet de piloter l'activité des sondes et des injecteurs, et de les reparamétrer à chaud. Son interface serveur `Test control` donne accès à toutes ces informations et fonctionnalités de manière centralisée.

Les deux composants en bas du schéma sont le composant de contrôle de l'injection et le composant de détection de saturation. Ils sont issus du module Selfbench de CLIF, qui implémentent à travers deux boucles de contrôle distinctes le processus de test ICAR. Ils sont du type Fractal `ControllerType`, qui spécifie une unique interface client de type `Test control`, qui permet ainsi aux contrôleurs de contrôler, observer et reconfigurer l'ensemble des lames déployées, en l'occurrence les sondes pour le contrôleur de saturation, et les injecteurs pour le contrôleur d'injection. Quant au composant d'optimisation,

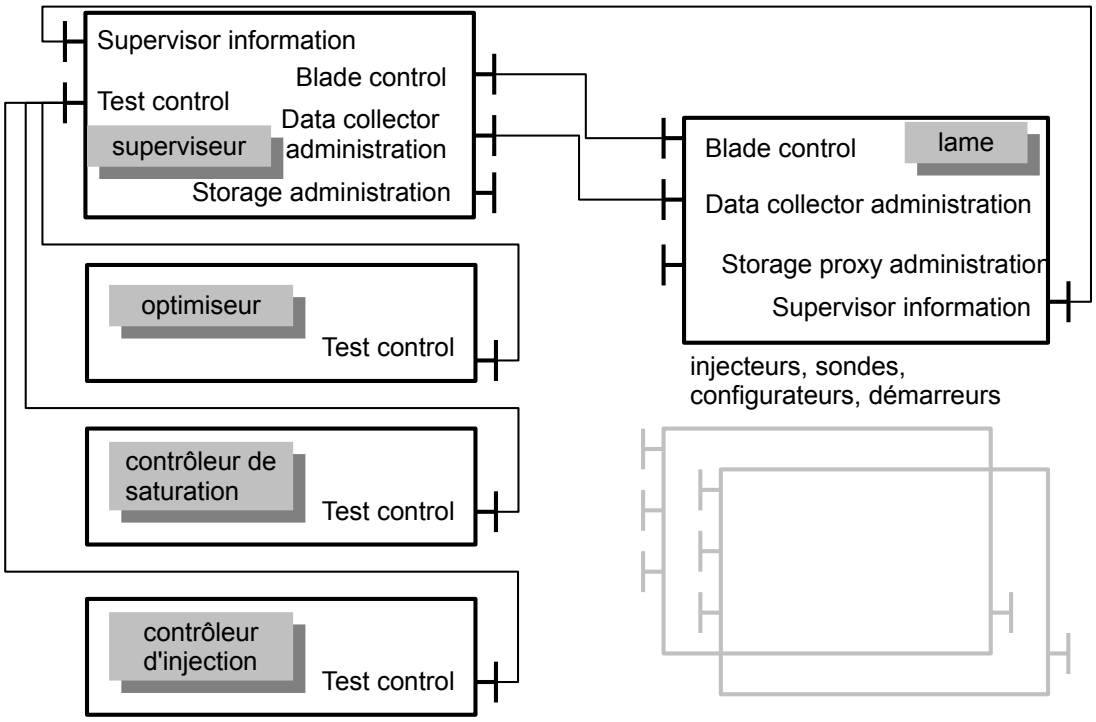

FIGURE 6.1: Vue globale des différents composants de l'architecture d'auto-optimisation

il implémente le même type Fractal `ControllerType`. Également relié au composant superviseur par l'interface `Test control`, il peut piloter les configurateurs et les démarreurs afin d'appliquer les paramétrages successivement calculés.

6.2 Le composant optimiseur

Le composant contrôleur d'optimisation a pour rôle de générer différents paramétrages du système à optimiser, à partir :

- d'une description des paramètres et des valeurs possibles (bornes, granularité),
- de l'algorithme 1,
- des résultats retournés par ICAR.

Afin d'appliquer un paramétrage, l'optimiseur fait appel aux configurateurs, et plus particulièrement à leur méthode `changeParameter()`, telle que définie par l'interface `Test control` du superviseur. Cette interface consiste essentiellement en une délégation vers l'interface `Blade control` de chaque lame (voir 6.1). La sémantique de cette méthode est de changer un couple clé-valeur, sensé représenter un paramètre de la lame. La modifica-

tion de ce paramètre est prise en compte immédiatement.

Listing 6.1 – Extrait de la définition des interfaces Test control (superviseur) et Blade control (lames) du canevas logiciel CLIF

```
public abstract interface ActivityControl
{
        ...
        /**
         * suspension de l'activité
         */
        public void suspend();

        /**
         * reprise de l'activité
         */
        public void resume();
}

public interface BladeControl extends ActivityControl
{
        ...
        /**
         * Change la valeur du paramètre indiqué avec la valeur donnée.
         */
        public void changeParameter(String name, Serializable value)
        throws ClifException;
}

public interface TestControl extends ActivityControl
{
        ...
        /**
         * délégation à BladeControl.changeParameter(name, value) de la lame
         * désignée par bladeId
         */
    public void changeParameter(String bladeId, String name, Serializable value)
    throws ClifException;

        /**
         * suspension de l'activité de la lame désignée par bladeId
         */
        public void suspend(String bladeId);

        /**
         * reprise de l'activité de la lame désignée par bladeId
         */
        public void resume(String bladeId);
}
```

L'utilisation de cette méthode par l'optimiseur est donc assez immédiate : le couple clé-valeur passé en argument à l'appel de la méthode donne le paramètre à modifier et sa

nouvelle valeur. Le configurateur concerné est spécifié à travers son identifiant de lame, tel qu'indiqué dans la définition du plan de test CLIF, et dans la description du paramètre donnée à l'optimiseur. De cette façon, l'identification du paramètre et la modalité de sa modification effective au sein du système à optimiser sont complètement transparentes, et déléguées à l'implantation du composant configurateur désigné. Le listing C.1 donne un exemple de définition de paramètres d'optimisation. Les propriétés `configurator` indiquent les identifiants des configurateurs.

Enfin, une fois appliquées toutes les nouvelles valeurs des paramètres via les configurateurs, il peut être nécessaire de redémarrer tout ou certaines parties du système sous test afin que ces nouvelles valeurs soient effectivement prises en compte. Une autre motivation pour le redémarrage est le souhait de démarrer avec un système "neuf", dont les performances ne risquent pas d'être perturbées par le test ICAR précédent. Pour cela, l'optimiseur fait la liste des identifiants de lame spécifiés comme démarreurs dans la définition des paramètres (cf. propriétés starter dans le listing C.1). Pour chaque démarreur distinct recensé, l'optimiseur fait appel à la méthode suspend() puis resume() de l'interface `Blade control` (via l'interface `Test control` du superviseur, selon le même principe que pour la méthode `changeParameter()`, cf. listing 6.1) afin d'arrêter et relancer le système sous test ou une partie de ses éléments.

Listing 6.2 – Exemple de définition de 2 paramètres pour l'optimisation de Tomcat au sein du serveur d'application JOnAS. Chaque paramètre est ici désigné par une expression XPATH car le paramétrage repose sur des fichiers XML.

```
parameter.0.name=/Server/Service[@name='JOnAS']/Connector[@executor='tomcatThreadPool']/
    @maxThreads
parameter.0.min=30
parameter.0.max=300
parameter.0.grain=30
parameter.0.configurator=tomcat6-server
parameter.0.starter=jonas

parameter.1.name=/Context/@cacheMaxSize
parameter.1.min=205
parameter.1.max=2050
parameter.1.grain=205
parameter.1.configurator=tomcat6-context
parameter.1.starter=jonas
```

Les identifiants de lame indiqués pour les configurateurs et les démarreurs correspondent aux identifiants donnés dans le plan de test CLIF qui spécifie le déploiement des lames. C'est donc dans le plan de test que sont précisées les différentes implantations de

configurateur et de démarreur spécifiques aux éléments du système à reparamétrer et à redémarrer. Le listing A.1 en donne un exemple.

Listing 6.3 – Exemple de plan de test avec 2 injecteurs, 2 sondes, 2 configurateurs et un démarreur, pour l'optimisation d'une application sur le serveur d'application JOnAS.

```
blade.0.id=injector1
blade.0.injector=IsacRunner
blade.0.server=clif1
blade.0.argument=mystore.xis
blade.0.comment=un injecteur

blade.1.id=injector2
blade.1.injector=IsacRunner
blade.1.server=clif2
blade.1.argument=mystore.xis
blade.1.comment=un autre injecteur

blade.2.id=jvm sut
blade.2.probe=org.ow2.clif.probe.jmx_jvm.Insert
blade.2.server=sut
blade.2.argument=1000 999999 jmx_jvm.properties
blade.2.comment=sonde d'observation de la consommation mémoire JVM du système sous test,
    via JMX

blade.3.id=cpu sut
blade.3.probe=org.ow2.clif.probe.cpu.Insert
blade.3.server=sut
blade.3.argument=1000 999999
blade.3.comment=sonde d'observation de la consommation processeur du système sous test

blade.4.id=tomcat6-server
blade.4.injector=XmlConf
blade.4.server=sut
blade.4.argument=/root/jonas-full-5.2.2/conf/tomcat6-server.xml
blade.4.comment=configurateur de fichier XML, dédié au fichier tomcat6-server.xml

blade.5.id=jonas
blade.5.injector=JonasStarter
blade.5.server=sut
blade.5.argument=/root/jonas-full-5.2.2
blade.5.comment=démarreur de JOnAS

blade.6.id=tomcat6-context
blade.6.injector=XmlConf
blade.6.server=sut
blade.6.argument=/root/jonas-full-5.2.2/conf/tomcat6-context.xml
blade.6.comment=configurateur de fichier XML, dédié au fichier tomcat6-context.xml
```

6.3 Les composants configurateurs

6.3.1 Application de l'architecture commune de lame CLIF

Les configurateurs sont des composants de type lame CLIF. L'implantation d'une lame implique la prise en compte de plusieurs interfaces clientes et serveurs. Toutefois, le travail est grandement simplifié par la disponibilité au sein de CLIF de l'architecture commune CBA (Common Blade Architecture [Dil09]), utilisée par les sondes et les injecteurs, afin de favoriser la réutilisation de code et limiter le travail d'écriture d'une lame à ses éléments strictement spécifiques. En effet, grâce au caractère récursif du modèle Fractal, un composant lame peut lui-même être constitué de plusieurs sous-composants. Comme le montre la figure 6.2 en l'occurrence 4 dans le cas de CBA :

- l'*insert* implante l'activité propre à la lame (e.g. mesure périodique de consommation de ressource pour les sondes, émission de requêtes et production de rapports sur les résultats et temps de réponse pour un injecteur) ;
- l'adaptateur d'insert (`blade-insert-adapter`) prend en charge les éléments génériques de cycle de vie, et facilite l'écriture de l'insert, notamment dans la gestion cohérente de son état et des fils d'exécution ;
- le collecteur de données (`data-collector`) se charge de filtrer toutes les mesures produites par l'insert et de produire des statistiques glissantes sur ces mesures (e.g. moyenne des différentes métriques pour une sonde, débit moyen et temps de réponse moyen pour les injecteurs) ;
- le proxy de stockage (`storage-proxy`) stocke localement les mesures et événements retransmis par le composant `Data collector` (après éventuel filtrage) pendant la durée du test, et les met à disposition pour la collecte finale.

Cette architecture nous permet de limiter notre implantation au sous-composant insert, et de réutiliser, par commodité, l'implantation dédiée aux injecteurs en ce qui concerne les autres sous-composants. En effet, le `data-collector` nous est indifférent, puisque notre insert ne produit aucune mesure, et le `blade-insert-adapter` et le `storage-proxy` sont tout à fait génériques. Le travail d'implantation de l'insert concerne l'interface `Blade insert control`, qui n'est en fait qu'un alias de l'interface `Blade control` déjà présentée. Les méthodes d'intérêt sont :

- `changeParameter(String parameter, Serializable value)`, qui permet de changer la valeur d'un paramètre du système à optimiser,
- `setArgument(String argument)`, qui permet de récupérer une ligne d'arguments d'initialisation du configurateur, comme un nom de fichier XML ou une URI d'agent JMX permettant d'effectuer le changement de valeur d'un ou plusieurs paramètres.

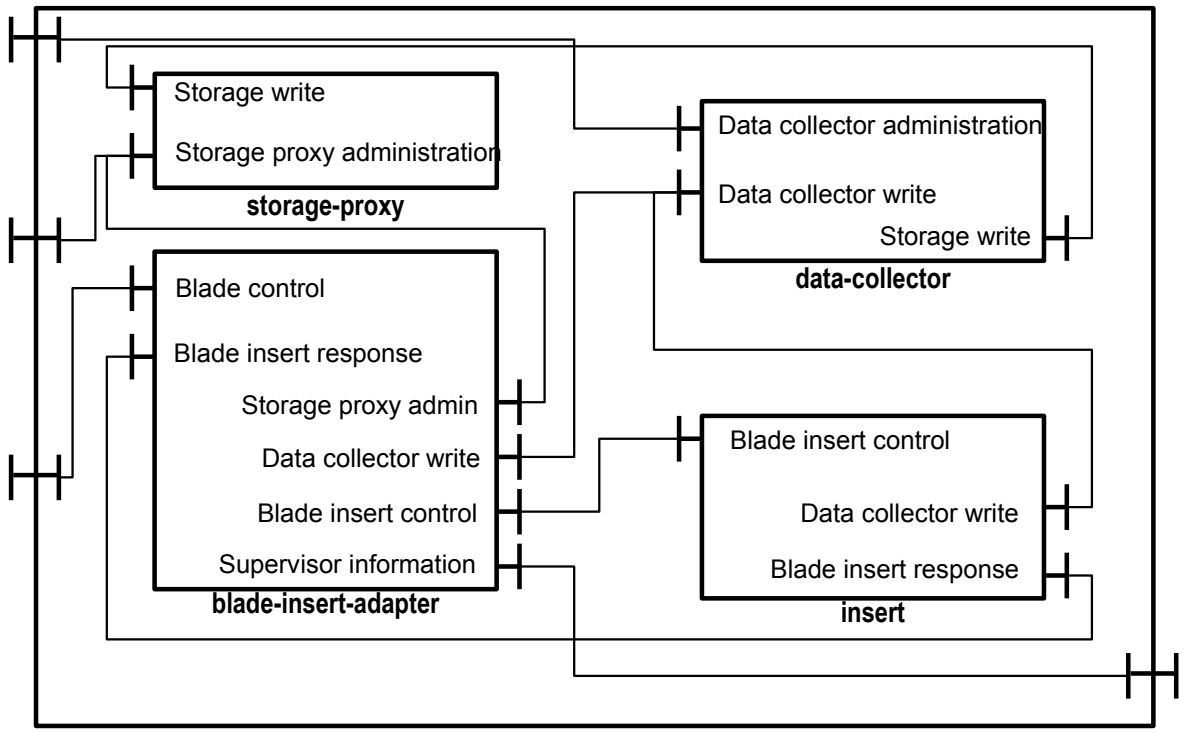

FIGURE 6.2: Architecture commune d'une lame CLIF (Common Blade Architecture)

6.3.2 Exemples de configurateurs

Le composant configurateur est chargé d'appliquer au système sous test un paramétrage donné, tel que calculé par l'optimiseur. Comme nous l'avons indiqué, la raison d'être des configurateurs est de permettre la généricité de l'optimiseur, pour qu'il puisse s'affranchir totalement des modalités de paramétrage. Ces modalités sont spécifiques à chaque système sous test, et l'on pourrait tout à fait envisager le développement d'un ou plusieurs configurateurs spécifiques pour tous les paramètres d'un système donné.

Toutefois, il est préférable de disposer d'un ensemble de configurateurs génériques dédiés à des modalités classiques de paramétrage, telles que fichiers XML, fichiers de propriétés Java ou protocoles d'administration SNMP [IET11] ou JMX [Ora12a]. Afin de limiter le travail d'adaptation de notre plate-forme à tel ou tel système sous test, nous pourrions ainsi nous reposer sur des configurateurs génériques pour chacune de ces modalités.

Dans le cadre de nos travaux, et afin de pouvoir réaliser les expérimentations sur le serveur d'application JOnAS, nous avons eu besoin d'implanter deux configurateurs de fichiers : le premier pour les fichiers de type propriété Java, et le second pour les fichiers de type XML. Ce sont les arguments de déploiement de ces configurateurs génériques,

donnés dans le plan de test CLIF, qui indiquent le fichier de configuration concerné (e.g. voir listing A.1). Ensuite, un nom de propriété Java (respectivement une expression Xpath) permet de désigner la paramètre à modifier.

Le cas des configurateurs de fichiers illustre parfaitement tout le bénéfice tiré de l'exploitation du type de composant lame. En effet, ces configurateurs déclarés dans un plan de test CLIF peuvent être déployés à distance sur le système sous test, ce qui leur donne l'accès indispensable au système de fichier sur lequel se trouve le fichier cible.

6.4 Les composants démarreurs

6.4.1 Principe

A l'instar des composants configurateurs, les composants démarreurs utilisent l'architecture CBA (figure 6.2), ainsi que les mêmes implantations des sous-composants data-collector, storage-proxy et blade-insert-adapter. L'implantation du sous-composant insert est, par contre, spécifique à chaque système ou élément de système à redémarrer.

Comme indiqué dans la description du composant optimiseur, les méthodes `suspend()` et `resume()` du composant démarreur sont appelées, respectivement pour arrêter et redémarrer le (sous-)système concerné par un paramètre donné. Ces méthodes doivent donc être implantées par le composant insert de manière spécifique. De plus, la méthode `setArgument()` permet à l'insert d'obtenir des informations utiles sur le (sous-)système à redémarrer, tels qu'un répertoire d'installation ou une URI d'administration.

6.4.2 Exemple

Nous avons réalisé un démarreur pour le serveur d'application JOnAS, en implantant un composant insert approprié. L'argument de ce démarreur, défini dans le plan de test CLIF (voir l'exemple de la lame numéro 4 dans le listing A.1), donne le chemin vers le répertoire d'installation de JOnAS. La méthode `suspend()` consiste à invoquer le script d'arrêt de JOnAS qui se trouve dans le répertoire d'installation, et à se synchroniser sur l'arrêt effectif. De manière similaire, la méthode `resume()` invoque le script de démarrage et se synchronise sur la disponibilité effective de JOnAS.

Cet exemple montre, comme les configurateurs, le bénéfice de l'exploitation du type lame afin de pouvoir déployer *in situ* les démarreurs liés au système à optimiser, à l'aide d'un plan de test CLIF standard.

6.5 Coordination des boucles de contrôle

6.5.1 Les 3 boucles de contrôle

Notre architecture de self-benchmarking est composée de 3 composants indépendants les uns des autres : le contrôleur de saturation, le contrôleur d'injection de charge et le contrôleur d'optimisation. En effet, aucune liaison Fractal ne vient relier ces composants entre eux, et aucune interface ne semble destinée à synchroniser les activités des 3 composants. Pourtant, comme indiqué dans la figure 5.6 du chapitre 5, ces composants représentent 3 boucles de contrôles qui doivent être coordonnées. Cette absence de synchronisation explicite dans l'architecture est un choix de notre part, guidé par la volonté de produire des boucles de contrôle utilisables indépendamment les unes des autres. Par exemple, la boucle de contrôle de saturation et celle d'injection de charge peuvent être utilisées en dehors du contexte de l'auto-optimisation.

Les 3 boucles de contrôles sont explicitées par la figure 6.3. La boucle 1 consiste à surveiller les mesures remontées par les sondes placées sur le système sous test, et à réagir en signalant la saturation. La boucle 2 (ICAR) observe les métriques de performance mesurées par les injecteurs, et augmente progressivement le nombre d'utilisateurs virtuels des injecteurs en fonction de ces mesures, jusqu'à saturation. La boucle 3 génère et applique de nouveaux paramétrages en fonction du résultat des cycles ICAR. Les 3 contrôleurs correspondant aux 3 boucles communiquent avec les sondes, injecteurs, configurateurs et démarreurs via le superviseur. Nous allons voir comment les boucles se coordonnent.

6.5.2 Problématique

De manière générale, trois options sont possibles pour assurer la coordination de boucles de contrôle :

- Communication directe pair-à-pair : les composants de contrôle ont conscience les uns des autres et communiquent explicitement via des interfaces prédéfinies spécifiques et des liaisons Fractal. Cette approche permet de manifester dans l'architecture un usage particulier des composants, ce qui est bien pour l'usage particulier, mais rend difficile des réutilisations partielles dans d'autres usages (nouveaux composants). En effet, les boucles de contrôle ne sont plus indépendantes, car le contrôle global est intégré dans l'architecture même.
- Contrôle hiérarchique : les boucles de contrôle ne se connaissent pas, mais leur activité est pilotée par un contrôleur de plus haut niveau (super-contrôleur). Par rapport à l'approche précédente, on centralise la logique globale de manière explicite

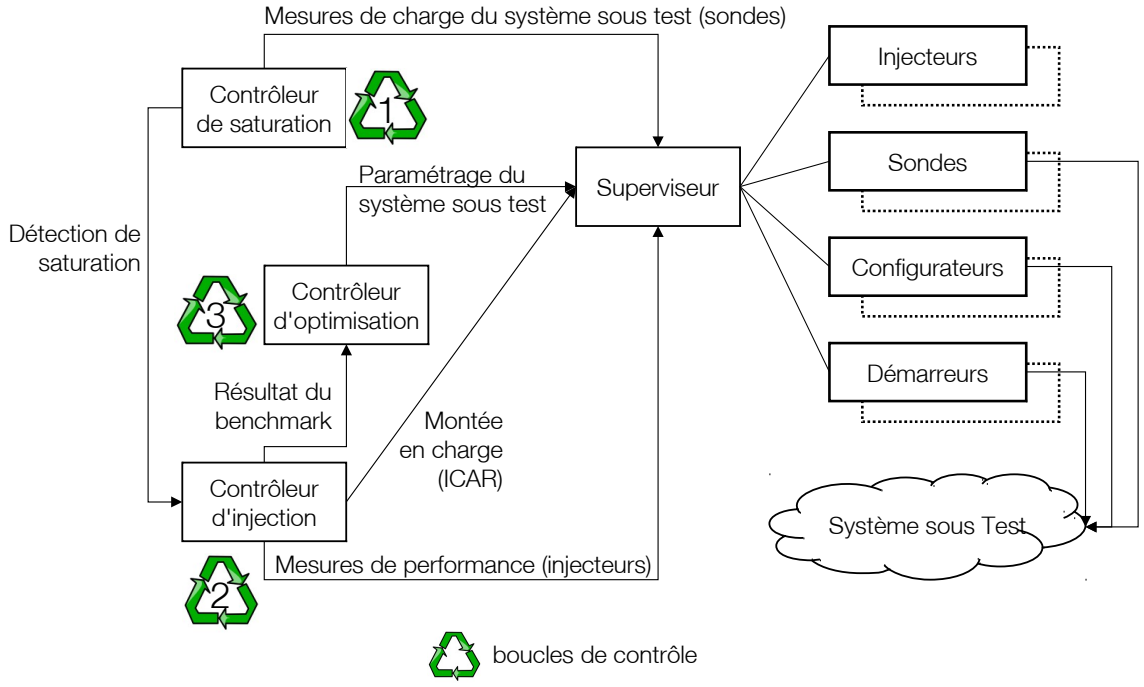

FIGURE 6.3: Vue globale des composants et des boucles de contrôle de l'architecture d'auto-optimisation

dans un seul composant. Les boucles de contrôle n'ont pas conscience les unes des autres mais on a toujours besoin de définir une architecture figée avec des liaisons entre le super-contrôleur et les contrôleurs, reliant des interfaces de synchronisation spécifiques.

– Communication implicite via l'environnement : l'environnement permet aux composants de diffuser ou de recevoir des notifications sans en préciser le destinataire ou, respectivement, sans en connaître l'origine. Concevoir la synchronisation entre les boucles de contrôle selon cette approche favorise la souplesse et la variété de leur utilisation, et ce de manière dynamique, sans avoir besoin de définir et déployer une architecture pour chaque usage. Les boucles sont lancées indépendamment les unes des autres, pas forcément toutes, et éventuellement complétées par d'autres. Bien entendu, la réaction aux notifications reçues et les notifications émises doivent être prévues au départ, à l'instar des interfaces de synchronisation des 2 approches précédentes. La logique globale est complètement répartie entre les composants en présence, et sera donc différente lorsqu'on changera, retirera ou ajoutera un composant.

6.5.3 Approche choisie

Rappelons d'abord notre choix de combinaison des deux processus de test et de configuration présenté dans 6.1, qui repose sur la séparation totale entre ces deux derniers. Par exemple, nous voulons pouvoir utiliser l'injection de charge auto-régulée indépendamment du composant optimiseur pour des usages extérieurs à cette thèse (évaluation d'un système sans processus d'optimisation). Au sein même de l'injection de charge auto-régulée, le contrôleur de saturation doit lui-même pouvoir être utilisé indépendamment du contrôleur d'injection de charge (e.g. pour signaler un dépassement de capacité à un contrôleur d'auto-dimensionnement). Plus généralement, du point de vue de l'autonomic computing, notre motivation est de montrer que des boucles de contrôle peuvent être lancées indépendamment les unes des autres, et observer différents comportements globaux selon les boucles activées (même si dans cette thèse, nous nous restreindrons à l'usage auto-optimisation).

Il nous a paru pertinent de réaliser notre plate-forme selon la troisième approche, et d'utiliser un système de communication asynchrone à base de notifications de type publication-abonnement. En effet, les modèles de communication asynchrones sont plus adaptés que les modèles synchrones client-serveur aux environnements complexes (répartis, ouverts et large échelle) tels que ceux visés par (et justifiant) l'autonomic computing. En effet, ils sont bien adaptés à la conception d'applications réparties sous forme de modules faiblement couplés.

6.5.4 Réalisation

Pour réaliser notre architecture de coordination de boucles indépendantes, nous avons bénéficié d'une nouvelle version du système ICAR, rendue disponible dans le module Selfbench du projet OW2 CLIF. Cette nouvelle version offre une totale séparation des 2 boucles de contrôle (détection de saturation et contrôle d'injection), désormais indépendantes. Pour communiquer avec l'environnement, ces 2 boucles utilisent une communication par messages asynchrones de type publication/souscription sur/à des *sujets* (”topics”), via le standard Java Message Service (JMS [Ora12b]). Le module CLIF/Selfbench est fourni avec l'implantation libre JORAM [OW212], mais une autre intergiciel conforme aux spécifications JMS peut être utilisée.

Comme indiqué par la figure 6.4, le contrôleur de saturation est abonné au sujet ”saturation”. Pour activer la boucle de contrôle, il faut émettre un message de politique de détection de saturation sur ce sujet. Cette politique est un ensemble de propriétés qui définissent :

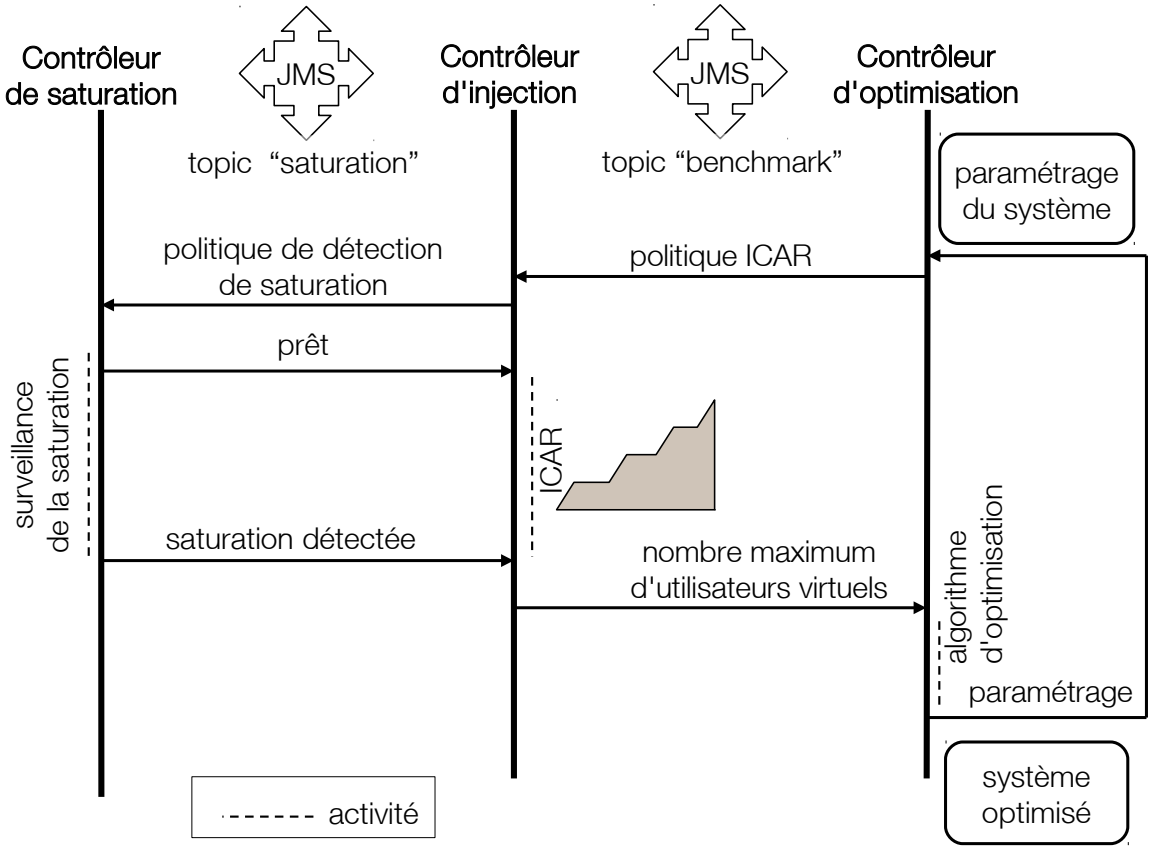

FIGURE 6.4: Vue globale des composants et des boucles de contrôle de l'architecture d'auto-optimisation

- les identifiants des sondes à observer, au sein du plan de test CLIF déployé ;
- les métriques à surveiller et les seuils bas et/ou haut à ne pas dépasser.

A réception du message, la boucle démarre et émet un message sur le même sujet pour prévenir que la surveillance de la saturation est effective. Lorsque les seuils définis par la politique sont franchis, un message est envoyé, toujours sur le même sujet, pour signaler l'atteinte de la saturation. La boucle de contrôle devient alors inactive jusqu'à réception d'une nouvelle politique sur le sujet "saturation". Ainsi, le contrôleur de saturation n'est pas spécifiquement lié à ICAR, et peut être utilisé dans tout contexte où un besoin de surveillance de l'utilisation des ressources est nécessaire.

En ce qui concerne le contrôleur d'injection, il est abonné au sujet JMS "benchmark". Pour activer la boucle de contrôle, il faut émettre un message de politique ICAR sur ce sujet. Cette politique est un ensemble de propriétés qui définissent :

- les identifiants des injecteurs à piloter, au sein du plan de test CLIF ;
- quelques paramètres de configuration de l'algorithme de montée en charge automatique d'ICAR (e.g. nombre de paliers intermédiaires, temps de stabilisation initial) ;

– la politique de détection de saturation qui sera transférée au sujet saturation.

A réception de ce message, le contrôleur transmet la politique de saturation sur le sujet "saturation". Lorsqu'il reçoit le message annonçant l'activation de la surveillance de la saturation, le contrôleur d'injection devient alors actif et démarre le premier palier d'ICAR. Par la suite, la boucle de contrôle ICAR poursuit la montée en charge progressive jusqu'à ce qu'un message sur le sujet "saturation" lui indique la saturation d'au moins l'une des ressources désignées par la politique de saturation. A ce moment-là, l'injection de charge s'arrête, et le contrôleur d'injection émet un message sur le sujet "benchmark" pour annoncer la fin du test et indiquer le nombre maximum d'utilisateurs virtuels atteint avant la saturation.

Il ne reste plus qu'à introduire notre contrôleur d'optimisation. Celui-ci commence par appliquer un paramétrage par défaut au système sous test. Ensuite, il émet un message de politique ICAR sur le sujet "benchmark", puis il attend le message de fin de test qui sera envoyé sur le même sujet, et qui lui donnera la métrique de performance du paramétrage courant. Un nouveau paramétrage est alors calculé par notre algorithme d'optimisation et appliqué par nos composants configurateurs et démarreurs. Un nouveau test ICAR peut alors être demandé, et ainsi de suite jusqu'au déclenchement de la condition d'arrêt de l'algorithme. Le système sous test est alors considéré comme optimisé.

6.6 Conclusion

Ce chapitre montre les aspects architecturaux de notre contribution. D'une part, nous avons montré comment nous avons exploité le canevas logiciel CLIF, sans modification, pour introduire nos composants configurateurs et démarreurs nécessaires au reparamétrage et au redémarrage du système à optimiser. Grâce à l'exploitation du type de composant "lame" et de l'architecture commune de lame CBA, nos travaux s'intègrent ainsi de manière sans couture à l'environnement de test CLIF. En particulier, nous bénéficions du déploiement standard CLIF pour placer nos configurateurs et démarreurs à l'endroit le plus opportun pour accéder au système de fichier de l'élément à paramétrer (cf. fichiers de configuration) ou à redémarrer (cf. scripts ou binaires d'administration).

D'autre part, nous avons mis en œuvre le principe de coordination de boucles de contrôle par messages asynchrones selon un mode de communication publication-souscription. Le principe général est l'activation des boucles de contrôle par envoi d'une politique sur le sujet approprié. Nous avons ainsi construit notre plate-forme d'auto-optimisation en intégrant, de manière faiblement couplée, notre contrôleur d'optimisation au système de test ICAR du module CLIF/Selfbench, lui-même constitué de 2 boucles de contrôle faiblement

couplées.

Afin de valider nos propositions, nous présentons dans le chapitre suivant deux cas d'usage de notre plate-forme d'auto-optimisation, dans le contexte d'applications multi-tiers.

Troisième partie

Validation expérimentale

Chapitre 7

Expérimentations

Nous rendons compte de deux expérimentations montées pour validation de notre contribution. La première concerne application WEB de type achat en ligne simplifiée (`MyStore`) s'exécutant sur le serveur d'application `JOnAS`. La deuxième étudie une application Java EE (`sampleCluster`) dont les trois tiers sont répartis sur des machines distinctes.

Sommaire

7.1 Auto-optimisation d'une application Java EE de type achat en ligne

Il s'agit d'une série d'expériences sur l'application `MyStore` déployée dans le serveur d'applications JOnAS. Nous décrivons l'infrastructure d'exécution et l'outillage mis en place, ainsi que les différentes étapes des expériences. Enfin, nous présentons et analysons les résultats obtenus.

7.1.1 Cas d'étude MyStore sur JOnAS

7.1.1.1 JOnAS

JOnAS 5 est un serveur d'applications d'entreprise certifié Java EE. Il s'agit de l'un des projets majeurs du consortium de logiciel libre OW2.org. Il supporte entre autres les EJBs grâce au conteneur d'EJB `EasyBeans`, qui est également un projet du consortium OW2. JOnAS offre un support Web Java EE complet grâce aux deux conteneurs Web alternatifs, Tomcat et Jetty, qu'il intègre en standard. Par l'intermédiaire de la technologie `JCA` *(Java EE Connector Architecture)*, JOnAS intègre le serveur `JMS`*(Java Message Service)* `JORAM`, toujours du même consortium. Pour les aspects administration, JOnAS implémente la spécification `JMX` *(Java Management eXtensions)*. La console d'administration de JOnAS appelée `JonasAdmin` est une application Web qui permet la consultation et la manipulation des `MBeans` du serveur.

La structure d'une distribution JOnAS comporte plusieurs répertoires :

- Le répertoire `deploy/` est l'endroit principal pour le déploiement des différents composants et applications. Au démarrage de JOnAS, les composants suivant sont déployés dans cet ordre : plans de déploiement des dépôts, *bundles* OSGI, archives RAR, plan de déploiement des ressources, archives EJB, archives WAR et enfin archives EAR. JonasAdmin est déployé sous forme d'un archive WAR.
- Le répertoire `bin/` contient les fichiers binaires et scripts (Windows et Linux) nécessaires au lancement et au contrôle d'exécution de JOnAS.
- Le répertoire `lib/` contient les différentes bibliothèques Java nécessaires.
- Le répertoire `logs/` est utilisé pour la journalisation.
- Le répertoire `repositories/` contient les applications et composants à déployer (bundles OSGI, applications jonasAdmin, documentation, etc.)
- Le répertoire `conf/` est le répertoire qui nous intéresse car il contient l'ensemble de fichiers de configuration du serveur.

JOnAS contient plus de 250 bundles, couvrant les 28 services techniques de JOnAS

(registry, JMX, persistance, Web, sécurité, etc.) possédant au total des centaines de paramètres de configuration.

Un paramètre de configuration est un réglage quantitatif ou qualitatif du fonctionnement du serveur ou de l'un de ses services. Ces réglages ont un impact sur le fonctionnement du serveur, et potentiellement sur ses performances, en fonction de l'application et du trafic de requêtes qu'elle subit. Le choix optimal des valeurs des paramètres pour un cas d'usage précis dans le but d'obtenir les meilleures performances (meilleures au sens d'un cahier des charges et d'exigences fixées) correspond à la pratique du *tuning*, que nous nommerons ici tout simplement *optimisation*.

Les serveurs Java EE permettent de développer et déployer des applications d'entreprises conçues pour fonctionner en architecture constituée de trois tiers (Web, métier, base de données). Le paramétrage de ce type d'infrastructure peut être décomposé en 3 niveaux correspondant aux 3 tiers. L'expérience dans ce domaine démontre qu'une partie importante des problèmes de performances (dégradation, goulet d'étranglement, etc.) est due à un *mauvais réglage* au niveau de ces tiers. Par exemple, les mécanismes de *cache* et de *pooling*, justement destinés à améliorer les performances, peuvent êtres sources de sérieuses dégradations s'ils ne sont pas correctement paramétrés. Nous nous limitons dans nos expérimentations à un certain nombre de paramètres de réglage appartenant à cette catégorie.

7.1.1.2 MyStore

MyStore est une application Web de type achat en ligne développée à titre d'exemple dans le cadre du projet JASMINe du consortium OW2. Il s'agit d'une application simplifiée sans base de données, développée à l'aide de `servlets` Java. Elle offre à l'utilisateur la possibilité d'ajouter et de retirer un certain nombre d'articles dans son *e-panier*. MyStore est téléchargeable depuis le serveur SVN du projet JASMINe sur OW2.org, sous forme d'une archive EAR (*(Entreprise Application aRchive*). Pour la déployer, il suffit de copier cette archive dans le répertoire `deploy/` de la distribution JOnAS ou d'utiliser la console d'administration JonasMyadmin.

7.1.2 Description du problème d'optimisation

Dans nos expérimentations, nous avons choisi comme critère de performance le *nombre d'utilisateurs virtuels actifs simultanés* noté `VUsers`. Cette métrique est tout à fait pertinente du point de vue du fournisseur de service, car l'optimisation consiste alors à maximiser le nombre de clients que l'on peut servir simultanément sans que le serveur ne

paramètre	description	valeur par défaut
maxThreads	taille max. du pool de threads du connecteur HTTP	200
cacheMaxSize	taille max. du cache Web	1024
minSpareThreads	nombre min. de threads inactifs	2
maxpareThreads	nombre max. de threads inactifs	50
acceptCount	nombre de requêtes pouvant être mises en attente	100

TABLE 7.1: Description des paramètres d'optimisation pour l'application MyStore

sature. Dans la phase de dimensionnement, l'administrateur peut ainsi minimiser la quantité de ressources à attribuer au service, et ainsi augmenter le retour sur investissement et coût de fonctionnement (cf. administration et énergie pour l'infrastructure). L'objectif de ces expérimentations est donc trouver la configuration qui maximise le nombre de VUsers. Nous faisons par la suite une comparaison avec les performances obtenues avec une configuration par défaut pour valider nos résultats et mesurer le gain relatif apporté.

Le tableau 7.1 liste les paramètres d'optimisation de l'application MyStore dans JOnAS. Les valeurs par défaut correspondent à celles fixées à l'installation de la distribution originale de JOnAS. Par ailleurs, les règles suivantes sont à prendre en considération lors du choix des domaines des paramètres afin d'éviter une éventuelle incompatibilité entre les valeurs possibles des différents paramètres :

- minSpareThreads < maxThreads (obligatoire) ;
- maxSpareThreads < = maxThreads (recommandé) ;
- acceptCount < = maxThreads (recommandé) ;
- maxConnPool < maxThreads (recommandé).

Pour chaque paramètre retenu, le tableau 7.2 donne l'intervalle et la granularité des valeurs que nous avons choisis. Il en découle une liste de valeurs possibles telles qu'elles pourraient être testées dans un parcours exhaustif, en partant de la borne inférieure et en appliquant itérativement la granularité. Ces valeurs sont données à titre d'information dans la quatrième colonne. En cinquième colonne, le tableau liste les valeurs susceptibles d'être testées par notre algorithme, compte-tenu du découpage dichotomique des intervalles. La combinaison entre les différentes valeurs des différents paramètres donne les tuples susceptibles d'être appliqués et testés. Ces tuples sont calculés par l'algorithme élémentaire (voir Algorithme 2). Seulement certains de ces tuples vont être effectivement appliqués sur le système sous test, et ce, conformément au fonctionnement de l'algorithme élémentaire (voir section 5.3. Nous les avons cités ici pour montrer le gain en nombre d'appels au test réalisé par notre approche.

paramètre	domaine	gran.	valeurs possibles	valeurs par dichotomie
maxThreads	[30,300]	30	30, 60, 90, 120, 150, 180, 210, 240, 270, 300	30, 63, 97, 131, 165, 198, 232, 266, 300
cacheMaxSize	[205,2050]	205	205, 410, 615, 820, 1025, 1230, 1435, 1640, 1845, 2050	205, 435, 666, 896, 1127, 1357, 1588, 1819, 2050
minSpareThreads	[2,10]	2	2, 4, 6, 8, 10	2 , 4, 6, 8, 10
maxSpareThreads	[10,100]	10	10, 20, 30, 40, 50, 60, 70, 80, 90, 100	10, 21, 32, 43, 55, 66, 77, 88, 100
acceptCount	[15, 150]	15	15, 30, 45, 60, 75, 90, 105, 120, 135, 150	15, 31, 48, 65, 82, 99, 116, 133, 150

TABLE 7.2: Calcul par dichotomie des valeurs possibles en prenant en compte la granularité de chaque paramètre

machine	rôle	logiciel et outils	composants déployés
Inj1	contrôle général d'auto-optimisation, injection de charge	JDK, Ant, Selfbench, JORAM, clif-optimizer	contrôleurs d'optimisation, de saturation et d'injection, injecteur de charge
Inj2	injection de charge	JDK, Ant, CLIF	injecteur de charge
SUT	Machine sous test	JDK, Ant, CLIF, JOnAS et MyStore	sondes cpu et jvm, 2 configurateurs XmlConf, démarreur JonasStarter

TABLE 7.3: Tableau récapitulatif de l'outillage nécessaire pour une expérience d'auto-optimisation avec le dispositif Self-Optimizer

7.1.3 Plate-forme de test et d'optimisation

Le tableau 7.3 résume les éléments de l'infrastructure de test que nous avons mis en place pour nos expérimentations. Pour mémoire (cf. chapitre 6), CLIF est un canevas logiciel pour l'injection de charge et le test de performance, et Selfbench est un module complémentaire qui permet de rechercher automatiquement le point de saturation du système sous test en augmentant le nombre de VUsers. Le module clif-optimizer est notre contribution logicielle. Il comporte le composant de contrôle d'optimisation (algorithme d'auto-optimisation) et les composants configurateurs et démarreurs qui permettent de changer le paramétrage du système sous test et son redémarrage.

Le fonctionnement de Selfbench est défini par trois fichiers :

- `selfbench.props` est le fichier de configuration de Selfbench. Il contient un ensemble de paramètres gérant le fonctionnement des deux boucles de contrôle d'injection et de détection de saturation (identification des injecteurs et des sondes, métriques à surveiller et seuils de saturation, paramètres de montée en charge, cf. Annexe B.1).
- `mystore.ctp` est le fichier de définition du plan de test CLIF, i.e. la liste des injecteurs de charge et des sondes à déployer, ainsi que des configurateurs et démarreurs. Pour nos expériences, nous avons besoin de 5 composants de type *Injector* (2 vrais injecteurs de charge sur les machines d'injection, ainsi que 3 pseudo-injecteurs qui sont en fait 2 configurateurs XML et 1 démarreur sur la machine sous test) et de 2 composants de type *Probe* (JVM et CPU) sur la machine sous test (cf. Annexe A.1),
- `mystore.xis` est la définition du scénario d'injection, i.e. la spécification du flux de requêtes à générer. En réalité, seule la partie définition du comportement des VUsers est utile (cf. Annexe A.2), puisque le profile de charge (nombre de VUsers simultanément actifs en fonction du temps) est pris en charge dynamiquement par Selfbench.

La figure 7.1 montre un exemple de l'évolution de la montée en charge durant une campagne de test. C'est une capture d'une application Flash qui affiche en temps réel l'évolution du nombre de VUsers en fonction du temps.

7.1.4 Observations et résultats obtenus

Le listing 7.1 est un extrait de la fin du fichier log de la campagne d'optimisation de MyStore. Comme on peut le voir à la fin du listing, cette campagne a duré 800 minutes et 10 secondes.

La métrique optimale retournée par le dispositif et mesurée sur le système est 687 VUsers.

La configuration correspondante est **[2, 205, 48, 21, 165]** soit :

- minSpareThreads = 2
- cacheMaxSize = 205 (kilos octet)
- acceptCount = 48
- maxSpareThreads = 21
- maxThreads = 165

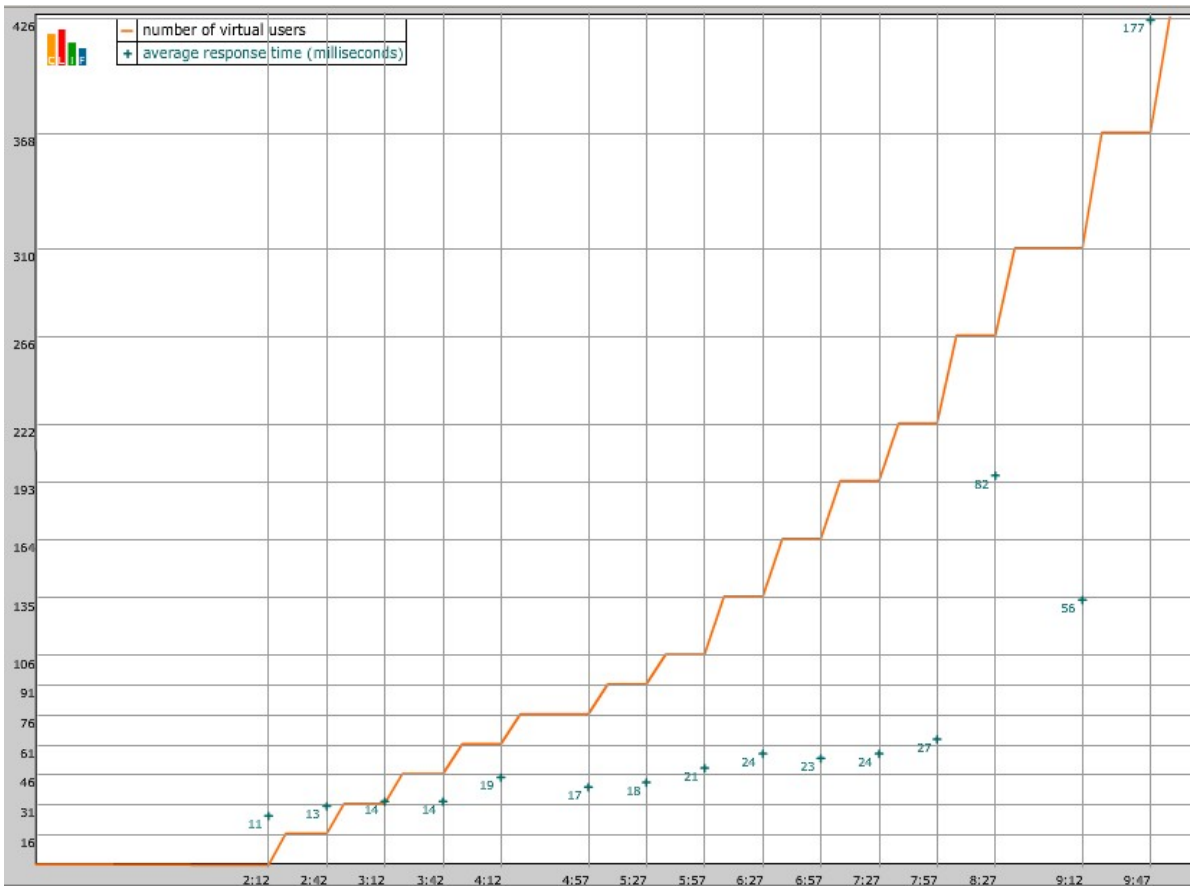

FIGURE 7.1: Exemple de montée en charge ICAR, correspond à un test de MyStore avec une configuration avec des MIN

Listing 7.1 – Extrait de la fin du log de l'optimizer

```
     [java]                                                  - - - - - - - - - - - - - - -
        -
     [java] P_bar the setting parameteres vector becomes: |      [2, 205, 48, 21, 165]
             |
3    [java]                                                  - - - - - - - - - - - - - - -
        -
     [java]  - - - - - - - - - - - - - - -
     [java] |           Go to the following parameter p6         |
     [java]  - - - - - - - - - - - - - - -
     [java]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
8    [java] |           END of All parameters optimal values computing for this pass
              |
     [java]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     [java]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     [java] |           The New Setting Variation varP, after 3pass is: 0.0             |
     [java]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
13   [java]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     [java] |           The New Metric Variation varM, after 3pass is: 0.3391812865497076
                    |
     [java]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     [java]  - - - - - - - - - - - - |   Let's go to a NEW pass     | - - - - - - - - - - - -
     [java]                                          __  __  __  __  __ __
18   [java]                                        |       END          |
     [java]                                        |       OF           |
     [java]                                        |   OPTIMIZATION     |
     [java]                                         __  __  __  __  __ __
     [java]
23   [java] For the following cause(s) :
     [java]
     [java] varM condition was it verified ?0.0<1.0E-4 ?
     [java] varP condition was it verified ?0.3391812865497076<5.0E-4 ?
     [java] Max pass condition was it verified ?4<3 ?
28   [java]                                          __  __  __  __  __ __
     [java]                                        |       FINAL        |
     [java]                                        |      RESULTS       |
     [java]                                         __  __  __  __  __ __
     [java]
33   [java]                                                    - - - - - -
     [java] The optimal metric is:                        |  687   |
     [java]                                                    - - - - - -
     [java] The optimal Settings are: [2, 205, 48, 21, 165]
     [java]
38   [java] - /Server/Service[@name='JOnAS']/Connector[@executor='tomcatThreadPool']/
        @minSpareThreads : 2
     [java] - /Context/@cacheMaxSize : 205
     [java] - /Server/Service[@name='JOnAS']/Connector[@executor='tomcatThreadPool']/
        @acceptCount : 48
     [java] - /Server/Service[@name='JOnAS']/Connector[@executor='tomcatThreadPool']/
        @maxSpareThreads : 21
     [java] - /Server/Service[@name='JOnAS']/Connector[@executor='tomcatThreadPool']/
        @maxThreads : 165
43

BUILD SUCCESSFUL
Total time: 800 minutes 10 seconds
```

7.1.4.1 Performances de la configuration par défaut

Nous avons procédé à la mesure de performances de l'application avec une configuration par défaut (délivrée avec la distribution originale, voir le tableau 7.1). La configuration (matérielle et logicielle) de la plate-forme de test (voir section 7.1.3) reste telle qu'elle est.

La configuration par défaut est **[4, 1024, 100, 50, 200]**, où :

- minSpareThreads = 4
- cacheMaxSize = 1024 (kilo octets)
- acceptCount = 100
- maxSpareThreads = 50
- maxThreads = 200

La métrique correspondante mesurée par *Selfbench* est **399 VUsers**.

Donc, par rapport aux performances réalisées par notre dispositif, on passe de 399 VUsers à 687 VUsers, ce qui représente une amélioration de performance de 72,18%.

7.1.4.2 Performances de la configuration avec des valeurs maximales

De la même manière, la configuration avec les valeurs maximales est **[10, 2050, 150, 100, 300]**, où :

- minSpareThreads = 10
- cacheMaxSize = 2050 (kilo octets)
- acceptCount = 150
- maxSpareThreads = 100
- maxThreads = 300

La métrique correspondante mesurée par *Selfbench* est de 401 VUsers.

Donc, par rapport aux performances réalisées par notre dispositif, on passe de 401 VUsers à 687 VUsers, ce qui représente une amélioration de performance de 71,32%.

7.1.4.3 Performances de la configuration avec des valeurs minimales

La configuration avec des valeurs minimales est d'office calculée par notre algorithme, c'est le premier appel à ICAR (un choix du paramétrage algorithmique).

La configuration est **[2, 205, 15, 10, 30]**, où :

- minSpareThreads = 2
- cacheMaxSize = 205 (kilo octets)

– acceptCount = 15
– maxSpareThreads = 10
– maxThreads = 30

La métrique correspondante est **465 VUsers**.

Donc, par rapport aux performances réalisées par notre dispositif, on passe de 465 VUsers à 687 VUsers, ce qui représente une amélioration de performance 47,74%.

7.1.5 Analyse de résultats

Les résultats obtenus démontrent l'efficacité de notre dispositif d'auto-optimisation, l'amélioration de performances est de l'ordre de 72,18% par rapport aux réglages par défaut. La figure 7.2 présente le gain en performance de l'application MyStore apporté par notre dispositif d'auto-optimisation.

Bien que MyStore soit une application « légère », nos résultats montrent qu'un paramétrage par défaut n'est toujours pas la meilleure solution pour obtenir les meilleures performances. Une pratique régulière de test s'impose donc avant la mise en production d'applications.

Les résultats démontrent aussi un degré important de similitude entre les performances obtenues par le biais d'un paramétrage avec les valeurs maximales des paramètres et celles obtenues avec le réglage par défaut. Ceci illustre une tendance générale chez les éditeurs de logiciels notamment les développeurs de serveurs d'applications avant la délivrance de leurs produits, qui consiste à paramétrer ces dernières aux valeurs quasi maximales, c'est à dire en fixant les paramètres à des valeurs « grandes », tout en veillant à ne pas provoquer un phénomène de sur-dimensionnement de certains paramètres. Un sur-dimensionnement a en effet souvent un effet négatif sur les performances et peut engendrer un dysfonctionnement du système voire une panne complète. Ce type de réglage concerne essentiellement les paramètres des pools et caches des serveurs WEB, serveurs d'application et gestionnaires de bases de données.

7.1.5.1 Gain en temps

Le gain en temps d'exécution est un objectif important de l'automatisation du réglage optimal comme nous l'avons souligné au chapitre 5 en section 5.1.3. Rappelons qu'un paramètre p_i qui prend ses valeurs dans un domaine $d_i = [\mathrm{MIN}_i, \mathrm{MAX}_i]$ et a une granularité g_i peut prendre $[(\mathrm{MAX}_i - \mathrm{MIN}_i)/g_i] + 1$ valeurs. Ainsi, pour nos paramètres on aurait : 10 valeurs possibles pour le paramètre maxThreads, 10 pour cacheMaxSize, 5 pour minSpareThreads, 10 pour maxSpareThreads et 10 valeurs possibles pour le paramètre

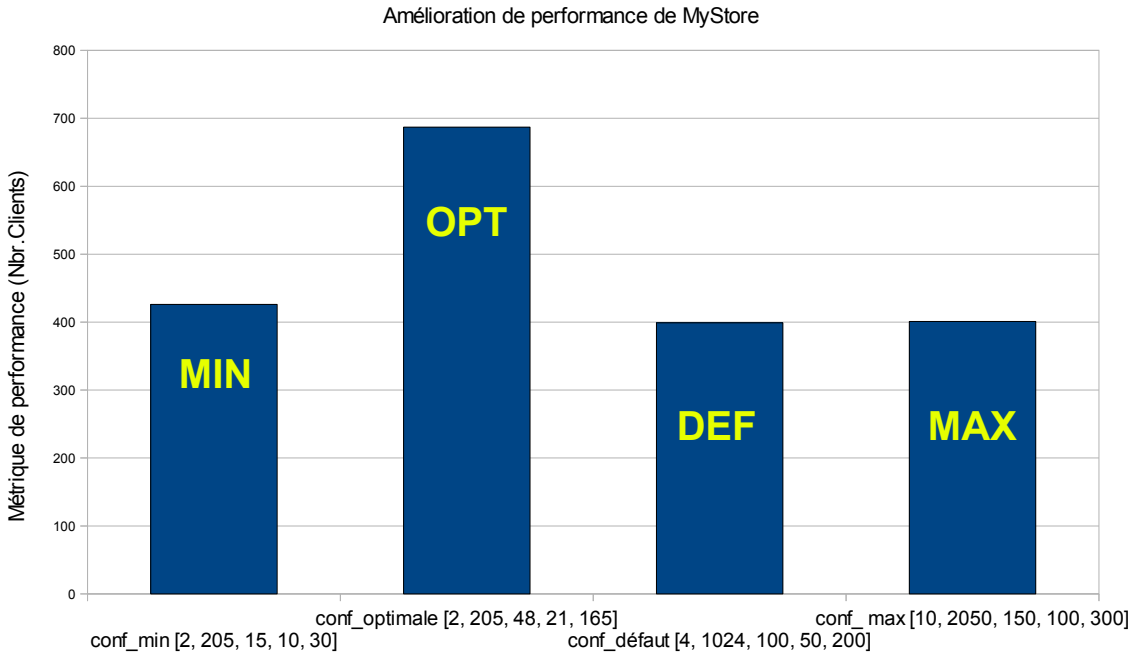

FIGURE 7.2: performance de l'application MyStore portée par JOnAS avec différentes configurations

acceptCount. Le nombre de tuples possibles est donc 50000 configurations.

En sachant qu'en moyenne un test ICAR dure 11 minutes environ, le test exhaustif sur l'ensemble de ces configurations durera $50000.11 = 550000$ minutes $= 91666,66$ heures$= 381,94$jours !

En ce qui concerne une recherche exhaustive par dichotomie en tenant en compte la granularité de chaque paramètre (voir le tableau 7.2) on aura : 9 valeurs pour le paramètre maxThreads, 9 pour cacheMaxSize, 5 pour minSpareThreads, 9 maxSpareThreads et 9 valeurs pour acceptCount.

Le nombre de combinaisons possibles serait alors 32805, ce qui correspond à une durée totale égale à 360855 minutes = 6014,25 heures = 250,59 jours, ce qui représente déjà 52,41% de gain de temps par rapport à un test exhaustif sans dichotomie.

Toutefois, en réalité les 32805 configurations possibles ne sont pas toutes appliquées sur le système. Notre algorithme est justement guidé par la valeur de la métrique retournée et en fonction de cette valeur, il écarte de nombreuses configurations. Le nombre exact de tuples à écarter n'est pas calculable. Pour la présente campagne, 176 appels à ICAR ont été effectuées. Ce qui représente un gain en nombres d'appels à ICAR (et donc en temps) d'environ 186 par rapport à une recherche exhaustive (32805).

7.1.5.2 Gain dû au cache du benchmark

Par ailleurs, le mécanisme du cache que nous avons mis en place pour éviter d'appeler ICAR pour des configurations déjà testées (voir section 5.3.2), nous a permis de n'effectuer que 75 appels à ICAR, ce qui représente encore une diminution de nombre d'appels de 134,66%, ce qui donne un pourcentage identique en gain en temps si l'on considère que la durée d'un test ICAR est constante et que le temps d'arrêt, de configuration et de re-démarrage du serveur est également constant.

7.2 Auto-optimisation d'une application multi-tiers

7.2.1 Cas d'études : l'application clusterSample : Apache2-JOnAS-MySQL SampleCluster

L'application clusterSample est dérivée des exemples fournis dans la distribution JOnAS pour démontrer les architectures *cluster* dans JOnAS. Nous l'avons appropriée à notre contexte de travail pour qu'elle fonctionne avec MySQL toutes versions, ainsi que pour HSQLDB et PostgreSQL. Pour rester générique, c'est une application pure `Java EE 5` qui met en oeuvre un *tier* de présentation et un tier applicatif sous forme d'EJB accédant à un *backend* où les données sont stockées. Le tier de présentation de cette application est implémenté par des *servlets* qui offrent à l'utilisateur les fonctionnalités suivantes :

- Ouverture d'une session http,
- Fermeture de cette session,
- Vérification des caractéristiques de cette session (session ID et présence de cookie).

A partir de sa session http, l'utilisateur peut émettre des requêtes qui mettent en oeuvre le backend persistant. Pour chaque requête, l'application indique le nom des nœuds JOnAS impliqués (si on en a plusieurs) dans l'exécution tant au niveau du conteneur Web en l'occurrence Tomcat ou Jetty, que du tier EJB ainsi que le compteur des appels réalisés dans la session. Le retour d'une requête se présente de la manière suivante :

Listing 7.2 – Extrait du retour d'une requête d'utilisateur dans clusterSample avec une architecture mono-JOnAS

```
SessionServlet Output

getRequestURL http://192.168.143.20:9005/clusterSample/jsp/sessionRsp.jsp for the 3 times
    .
from 10.7.244.181 (user is null) to 192.168.143.20 on port 9005.
```

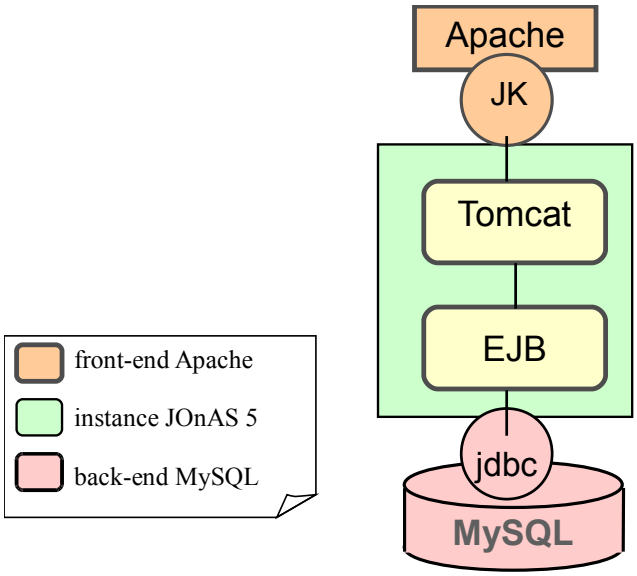

FIGURE 7.3: clusterSample en architecture mono-JOnAS

```
Request path:
Servlet executed on JOnAS instance: jonas_192.168.143.20
Stateless EJB executed on JOnAS instance: jonas_192.168.143.20

Ask again, release session, check session, home.

Session Data: is new Session? false
Session ID: 4443B0FD7FB5C83400CDF3212EC389C9
Creation Time: Mon Sep 03 10:37:08 CEST 2012
Accessed Time: Mon Sep 03 10:37:12 CEST 2012

Statistics:
Owner: 4443B0FD7FB5C83400CDF3212EC389C9
Logging thru org.ow2.jonas.examples.cluster.javaee5.sample.beans.MyStatefulBean@76fa4dab
running on jonas_192.168.143.20
Logging count = 3

servlet running on        EJB created on        total calls
jonas_192.168.143.20        jonas_192.168.143.20        1
jonas_192.168.143.20        jonas_192.168.143.20        2
jonas_192.168.143.20        jonas_192.168.143.20        5


Sample is OK.
```

L'affichage présente les caractéristiques de la session http (ID et de dates ; création et accès courant). Le dernier bloc affiche les informations conservées par un EJB à état *Stateful Session Bean* qui est référencé par la session courante de l'utilisateur (voir la description du tier applicatif ci-après). Le scénario qui a produit l'affichage présenté dans 7.2,

est le suivant :

Un premier utilisateur émet coup sur coup deux requêtes (ce qui correspond aux 2 premières lignes du tableau des statistiques présenté dans la partie inférieure de l'affichage). Un autre utilisateur effectue 2 requêtes (sa session n'est pas représentée ici) Le premier utilisateur réémet une nouvelle requête toujours dans le cadre de sa session initiale : il s'agit donc de son troisième appel qui s'affiche par *for the 3 times* tout en haut de l'affichage. Ce qui est bien en cohérence avec *Logging count = 3* qui est issu de l'état conservé par l'EJB à état. Par ailleurs la dernière case du tableau des statistiques indique le nombre total de requêtes tous utilisateurs confondus qui vaut ici 5 conformément au scénario suivi.

Le tier applicatif de clusterSample met en oeuvre les composants EJB3 suivants :

- Un `SSB` *(Stateless Session Bean)* pour prendre en charge les requêtes http via la méthode process(). Conformément aux patterns classiques Java EE, cet objet est une façade de l'objet détaillé ci-après.
- Un `Entity Bean` qui est un `POJO` grâce à la nouvelle norme EJB3 implémenté grâce au conteneur `EasyBeans` dans JOnAS, cet objet est la représentation des données qui sont persistées par la Base de Données.
- Un `SFSB` *(StateFul Session Bean)* qui est référencé par la session http de l'utilisateur de manière à garder en mémoire les appels (nom des instances JOnAS et compteur du nombre de requêtes).

La figure 7.3 illustre l'architecture que nous avions mis en place dans le cadre de cet expérimentation. Il faut signaler que notre architecture n'implémente pas de mécanisme de load balancing ni de fail over, pour intégrer ces deux mécanisme il aurait fallu juste répliquer les les tiers web et EJB dans une seule (ou plusieurs) instance(s) de JOnAS. Ceci est fait exprès car notre but dans le cadre de ce travail est d'optimiser le réglage d'une instance de JOnAS (et de Tomcat intégré), une fois ceci est fait, ces mécanismes peuvent être intégrés selon les besoins et les exigences architecturale pour renforcer l'amélioration des performances sans aucun problème.

7.2.2 Description du problème d'optimisation

L'objectif de cette expérimentation est trouver la configuration qui maximise le nombre de VUsers. Nous faisons par la suite une comparaison avec les performances obtenues avec une configuration par défaut pour valider nos résultats et mesurer le gain relatif apporté.

Le tableau 7.4 liste l'ensemble de paramètres d'optimisation. Nous avons choisi un paramètre pour chaque tiers (Apache-Tomcat-MySQL). Il faut noter que le paramètre `jdbc.maxconnpool` est un paramètre de JOnAS qui concerne la base de données MySQL,

paramètre	description	valeur par défaut
acceptCount	nombre de requêtes pouvant être mises en attente dans Tomcat	100
jdbc.maxconnpool	taille max. du pool de connexions à MySQL	100
maxClients	nombre max. de connexions parallèles admises dans Apache	200

TABLE 7.4: Description des paramètres d'optimisation pour l'application clusterSample

il est donc configurable à partir de la machine portant JOnAS. En résumé, nous ne configurons pas le SGBD et donc il n y aura pas de configurateur ni de démarreur sur la machine portant MySQL.

Dans le contexte de cette expérience, il faut distinguer entre deux choses, la saturation du SGBD, et l'impact de la reconfiguration du SGBD sur les performances globales.

La nature de l'application clusterSample et l'ensemble d'expériences que nous avions mené nous ont montrés qu'il y a pratiquement très peu de chance que le serveur de BD soit saturé (cpu ou mémoire et même disque) avant les deux autres tiers. Nous avons constaté, pour cette expérience, que la saturation se produit en premier lieu dans JOnAS, et dans quelques situations au niveau d'Apache. Concernant l'impact, la configuration de la BD aurait pu avoir un rôle important s'il y avait un flux considérable de lecture/écriture sur la BD, ce qui n'est pas le cas pour l'application clusterSample, qui est -bien qu'elle soit bel et bien une application avec 3tiers- elle était développée pour tester les aspects de clustering dans JOnAS. J'ajouterai que bien qu'on peut avec notre outil dispositif optimiser un serveur web, un serveur d'application ou un SGBD, le but de la présnete expérience est de tester et d'optimiser les performances d'une application 3tiers, en mettant l'accent sur l'assurance de fluidité de flux entre les 3 tiers et cela en prenant en premier lieu en compte le mécanisme du pooling/cache très important en matière de performance dans de telle architecture, du fait qu'il illustre et résout deux phénomènes fréquents dans le *tuning* de performances de ce type d'applications à savoir le sur/sous dimensionnement de certains paramètres de pool/cache et le phénomène de goulets d'étranglement dans un tier quelconque.

Pour chaque paramètre retenu, le tableau 7.5 donne l'intervalle et la granularité des valeurs que nous avons choisis.

De la même façon que pour la première expérience 7.1.2, le tableau 7.5 contient les informations concernant les valeurs possibles qu'elles pourraient être testées dans un parcours exhaustif et les valeurs susceptibles d'être testées par notre algorithme, compte-tenu du découpage dichotomique des intervalles. Pour le même but, nous les avons cités ici pour

paramètre	**domaine**	**gran.**	**valeurs possibles**	**valeurs par dichotomie**
acceptCount	[100, 300]	25	100, 125, 150, 175, 200, 225, 250, 275, 300	100, 125, 150, 175 ,200, 225, 250, 275 ,300
jdbc.maxconnpool	[100,300]	25	100, 125, 150, 175, 200, 225, 250, 275, 300	100, 125, 150, 175 ,200, 225, 250, 275 ,300
MaxClients	[1000,2000]	100	1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000	1000, 1125, 1250, 1375, 1500, 1625, 1750, 1875, 2000

TABLE 7.5: Calcul par dichotomie des valeurs possibles en prenant en compte la granularité de chaque paramètre

montrer le gain en nombre d'appels au test réalisés par notre approche (voir section 5.3).

7.2.3 Plate-forme de test et d'optimisation

Le tableau 7.6 résume les éléments de l'infrastructure de test que nous avons mis en place pour nos expérimentations. Pour mémoire (cf. chapitre 6), CLIF est un canevas logiciel pour l'injection de charge et le test de performance, et Selfbench est un module complémentaire qui permet de rechercher automatiquement le point de saturation du système sous test en augmentant le nombre de VUsers. Le module clif-optimizer est notre contribution logicielle. Il comporte le composant de contrôle d'optimisation (algorithme d'auto-optimisation) et les composants configurateurs et démarreurs qui permettent de changer le paramétrage du système sous test et son redémarrage.

Le fonctionnement de Selfbench est toujours défini par trois fichiers :

– `selfbench.props` est le fichier de configuration de Selfbench. Il contient un ensemble de paramètres gérant le fonctionnement des deux boucles de contrôle d'injection et de détection de saturation (identification des injecteurs et des sondes, métriques à surveiller et seuils de saturation, paramètres de montée en charge, cf. Annexe B.2). Il faut noter que dans cette expérience l'injection s'arrête dès que l'une de ces règles soit vérifiée (voir à partir de la ligne 34 de B.2) :

  1. Taux maximum d'usage CPU d'un tier égale ou supérieur à 60%
  2. Taux maximum d'usage MEM d'un tier égale ou supérieur à 60%
  3. Temps maximum de réponse à une requête égale ou supérieur à 10s, dépassé ce temps on arrête l'injection et on passe au paramétrage suivant.

machine	rôle	logiciel et outils	composants déployés
Inj1	contrôle général d'auto-optimisation, injection de charge	JDK, Ant, Selfbench, JORAM, clif-optimizer	contrôleurs d'optimisation, de saturation et d'injection, injecteur de charge
Inj2	injection de charge	JDK, Ant, CLIF	injecteur de charge
SUT1	1ère machine sous test	JDK, Ant, CLIF, Apache	sondes cpu, mem et jvm, 1 configurateur PropConf, 1 démarreur ApacheStarter
SUT2	2ème machine sous test	JDK, Ant, CLIF, JOnAS et clusterSample	sondes cpu, mem et jvm, 1 configurateur XmlConf, 1 démarreur JonasStarter
SUT3	3ème machine sous test	JDK, Ant, CLIF, MySQL	sondes cpu, mem, jvm et disk

TABLE 7.6: Tableau récapitulatif de l'outillage nécessaire pour une expérience d'auto-optimisation avec le dispositif Self-Optimizer

Les résultats montrés dans cet exemple ne tenait pas en considération cette dernière règle, d'où le temps relativement grand comparé à celui du premier exemple.

- `cs.ctp` est le fichier de définition du plan de test CLIF, i.e. la liste des injecteurs de charge et des sondes à déployer, ainsi que des configurateurs et démarreurs. (cf. Annexe A.3 ou Tableau 7.6),
- `cs.xis` est la définition du scénario d'injection, i.e. la spécification du flux de requêtes à générer. En réalité, seule la partie définition du comportement des VUsers est utile (cf. Annexe A.4), puisque le profile de charge (nombre de VUsers simultanément actifs en fonction du temps) est pris en charge dynamiquement par Selfbench.

Il reste à souligner qu'une différence existe par rapport à la première expérience est le fait que le système sous test dans la présente expérience est composé de trois éléments à savoir les trois tiers de notre architecture `Apache-JOnAS-MySQL` nécessaire pour le bon focntionnement de l'application `clusterSample`.

7.2.4 Observation et résultats obtenus

Le listing 7.3 est un extrait de la fin du fichier log de la campagne d'optimisation pour clusterSample. Comme on peut le voir à la fin du listing, cette campagne a duré 5024 minutes et 4 secondes.

La métrique optimale retournée par le dispositif et mesurée sur le système est 197

VUsers.

La configuration correspondante est **[225, 100, 1000]** soit :

- acceptCount = 225
- jdbc.maxconpool = 100
- MaxClients = 1000

Listing 7.3 – Extrait de la fin du log de l'optimizer

```
Launch Selfbench for : [225, 100, 1000] [java] 01:45:31
    is ALREADY COMPUTED
     [java] 01:45:31 Selfbench returns the metric: 197
     [java] The optimal value for the parameter:MaxClients is: 1000
     [java]                                                  - - - - - - - - - - - - - - -
       -
     [java] P_bar the setting parameteres vector becomes: |      [225, 100, 1000]
         |
     [java]                                                  - - - - - - - - - - - - - - -
       -
     [java]  - - - - - - - - - - - - - - - -
     [java] |           Go to the following parameter p4         |
     [java]  - - - - - - - - - - - - - - - -
     [java]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     [java] |           END of All parameters optimal values computing for this pass
          |
     [java]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     [java]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     [java] |           The New Setting Variation varP, after 2 pass is: 0.0
         |
     [java]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     [java]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     [java] |           The New Metric Variation varM, after 2 pass is: 0.0            |
     [java]  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     [java]  - - - - - - - - - - - - - |   Let's go to a NEW pass     | - - - - - - - - - - - - -
     [java]                                        __  __  __  __  __ __
     [java]                                       |       END          |
     [java]                                       |       OF           |
     [java]                                       |   OPTIMIZATION     |
     [java]                                        __  __  __  __  __ __
     [java]
     [java] For the following cause(s) :
     [java]
     [java] varM condition was it verified ?0.0 < 1.0E-4 ?
     [java] varP condition was it verified ?0.0 < 5.0E-4 ?
     [java] Max pass condition was it verified ?3 < 2 ?
     [java]                                        __  __  __  __  __ __
     [java]                                       |       FINAL        |
     [java]                                       |       RESULTS      |
     [java]                                        __  __  __  __  __ __
     [java]
     [java]                                                  - - - - - -
     [java] The optimal metric is:                      |  197   |
     [java]                                                  - - - - - -
     [java] The optimal Settings are : [225, 100, 1000]
```

```
     [java]
     [java] Second version of printing :
     [java] - /Server/Service[@name='JOnAS']/Connector[@executor='tomcatThreadPool']/
         @acceptCount :      225
     [java] - jdbc.maxconpool :      100
44   [java] - MaxClients :      1000

BUILD SUCCESSFUL
Total time: 5024 minutes 4 seconds
```

7.2.4.1 Performances de la configuration par défaut

Nous avons procédé à la mesure de performances de l'application avec une configuration par défaut (délivrée avec la distribution originale des trois tiers, voir le tableau 7.4). La configuration (matérielle et logicielle) de la plate-forme de test (voir section 7.2.3) reste telle qu'elle est.

La configuration par défaut est **[100, 10, 200]**, où :

- acceptCount = 100
- jdbc.maxconpool = 10
- MaxClients = 200

La métrique correspondante mesurée par *Selfbench* est **103 VUsers**.

Donc, par rapport aux performances réalisées par notre dispositif, on passe de 103 VUsers à 197 VUsers, ce qui représente une amélioration de performance de 91,26%.

7.2.4.2 Performances de la configuration avec des valeurs maximales

La configuration avec des valeurs maximales choisies est **[300, 300, 2000]**, où :

- acceptCount = 300
- jdbc.maxconpool = 300
- MaxClients = 2000

La métrique correspondante mesurée par *Selfbench* est **92 VUsers**.

Donc, par rapport aux performances réalisées par notre dispositif, on passe de 92 VUsers à 197 VUsers, ce qui représente une amélioration de performance de 141,30%.

7.2.4.3 Performances de la configuration avec des valeurs mainimales

La configuration avec des valeurs mnimales choisies est **[100, 100, 1000]**, où :

- acceptCount = 100
- jdbc.maxconpool = 100
- MaxClients = 1000

La métrique correspondante mesurée par *Selfbench* est **107 VUsers**.

Donc, par rapport aux performances réalisées par notre dispositif, on passe de 107 VUsers à 197 VUsers, ce qui représente une amélioration de performance de 84,11%.

7.2.5 Analyse de résultats

Les résultats obtenus démontrent que l'amélioration de performances est de l'ordre de 91,26% par rapport aux réglages par défaut. La figure 7.4 illustre le gain en performance de l'application clusterSample apporté par notre dispositif d'auto-optimisation.

Ces résultats montrent qu'un paramétrage par défaut n'est toujours pas la meilleure solution pour obtenir les meilleures performances.

Les résultats démontrent aussi un degré important de similitude entre les performances obtenues par le biais d'un paramétrage avec les valeurs maximales ou minimales des paramètres et celles obtenues avec le réglage par défaut. Ceci illustre, bien qu'il ne soit pas une règle, que dans ce type d'applications à 3-tiers, le paramétrage de chaque tiers séparément n'a en pratique aucun sens et que pour avoir des performances meilleures il faut veiller à bien trouver la bonne combinaison de valeurs de paramètres au niveau de chaque tiers. Les résultats montrent que pour des configurations complètement différente, si on prenanit la similitude de chaque paramètre un par un, alors que les performances finales sont proches voire très proches (92, 103,107 VUsers) ce qui confirme la corrélation entre les paramètres dans le sens où on peut obtenir des perfromances similaires avec des configurations différentes.

7.2.5.1 Gain en temps

Rappelons qu'un paramètre p_i qui prend ses valeurs dans un domaine $d_i = [\mathrm{MIN}_i, \mathrm{MAX}_i]$ et a une granularité g_i peut prendre $[(\mathrm{MAX}_i - \mathrm{MIN}_i)/g_i] + 1$ valeurs. Ainsi, pour nos paramètres on aurait : 9 valeurs possibles pour le paramètre acceptCount, 9 pour jdbc.maxconpool et 11 valeurs possibles pour le paramètre MaxClients. Le nombre de tuples possibles est donc 891 configurations.

Pour calculer le gains en temps, il faudra calculer la durée moyenne d'un test ICAR. Dans cette expérience il est : $5024/16 = 314$ minutes

Par conséquent,

Une recherche exhaustive sur l'ensemble des configurations possibles durerait $891.314 = 279774$ minutes Le gain en temps correpsondant est : $5468,75\%$

Une recherche par dichotomie classique aurait pris : $729.314 = 228906$ minutes Ce qui représente un gain en temps de : $4456,25\%$

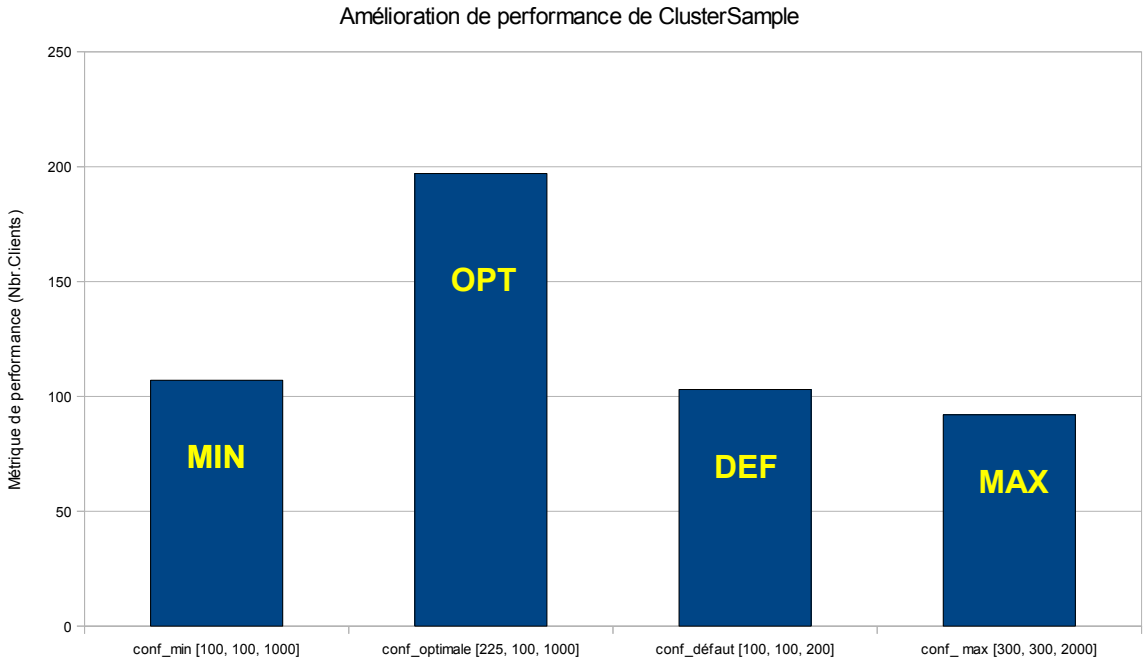

FIGURE 7.4: performance de l'application clusterSample avec différentes configurations

Une recherche avec notre algorithme dichotomique adapté prendrait : $69.314 = 21666$ minutes Ce qui représente un gain en temps de : $1191,30\%$

Et enfin rappelons qu'une recherche avec notre algorithme adapté avec le mécanisme de cache a pris : $16.314 = 5024$ minutes

7.2.5.2 Gain dû au cache du benchmark et reproductivité des résultats

La réduction de nombre d'appels est importante on passe de 891 appels à 16 appels, et par conséquent le temps global est significativement sans commune mesure avec celui nécessaire à une recherche exhaustive.

Quant au gain en temps apporté par le mécanisme de cache il est de $331,25\%$ par rapport à un test sans cache. Il faut rappeler que le cache evite de refaire le test des configurations qui ont déjà été testées dans des passes antérieures.

Concernant la reproductivité des résultats, il est à signaler que ces résultats obtenus supposent que la durée d'un test ICAR est fixe et le temps de configuration d'arrêt et de démarrage des SST sont également fixes. Si on change le paramétrage de selfbench (e.g. ajout d'une règle de saturation en fonction de temps de réponse) c'est la durée de ICAR qui devrait changer et par conséquent la durée globale de la campagne, par contre le nombre d'appels et donc les gains approtés ne devraient pas changés.

7.3 Bilan et conclusions

Les résultats des deux expérimentations prouvent que la méthode que nous avons proposée et mise en oeuvre dans cette thèse donne des résultats significativement positifs. Nous obtenons en effet un réglage qui fournit des performances meilleures que celles que donne un réglage par défaut (de l'ordre de 70% pour la première et de 90% pour la seconde). De plus, ces réglages sont obtenus dans un temps sans commune mesure avec celui nécessaire à une recherche exhaustive du réglage optimal.

La première série d'expériences a visée l'auto-optimisation de `MyStore` une application simple de type achat en ligne, sans base de données. Les cinq paramètres retenus sont liés aux fonctions de pool de fils d'exécution et de cache, dont l'expertise nous enseigne qu'ils peuvent avoir un effet notoire sur les performances. Les bornes et la granularité de valeur choisies pour chaque paramètre engendre 176 combinaisons possibles de paramètres. Une comparaison entre les performances obtenues avec le paramétrage calculé par notre dispositif et celui par défaut montre un gain de performance notoire de 72% en terme de nombre de clients admissibles simultanément par l'application. Notre algorithme a testé 75 combinaisons de valeurs de paramètre en 13h et 20mn.

La deuxième campagne a visée l'auto-optimisation de `clusterSample` une application à trois tiers effectifs (WEB, métier (EJB JOnAS) et base de données). Les trois tiers sont sur des machines physiques distinctes. Ainsi, nous avons trois systèmes sous test à reparamétrer à chaque test et à trouver leur paramétrage optimale pour cette application : Apache2, Tomcat6-JOnAS et enfin MySQL. Nous avons retenu un paramètre par tier liés aux fonctions de *pooling* au niveau de chaque tier. Les bornes et la granularité de valeurs choisies pour chaque paramètre ont donné lieu à 69 combinaisons possibles de paramètres.

Notre algorithme a testé 16 combinaisons de valeurs de paramètre en 83h et 44mn.

Une comparaison entre les performances obtenues avec le paramétrage calculé par notre dispositif et celui par défaut montre un gain de performance notoire de 91% en terme de nombre de clients admissibles simultanément par l'application.

Il faut rappeler que le temps global de l'expérience relativement grand comparé à celui de la première expérience est dû à la complexité de ce type d'architecture N-tiers, d'où l'importance de choisir miticuleusement des paramétrages adéquats visant à améliorer le temps de réponse de l'application, augmenter le nombre de clients et éviter le phénomène de goulets d'étranglement, nous avons constaté pour cette expérience que le temps de réponse de bout en bout avait tendance à s'exploser sous certaines conditions sans pour autant qu'il génère une erreur qui pourrait mettre à fin le test courant (voir critères d'arrêt dans selfbench.props B.1). Cette explosion du temps de réponse est dûe à certaines *mauvaises* combinaisons de paramétrage générées automatiquement ce qui fait que le

système continue à fonctionner mais avec des performances de plus en plus moins bonnes. Suite à ce retour d'expérience, une nouvelle version de selfbench a été mise en place qui permet de définir une nouvelle sonde permettant d'arrêter le test dans le cas où le temps de réponse dépasse un seuil défini (10s par exémple, voir B.2). Des problèmes de performances peuvent se produire aussi à cause des contraintes d'espaces disques, d'où l'importance des sondes `disk` que nous avions mise en place naturellement sur la machine qui porte MySQL mais aussi celle qui héberge JOnAS du fait qu'il y ait beaucoup d'opérations d'arrêt et de redémarrage ce qui risque d'exploser la taille des fichiers logs propore à JOnAS. (voir A.3).

Dans tout les cas, en outre le gain en performances réalisé, les réglages obtenus par notre dispositif d'auto-optimisation sont obtenus dans un temps sans commune mesure avec celui nécessaire à une recherche exhaustive du réglage optimal(voir 7.2.5.1 et 7.2.5.1).

Chapitre 8

Conclusion générale

8.1 Approche

Les travaux décrits dans ce mémoire de thèse concernent l'optimisation automatisée de systèmes informatiques, en lien avec le domaine du test de performance par injection de charge (*benchmarking*). En effet, la pratique du benchmarking engendre un besoin d'optimisation du système à tester afin d'obtenir des résultats quantitatifs signifiants et comparables entre alternatives techniques du système testé. De manière réciproque, le benchmarking est aussi souvent motivé par une recherche d'optimisation d'un système.

Nous nous sommes intéressés à la recherche du paramétrage optimal d'un système informatique en vue de maximiser une métrique de performance. Afin de rendre cette optimisation totalement autonome, nous avons adopté le principe classique d'itération sur des cycles de génération de paramétrages possibles et d'évaluation par benchmark. Nous avons été confrontés, d'une part, à un problème de complexité algorithmique (taille de l'espace des solutions résultant de la combinatoire entre toutes les valeurs possibles de tous les paramètres) et, d'autre part, à la durée de l'évaluation par test en charge de chaque paramétrage. Un parcours exhaustif de l'espace des solutions prendrait un temps totalement rédhibitoire, incompatible avec les cycles de vie des logiciels et avec les coûts acceptables pour les campagnes de benchmark.

Face à cette question critique de délai d'obtention des résultats, nous proposons deux contributions.

Au niveau de la méthode d'optimisation, nous proposons un algorithme original de parcours de l'espace des solutions, de la famille des algorithmes d'ascension de colline, inspiré par la technique de recherche *en croix* des victimes d'avalanche. L'applicabilité de cet algorithme est soumise à une hypothèse sur la forme générale de la fonction qui associe la valeur de la métrique de performance générale aux valeurs des paramètres du

système sous test. Sous réserve du respect de cette hypothèse, notre algorithme trouve le paramétrage optimal avec une complexité temporelle fortement réduite par rapport à la méthode exhaustive, mesurée en nombre d'appels à la fonction d'optimisation à paramètres de réglage fixés.

La seconde contribution est un cadre logiciel qui permet l'automatisation complète de l'optimisation du système, prenant en charge l'application des valeurs des paramètres au système sous test, ainsi que l'évaluation par benchmark automatique. Pour cela, nous poursuivons une approche architecturale à base de composants et de boucles de contrôle autonomes, développée pour la construction de systèmes d'*autonomic computing*.

8.2 Bilan

Nous avons construit une plate-forme d'auto-optimisation en Java, en partant d'un cadre logiciel pour l'injection de charge (CLIF) et d'une extension (Selfbench) qui permet une recherche automatique du nombre maximum de clients admissible par un système sous test. Cette base est conçue selon les principes de l'autonomic computing, avec deux boucles de contrôle hiérarchiques, et selon une approche architecturale à composants issue du projet ANR Selfware. Nous avons complété ces travaux pour y introduire les composants nécessaires au paramétrage et au redémarrage des éléments du système à optimiser (configurateurs, démarreurs), ainsi que le composant de contrôle de l'optimisation, qui implante notre algorithme. De plus, nous avons mis en œuvre une coordination décentralisée (non hiérarchique) entre les trois boucles de contrôle, par le biais d'un mécanisme de communication de type publication-souscription. Par une réutilisation judicieuse du cadre logiciel CLIF, nos nouveaux composants font une réutilisation maximale de CLIF et s'y intègrent parfaitement sans la moindre modification de celui-ci.

Ensuite, nous avons réalisé deux séries d'expériences à l'aide de cette plate-forme, dans le contexte des applications multi-tiers Java EE, et plus particulièrement sur le serveur d'application JOnAS.

La première série d'expériences concerne l'auto-optimisation d'une application simple de type achat en ligne, sans base de données. Nous avons retenu cinq paramètres liés aux fonctions de pool de fils d'exécution et de cache, dont l'expertise nous enseigne qu'ils peuvent avoir un effet notoire sur les performances. Les bornes et la granularité de valeur choisies pour chaque paramètre engendre 176 combinaisons possibles de paramètres. Une comparaison entre les performances obtenues avec le paramétrage calculé par notre dispositif et celui par défaut montre un gain de performance notoire de 72% en terme de nombre de clients admissibles simultanément par l'application. Notre algorithme a testé

75 combinaisons de valeurs de paramètre en 13h et 20mn.

La deuxième campagne a visée l'auto-optimisation de l'application à trois tiers effectifs (WEB, métier (EJB JOnAS) et base de données) clusterSample. Les trois tiers sont sur des machines physiques distinctes. Ainsi, nous avons trois systèmes sous test à reparamétrer à chaque test et à trouver leur paramétrage optimale pour cette application : Apache2, Tomcat6-JOnAS et enfin MySQL. Nous avons retenu un paramètre par tier liés aux fonctions de *pooling* au niveau de chaque tier. Les bornes et la granularité de valeurs choisies pour chaque paramètre ont donné lieu à 69 combinaisons possibles de paramètres. Notre algorithme a testé 16 combinaisons de valeurs de paramètre en 83h et 44mn. Une comparaison entre les performances obtenues avec le paramétrage calculé par notre dispositif et celui par défaut montre un gain de performance notoire de 91% en terme de nombre de clients admissibles simultanément par l'application.

8.3 Perspectives

8.3.1 Extensions de l'outil d'auto-optimisation

Tout d'abord, il s'agit de répondre à la problématique de validité des tuples de paramètres, évoquée section 5.5. L'existence de contraintes entre les paramètres incite à introduire une méthode de type satisfaction de contraintes (*Constraint Satisfaction Problem*, CSP). L'idée est de vérifier la validité de chaque tuple avant d'appliquer et évaluer un paramétrage. Ceci peut être réalisé avec un solveur de contraintes, tel que Choco [CJLR06], dont on utilise la fonctionnalité de vérification de la consistance d'un tuple. Notons que le temps de vérification (de l'ordre de quelques millisecondes) n'a pas d'impact significatif sur l'ensemble de la méthode d'optimisation, le temps d'un test de charge ICAR étant de l'ordre de 10 minutes. Du point de vue de l'algorithme de recherche en croix, une adaptation du parcours est nécessaire car la fonction qui associe la métrique de performance aux valeurs de paramètres est alors définie seulement par morceaux.

A plus long terme, notre canevas logiciel pourrait être étendu afin de pouvoir générer et appliquer non seulement différents paramétrages, mais également différentes configurations en termes de nombre de composants du système testé (par exemple, nombre d'instances des différents tiers d'une application Java EE). L'objectif serait alors de trouver la configuration minimale (afin de réduire les coûts de l'exploitant) permettant d'atteindre un objectif de performance donné (par exemple, capable de servir un nombre donné de clients simultanés, avec une qualité de service et d'expérience donné). Cette perspective étend considérablement la portée de l'auto-optimisation, en intégrant la problématique du

dimensionnement des infrastructures. Elle implique un algorithme global combinant notre algorithme d'optimisation du paramétrage à des règles de réplication des composants. Les outils de description d'architecture par intention et de reconfiguration, par exemple basées sur UML comme le propose [BHS+08], semblent particulièrement adaptés à ce besoin.

8.3.2 Amélioration de l'algorithme d'optimisation

Dans la version actuelle de notre algorithme, nous nous reposons sur une hypothèse assez forte d'unicité des maxima des fonctions projections (voir la section 5.1.2).

Deux approches peuvent être envisagées pour se passer de cette hypothèse.

Dans un premier temps, on peut tenir compte de maxima locaux de la métrique de performance rencontrés au cours de la recherche. Il peut en effet arriver que l'algorithme « rencontre » un maximum local qui ne soit pas la valeur qu'il renvoie. Dans ce cas, on pourrait détecter cette situation, qui montre un viol de nos hypothèses puis *a minima* renvoyer cette valeur au lieu de la valeur donnée par l'algorithme actuel. On peut aussi relancer l'algorithme en partant d'un voisinage du point extrémal trouvé en fixant de nouvelles bornes min et max pour chaque paramètre.

Dans un deuxième temps, on peut envisager, comme dans les méthodes d'ascension de colline, de provoquer une recherche plus ou moins systématique des différents maxima locaux. On quitte alors le domaine des méthodes exactes (voir la section 3.1.2). Parmi les approches possibles pour « s'approcher » des différents maxima, des méthodes de nature stochastique semblent utilisables puisque les contraintes sur les fonctions projections restent vérifiées dans la plupart des cas.

Quatrième partie

Annexes

Annexe A

Plans de test CLIF et scénarios d'injection ISAC

A.1 Plan de test CLIF pour MyStore

Listing A.1 – Extrait de la définition d'un plan de test pour MyStore

```
%#CLIF test plan
%#Thu Jun 16 13:03:53 CET 2012

blade.0.id=injector1
blade.0.injector=IsacRunner
blade.0.injector=IsacRunner
blade.0.server=clif1
blade.0.argument=mystore.xis jobdelay=500 threads=200
blade.0.comment=

blade.1.id=injector2
blade.1.injector=IsacRunner
blade.1.server=clif2
blade.1.argument=mystore.xis jobdelay=500 threads=200
blade.1.comment=

blade.2.id=jvm sut
blade.2.probe=org.ow2.clif.probe.jmx_jvm.Insert
blade.2.server=sut
blade.2.argument=1000 999999 jmx_jvm.properties
blade.2.comment=

blade.3.id=cpu sut
blade.3.probe=org.ow2.clif.probe.cpu.Insert
blade.3.server=sut
blade.3.argument=1000 999999
blade.3.comment=

blade.4.id=tomcat6-server
```

```
blade.4.injector=XmlConf
blade.4.server=sut
blade.4.argument=/home/ebend/jonas/conf/tomcat6-server.xml
blade.4.comment=

blade.5.id=jonas
blade.5.injector=JonasStarter
blade.5.server=sut
blade.5.argument=/home/ebend/jonas
blade.5.comment=

blade.6.id=tomcat6-context
blade.6.injector=XmlConf
blade.6.server=sut
blade.6.argument=/home/ebend/jonas/conf/tomcat6-context.xml
blade.6.comment=
```

A.2 Scénario d'injection ISAC pour MyStore

Listing A.2 – Extrait de la définition d'un scénario d'injection pour MyStore

```
<!DOCTYPE scenario SYSTEM "classpath:org/ow2/clif/scenario/isac/dtd/scenario.dtd"
    >
<scenario>
        <behaviors>
                <plugins>
                        <use id="replayHttp" name="HttpInjector">
                                <params>
                                        <param name="indepthload" value="enabled;
                                            "></param>
                                        <param name="proxyuserpass" value=""></
                                            param>
                                        <param name="useragent" value=""></param>
                                        <param name="proxyport" value=""></param>
                                        <param name="proxyhost" value=""></param>
                                        <param name="proxyusername" value=""></
                                            param>
                                        <param name="preemptiveauthentication"
                                                value="">
                                        </param>
                                        <param name="unit" value="microsecond"></
                                            param>
                                </params>
                        </use>
                        <use id="timer_1s" name="ConstantTimer">
                                <params>
                                        <param name="duration_arg" value="1000"><
                                            /param>
                                </params>
                        </use>
                        <use id="alea" name="Random">
                                <params>
                                        <param name="id" value=''></param>
                                </params>
                        </use>
                        <use id="const" name="Context">
                                <params>
                                        <param name="variables" value=''>
                                        </param>
                                        <param name="files" value='selfbench.
                                            props;'></param>
                                        <param name="id" value=''></param>
                                </params>
                        </use>
                        <use id="common" name="Common">
                                <params>
                                        <param name="id" value=''></param>
                                </params>
                        </use>
                </plugins>
```

```
                <behavior id="session">
45                      <control use="alea" name="setUniform">
                                <params>
                                        <param name="min" value="0"></param>
                                        <param name="max"
                                                value="${const:injection.rampup.
                                                    time}">
50                                      </param>
                                        <param name="id" value=""></param>
                                </params>
                        </control>
                        <timer use="alea" name="sleep">
55                              <params>
                                        <param name="id" value=''></param>
                                </params>
                        </timer>
                        <while>
60                              <condition use="common" name="true"></condition>
                                <timer use="timer_1s" name="period_begin">
                                        <params>
                                                <param name="id" value=""></param
                                                    >
                                        </params>
65                              </timer>
                                <sample use="replayHttp" name="get">
                                        <params>
                                                <param name="password" value=""><
                                                    /param>
                                                <param name="parameters"
70                                                      value="page=cart">
                                                </param>
                                                <param name="realm" value=""></
                                                    param>
                                                <param name="proxyport" value="">
                                                    </param>
                                                <param name="proxyhost" value="">
                                                    </param>
75                                              <param name="cookies" value=""></
                                                    param>
                                                <param name="proxypassword" value
                                                    =""></param>
                                                <param name="hostauth" value=""><
                                                    /param>
                                                <param name="cookiepolicy" value=
                                                    ""></param>
                                                <param name="uri"
80                                                      value="http://${
                                                            const:host}:9000/
                                                            MyStore/">
                                                </param>
                                                <param name="username" value=""><
                                                    /param>
                                                <param name="redirect" value=""><
                                                    /param>
```

```
                                                <param name="localaddress" value=
                                                    ""></param>
                                                <param name="proxylogin" value=""
                                                    ></param>
                                        </params>
                                </sample>
                                <timer use="timer 1s" name="period_end">
                                        <params>
                                                <param name="period_arg" value=""
                                                    ></param>
                                                <param name="id" value=''></param
                                                    >
                                        </params>
                                </timer>
                        </while>
                </behavior>
        </behaviors>
        <loadprofile>
                <group behavior="session" forceStop="true">
                        <ramp style="line">
                                <points>
                                        <point x="0" y="1"></point>
                                        <point x="999999" y="1"></point>
                                </points>
                        </ramp>
                </group>
        </loadprofile>
</scenario>
```

A.3 Plan de test CLIF pour SampleCluster

Listing A.3 – Extrait de la définition d'un plan de test pour SampleCluster

```
#CLIF test plan
#Thu Sep 20 11:29:21 CEST 2012

blade.0.id=SUT MEM3
blade.0.probe=org.ow2.clif.probe.memory.Insert
blade.0.server=sutMysql
blade.0.argument=1000 999999
blade.0.comment=

blade.1.id=SUT MEM1
blade.1.probe=org.ow2.clif.probe.memory.Insert
blade.1.server=sutApache
blade.1.argument=1000 999999
blade.1.comment=

blade.2.id=SUT MEM2
blade.2.probe=org.ow2.clif.probe.memory.Insert
blade.2.server=sutJonas
blade.2.argument=1000 999999
blade.2.comment=

blade.3.id=ejbConfigurator
blade.3.injector=PropConf
blade.3.server=sutJonas
blade.3.argument=/home/ebend/jonas/conf/jonas.properties
blade.3.comment=

blade.4.id=INJ1 MEM
blade.4.probe=org.ow2.clif.probe.memory.Insert
blade.4.server=clif1
blade.4.argument=1000 999999
blade.4.comment=

blade.5.id=injector1
blade.5.injector=IsacRunner
blade.5.server=clif1
blade.5.argument=cs.xis jobdelay=500 threads=200
blade.5.comment=

blade.6.id=injector2
blade.6.injector=IsacRunner
blade.6.server=clif2
blade.6.argument=cs.xis jobdelay=500 threads=200
blade.6.comment=

blade.7.id=mysqlConfigurator
blade.7.injector=PropConf
blade.7.server=sutJonas
blade.7.argument=/home/ebend/jonas/conf/MySQL.properties
```

```
blade.7.comment=

blade.8.id=tomcatConfigurator
blade.8.injector=XmlConf
blade.8.server=sutJonas
blade.8.argument=/home/ebend/jonas/conf/tomcat6-server.xml
blade.8.comment=

blade.9.id=SUT DSK1
blade.9.probe=org.ow2.clif.probe.disk.Insert
blade.9.server=sutApache
blade.9.argument=1000 999999 sda
blade.9.comment=sonde disque Apache

blade.10.id=SUT DSK2
blade.10.probe=org.ow2.clif.probe.disk.Insert
blade.10.server=sutJonas
blade.10.argument=1000 999999 sda
blade.10.comment=sonde disque JOnAS

blade.11.id=INJ1 CPU
blade.11.probe=org.ow2.clif.probe.cpu.Insert
blade.11.server=clif1
blade.11.argument=1000 999999
blade.11.comment=

blade.12.id=SUT DSK3
blade.12.probe=org.ow2.clif.probe.disk.Insert
blade.12.server=sutMysql
blade.12.argument=1000 999999 sda
blade.12.comment=sonde disque MySQL

blade.13.id=2jonasStarter
blade.13.injector=JonasStarter
blade.13.server=sutJonas
blade.13.argument=/home/ebend/jonas
blade.13.comment=

blade.14.id=INJ2 MEM
blade.14.probe=org.ow2.clif.probe.memory.Insert
blade.14.server=clif2
blade.14.argument=1000 999999
blade.14.comment=

blade.15.id=SUT CPU1
blade.15.probe=org.ow2.clif.probe.cpu.Insert
blade.15.server=sutApache
blade.15.argument=1000 999999
blade.15.comment=

blade.16.id=1apacheStarter
blade.16.injector=ApacheStarter
blade.16.server=sutApache
blade.16.argument=/usr/sbin
blade.16.comment=
```

```
blade.17.id=SUT NET3
blade.17.probe=org.ow2.clif.probe.network.Insert
blade.17.server=sutMysql
blade.17.argument=1000 999999 eth0
blade.17.comment=sonde resau MySQL

blade.18.id=3mysqlStarter
blade.18.injector=MysqlStarter
blade.18.server=sutMysql
blade.18.argument=/etc/init.d
blade.18.comment=

blade.19.id=SUT NET1
blade.19.probe=org.ow2.clif.probe.network.Insert
blade.19.server=sutApache
blade.19.argument=1000 999999 eth0
blade.19.comment=sonde reseau Apache

blade.20.id=SUT NET2
blade.20.probe=org.ow2.clif.probe.network.Insert
blade.20.server=sutJonas
blade.20.argument=1000 999999 eth0
blade.20.comment=sonde reseau JOnAS

blade.21.id=apacheConfigurator
blade.21.injector=PropConf
blade.21.server=sutApache
blade.21.argument=/etc/apache2/apacheWorkerMPM.conf
blade.21.comment=

blade.22.id=INJ2 CPU
blade.22.probe=org.ow2.clif.probe.cpu.Insert
blade.22.server=clif2
blade.22.argument=1000 999999
blade.22.comment=

blade.23.id=SUT CPU3
blade.23.probe=org.ow2.clif.probe.cpu.Insert
blade.23.server=sutMysql
blade.23.argument=1000 999999
blade.23.comment=

blade.24.id=SUT CPU2
blade.24.probe=org.ow2.clif.probe.cpu.Insert
blade.24.server=sutJonas
blade.24.argument=1000 999999
blade.24.comment=
```

A.4 Scénario d'injection ISAC pour SampleCluster

Listing A.4 – Extrait de la définition d'un scénario d'injection pour SampleCluster

```
version="1.0"?> <!DOCTYPE scenario SYSTEM "classpath:org/ow2/clif/scenario/isac/dtd/
    scenario.dtd">
<scenario>
        <behaviors>
                <plugins>
                        <use id="arrivee_expo" name="Random"></use>
                        <use id="HttpInjector_1.0_0" name="HttpInjector">
                                <params>
                                        <param name="proxyhost" value=""></param>
                                        <param name="proxyusername" value=""></
                                            param>
                                        <param name="useragent" value=""></param>
                                        <param name="proxyport" value=""></param>
                                        <param name="proxyuserpass" value=""></
                                            param>
                                        <param name="preemptiveauthentication"
                                             value="">
                                        </param>
                                        <param name="indepthload" value="enabled;
                                            "></param>
                                <param name="unit" value="microsecond"></param></
                                    params>
                        </use>

                        <use id="attente_reelle" name="ConstantTimer">
                                <params>
                                        <param name="duration_arg" value=""></
                                            param>
                                </params>
                        </use>

                        <use id="Common_0" name="Common">
                                <params>
                                        <param name="id" value=''></param>
                                </params>
                        </use>
                <use id="const" name="Context"><params><param name="files" value=
                    ""></param><param name="variables" value="host
                    =192.168.143.135;"></param></params></use></plugins>
                <behavior id="cs">
                        <control use="arrivee_expo" name="setNegativeExpo">
                                <params>
                                        <param name="min" value="0"></param>
                                        <param name="mean" value="1000"></param>
                                        <param name="id" value=""></param>
                                </params>
                        </control>


                        <timer use="arrivee_expo" name="sleep">
```

```
                <params>
                    <param name="id" value=""></param>
                </params>
            </timer>
            <while>
                <condition use="Common_0" name="true"></condition>
                <timer use="attente_reelle" name="period_begin"></timer>

                <sample use="HttpInjector_1.0_0" name="get">
                    <params>
                        <param name="uri"
                            value="http://${const:host}:81/clusterSample/servlet/session">
                        </param>
                        <param name="redirect" value=""></param>
                        <param name="headers" value=""></param>
                        <param name="parameters" value=""></param>
                        <param name="response-body" value=""></param>
                        <param name="username" value=""></param>
                        <param name="password" value=""></param>
                        <param name="hostauth" value=""></param>
                        <param name="realm" value=""></param>
                        <param name="proxylogin" value=""></param>
                        <param name="proxypassword" value=""></param>
                        <param name="cookies" value=""></param>
                        <param name="cookiepolicy"
                            value="compatibility">
                        </param>
                        <param name="proxyhost" value="">
                        </param>
                        <param name="proxyport" value="">
                        </param>
                        <param name="localaddress" value="">
                        </param>
                        <param name="id" value=""></param>
                    </params>
                </sample>
                <timer use="attente_reelle" name="period_end">
                    <params>
```

```
                                                    <param name="period_arg"
                                                           value="${arrivee_expo:int
                                                               }">
                                                    </param>
                                                </params>
82                                          </timer>
                                        </while>

                                </behavior>
                        </behaviors>
87                      <loadprofile>
                        <group behavior="cs" forceStop="true"><ramp style="line"><points><point x
                            ="0" y="1"></point><point x="999999" y="1"></point></points></ramp></
                            group></loadprofile>
</scenario>
```

Annexe B

Configuration de l'injection de charge auto-régulée ICAR

B.1 Fichier de propriétés de Selfbench pour MyStore

Listing B.1 – Extrait de la configuration de Selfbench pour MyStore

```
1
# just for mystore.xis scenario:
target host for HTTP traffic host=192.168.143.20

############################
# EVALUATION CONTROL LOOP #
############################

# ramp-up time for each step (ms)
injection.rampup.time=10000
11
# scenario's behavior name
injection.behavior=session

# regular expression (see Java regex) for identifying load injectors
16 injection.ids=injector.*

# maximum acceptable number of errors for all load injectors
injection.maxerrors=0

21 # number of steps to reach the currently assumed Cmax
injection.rampup.steps=6

# initial stabilization time for the first step (s)
injection.stabilization.time=15
26
# minimal number of measures for any step
injection.minimal=80
```

```
# polling period for getting the injectors' statistics (s)
injection.period=5

##############################
# SATURATION CONTROL LOOP #
##############################

# the SaturationException object published on the JMS topic will hold this key
saturation.key=optimizer

# time frame for saturation evaluation (s)
saturation.period=5

# rule 1 on maximum CPU load
saturation.1.ids=cpu sut
saturation.1.index=0
saturation.1.max=80

# rule 2 on minimum JVM free heap size in MB
saturation.2.ids=jvm sut
saturation.2.index=0
saturation.2.min=2
```

B.2 Fichier de propriétés de Selfbench pour SampleCluster

Listing B.2 – Extrait de la configuration de Selfbench pour ClusterSample

```
# just for cs.xis scenario: target host for HTTP traffic
host=192.168.143.135

############################
# EVALUATION CONTROL LOOP #
############################

# ramp-up time for each step (ms)
injection.rampup.time=10000

# scenario's behavior name
injection.behavior=cs

# regular expression (see Java regex) for identifying load injectors
injection.ids=injector.*

# maximum acceptable number of errors for all load injectors
injection.maxerrors=0

# number of steps to reach the currently assumed Cmax
injection.rampup.steps=10

# initial stabilization time for the first step (s)
injection.stabilization.time=15

# minimal number of measures for any step
injection.minimal=100

# polling period for getting the injectors' statistics (s)
#injection.period=5

############################
# SATURATION CONTROL LOOP #
############################

# the SaturationException object published on the JMS topic will hold this key
saturation.key=optimizer

# time frame for saturation evaluation (s)
#saturation.period=5

# rule 1 on maximum CPU load
saturation.1.ids=SUT CPU.
saturation.1.index=0
saturation.1.max=60

saturation.2.ids=SUT MEM.
saturation.2.index=0
```

```
49 saturation.2.max=60

   saturation.3.ids=injector.
   saturation.3.index=1
   saturation.3.max=10000000
```

Annexe C

Problème d'optimisation

C.1 Fichier de propriétés du problème d'optimisation pour MyStore

Listing C.1 – Extrait de la définition du problème d'optimisation pour MyStore

```
# ne pas oublier de decommenter cet element XML dans tomcat6-server.xml
parameter.0.name=/Server/Service[@name='JOnAS']/Connector[@executor='tomcatThreadPool']/@maxThreads
parameter.0.min=30
parameter.0.max=300
parameter.0.grain=30
parameter.0.configurator=tomcat6-server
parameter.0.starter=jonas

parameter.1.name=/Context/@cacheMaxSize
parameter.1.min=205
parameter.1.max=2050
parameter.1.grain=205
parameter.1.configurator=tomcat6-context
parameter.1.starter=jonas

parameter.2.name=/Server/Service[@name='JOnAS']/Connector[@executor='tomcatThreadPool']/@minSpareThreads
parameter.2.min=2
parameter.2.max=10
parameter.2.grain=2
parameter.2.configurator=tomcat6-server
parameter.2.starter=jonas

#parameter.3.name=/Server/Service[@name='JOnAS']/Connector[@executor='tomcatThreadPool']/@maxSpareThreads
#parameter.3.min=10
#parameter.3.max=100
#parameter.3.grain=10
#parameter.3.configurator=tomcat6-server
```

```
#parameter.3.starter=jonas

parameter.3.name=/Server/Service[@name='JOnAS']/Connector[@executor='tomcatThreadPool']/
    @acceptCount
parameter.3.min=15
parameter.3.max=150
parameter.3.grain=15
parameter.3.configurator=tomcat6-server
parameter.3.starter=jonas

#parameter.5.name=maxConnPool
#parameter.5.min=11
#parameter.5.max=180
#parameter.5.grain=15
#parameter.5.configurator=jmx
#parameter.5.starter=jonas
```

C.2 Fichier de propriétés du problème d'optimisation pour SampleCluster

Listing C.2 – Extrait de la définition du problème d'optimisation pour ClusterSample

```
# modifie le 09 sept 2012

passes=3

parameter.0.name=jdbc.maxconpool
parameter.0.min=100
parameter.0.max=300
parameter.0.grain=25
parameter.0.configurator=mysqlConfigurator
parameter.0.starter=2jonasStarter

parameter.1.name=/Server/Service[@name='JOnAS']/Connector[@executor='tomcatThreadPool']/
    @acceptCount
parameter.1.min=100
parameter.1.max=300
parameter.1.grain=25
parameter.1.configurator=tomcatConfigurator
parameter.1.starter=2jonasStarter

parameter.2.name=MaxClients
parameter.2.min=100
parameter.2.max=300
parameter.2.grain=25
parameter.2.configurator=apacheConfigurator
parameter.2.starter=1apacheStarter
```

Bibliographie

[ABC+95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets*. John Wiley & Sons, 1995.

[AL03] E. Aarts and J.K. Lenstra. *Local search in combinatorial optimization*. Princeton University Press, New York, NY, USA, 2nd edition, 2003.

[Bar04] S. Barber. Beyond performance testing, part 1-14. `www-128.ibm.com/developerworks/rational/library/4169.html`, 2004. (retrieved 2012-06-30).

[Bay00] B. Baynat. *Théorie des files d'attente : des chaînes de Markov aux réseaux à forme produit*. HERMES Science Europe, 2000.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(1-2) :11–12, 2006.

[BCS04] E. Bruneton, Th. Coupaye, and J.-B. Stefani. The fractal component model. http ://fractal.ow2.org/specification, 2004. (retrieved on 2012-06-01).

[Ber] Berkeley/Stanford Universities. The recovery-oriented computing (ROC) project. `http://roc.cs.berkeley.edu`. (retrieved on 2012-03-27).

[BHS+08] L. Broto, D. Hagimont, P. Stolf, N. De Palma, and S. Temate. Autonomic management policy specification in tune. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 1658–1663, New York, NY, USA, 2008. ACM.

[BPHT06] S. Bouchenak, N. De Palma, D. Hagimont, and Ch. Taton. Autonomic management of clustered applications. In *IEEE International Conference on Cluster Computing (Cluster 2006)*, Barcelona, Spain, September 2006. IEEE Comp. Soc. Press.

[Bri] HP Bristol. Jsr 318 : Enterprise javabeans.

[BSP+02] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills, and Y. Diao. Able : a toolkit for building multiagent autonomic systems. *IBM Syst. J.*, 41(3) :350–371, July 2002.

[BSW97] J. Bosch, C. A. Szyperski, and W. Weck. 2nd workshop on component-oriented programming (wcop'97). In *ECOOP Workshops*, pages 323–326, 1997.

[Cea68] J. Cea. Les méthodes de descente dans la théorie de l'optimisation. *Revue française d'informatique et de recherche opérationnelle*, 2(3) :79–101, 1968.

[CJLR06] H. Cambazard, N. Jussien, F. Laburthe, and G. Rochart. The choco constraint solver. In *INFORMS Annual meeting, Pittsburgh, PA, USA*, Pittsburgh, PA, USA, November 2006.

[CKZ+03] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. Jagr : An autonomous self-recovering application server. In *5th Annual Int. Workshop on Active Middleware Services (AMS 2003)*, pages 168–178, Seattle, WA, USA, 2003. IEEE Computer Society.

[Coi96] P. Cointe, editor. *European Conf. on Object-Oriented Programming (ECOOP'96)*, number 1098 in LLNCS. Springer-Verlag, 1996.

[Col] Imperial College. The darwin project.

[Cou] Transaction Processing Performance Council. Tpc2. `http://www.tpc.org/information/about/history.asp`. (retrieved on 2011-03-18).

[CS00] R. Chelouah and P. Siarry. Tabu search applied to global optimization. *European Journal of Operational Research*, 123 :256–270, 2000.

[DBS06] M. Dorigo, M. Birattari, and Th. Stutzle. Ant colony optimization : Artificial ants as a computational intelligence technique. *IEEE Computational Intelligence Magazine*, pages 28–39, 2006.

[Dil09] B. Dillenseger. Clif, a framework based on fractal for flexible, distributed load testing. *Annales des Télécommunications*, 64(1-2) :101–120, 2009.

[DMG11] A. Duarte, R. Marti, and F. Gortazar. Path relinking for large-scale global optimization. *Soft Comput.*, 15 :2257–2273, 2011.

[FF63] E.A. Feignebaum and J. Feldman. *Computer and Thought*. McGraw Hill, New-York, NY, USA, 1963.

[FGK02] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL : A Modeling Language for Mathematical Programming*. Brooks/Cole Publishing Company, 2002.

[FIC] FICO. Fico xpress optimization suite. `www.fico.com/en/Products/.../FICO-Xpress-Optimization-Suite.aspx`. (retrieved on 2012-02-08).

[FSLM02] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. Think : A software framework for component-based operating system kernels. In *Proceedings of the USENIX Annual Technical Conference*, pages 73–86, Monterey, CA, USA, 2002.

[GGM03] J.-M. Geib, Ch. Gransart, and Ph. Merle. Corba : des concepts à la pratique. `http://rangiroa.essi.fr/cours/car/00-poly-corba.pdf`, 2003. (retrieved on 2011-10-05).

[GHBP09] A. Gadafi, D. Hagimont, L. Broto, and J.-M. Pierson. Autonomic energy management of multi-tier clustered applications. In *10th IEEE/ACM International Conference on Grid Computing*, pages 201–208, Banff, Alberta, Canada, October 13-15 2009.

[Gra85] J. Gray. A measure of transaction processing power. *Microsoft Research*, 1985.

[Hai06] S. Haines. *Pro Java EE 5, Performance Management and Optimization*. Apress, 2006.

[Hir06] M.J. Hirsch. *GRASP-based heuristics for continuous global optimization problems*. PhD thesis, Department of Industrial and Systems Engineering, University of Florida, 2006.

[HLM+09] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy : a consolidation manager for clusters. In *Proc. of the 2009 ACM SIGPLAN/-SIGOPS Int.l conf. on Virtual execution environments*, VEE '09, pages 41–50, New York, NY, USA, 2009. ACM.

[HM01] S. Haddad and P. Moreaux. *Les réseaux de Petri - Modèles fondamentaux*, chapitre 9 - les réseaux de Petri stochastiques. Information – Commande – Communication. Hermes, Paris, 2001. sous la direction de M. Diaz.

[HSDV10] A. Harbaoui, N. Salmi, B. Dillenseger, and J. Vincent. Introducing queuing network-based performance awareness in autonomic systems. In *Proc. 2010 Sixth International Conference on Autonomic and Autonomous Systems*, ICAS '10, pages 7–12. IEEE Computer Society, 2010.

[Hup09] K. Huppler. The art of building a good benchmark. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 18–30. Springer Berlin / Heidelberg, 2009.

[IBMa] IBM. IBM ILOG CPLEX optimizer. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`. (retrieved on 2012-02-08).

[IBMb] IBM. The oceano project. `http://www.research.ibm.com/oceanoproject/index.html`. (retrieved on 2012-04-03).

[IBM01] IBM. Autonomic computing : IBM's perspective on the state of information technology. IBM internal report, 2001.

[ica04] *International Conference on Autonomic Computing*, New York, NY, USA, May 17-18 2004. IEEE Computer Society.

[IET11] IETF. A simple network management protocol (rfc 1157). `http://www.ietf.org/rfc/rfc1157.txt`, May 2011. (retrieved on 2011-06-11).

[Jai91] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley Professionnal Computing, April 1991.

[Jen00] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence and IJCAI-99*, 117 :277–296, 2000.

[Kan11] M. Kankaanniemi. Self-optimization in autonomic systems. *International Conference of Innovations in Information Technology (IIT)*, 2011.

[KC00] S. Kumar and Ph. R. Cohen. Towards a fault-tolerant multi-agent system architecture. In *Fourth Int. Conference on Autonomous Agents*, pages 459–466. ACM Press, 2000.

[KCS06] A. Konaka, D. W. Coitb, and A. E. Smithc. Multi-objective optimization using genetic algorithms : A tutorial. *Reliability Engineering and System Safety*, 91 :992–1007, 2006.

[KGV83] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598) :671–680, 1983.

[Laba] Columbia University Programming Systems Lab. Kinesthetics extreme. retrieved 2012-06-01.

[Labb] HP Labs. The smartfrog project. (Retrieved on 2012-06-01).

[LIN] LINDO. Lindo systems - optimization software : Integer programming, linear programming, nonlinear programming, stochastic programming, global optimization. `http://www.lindo.com/`. (retrieved on 2012-02-08).

[LPGG00] J. Link-Pezet, P. Glize, and M.-P. Gleizes. An adaptive multi-agent tool for electronic commerce. In *9th IEEE Int. Workshops on Enabling Technologies : Infrastructure for Collaborative Enterprises (WETICE 2000)*, pages 59–66, Gaithersburg, MD, USA, 4-16 June 2000. IEEE Computer Society.

[Max] Maximal Software. Mpl-mathematical programming language. `http://www.maximal-usa.com/mpl/`. (retrieved on 2012-02-08).

[McI68] M. D. McIlroy. Mass produced software components. *Proc. NATO Software Eng. Conf., Garmisch, Germany*, pages 138–155, 1968.

[Mer] Merriam-Webster's Online Dictionary. Optimization definition. `http://www.m-w.com/dictionary/optimization`. (retrieved on 2011-10-03).

[Met12] Metaheuristics.org. The metaheuristics network. `http://metaheuristics.org`, January 2012. (retrieved on 2012-02-08).

[MFB+07] J.D. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea. *Performance Testing Guidance for Web Applications, patterns and practices*. Microsoft press, 2007.

[Mic] Microsoft. Component object model (com).

[MLR03] V. Markl, G. M. Lohman, and V. Raman. Leo : An autonomic query optimizer for db2. *IBM Syst. J.*, 42(1) :98–106, 2003.

[MT00] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), 2000.

[N1S05] N1 sun grid engine, 2005.

[NCFC04] J. Norris, K. Coleman, A. Fox, and G. Candea. Oncall : Defeating spikes with a free-market application cluster. In *IEEE Proceedings of the 1st International Conference on Autonomic Computing (ICAC)*, New York, NY, USA, 2004.

[NSS58] A. Newell, J.C. Shaw, and H.A. Simon. Elements of a theory of human problem solving. *Psychological Review*, 23 :342–343, 1958.

[OL96] I. H. Osman and G. Laporte. Metaheuristics : A bibliography. *Annals of Operations research*, 63 :511–623, 1996.

[OM03] J. O.Kephart and D. M.Chess. The vision of autonomic computing. IBM Thomas J.Watson Research Center, January 2003.

[Ora12a] Oracle. Java management extensions (jmx) technology. `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`, January 2012. (retrieved on 2012-01-03).

[Ora12b] Oracle. Java message service. `http://docs.oracle.com/cd/E19957-01/816-5904-10/816-5904-10.pdf`, January 2012. (retrieved on 2012-01-03).

[OW2] OW2 Consortium. JASMINe, the smart tool for your SOA platform management. `http://jasmine.ow2.org`. (retrieved on 2011-04-07).

[OW212] OW2 Consortium. Joram : Java open reliable asynchronous messaging. `http://joram.ow2.org/`, January 2012. (retrieved on 2012-01-04).

[PBB+08] N. De Palma, S. Bouchenak, F. Boyer, D. Hagimont, S. Sicard, and Ch. Taton. Jade, un environnement d'administration autonome. *Technique et Science Informatiques*, 27(9-10) :1225–1252, 2008.

[PKB07] R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1) :33–57, 2007.

[Pro07] Projet ANR Selfware. Selfware : Lessons learned to build autonomic systems. Technical report, Agence Nationale pour la Recherche, 2007. Livrable Selfware SP1-L3.

[RAC+02] M. J. Rutherford, K. M. Anderson, A. Carzaniga, D. Heimbigner, and A. L. Wolf. Reconfiguration in the enterprise javabean component model. In *Proc. of the IFIP/ACM Working Conference on Component Deployment*, pages 67–81. Springer-Verlag, 2002.

[RBM05] S. Roy, S. Barry, and B. Martin. Pact : Personal autonomic computing tools. In *Proc. of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, ECBS '05, pages 519–527, Washington, DC, USA, 2005. IEEE Computer Society.

[RN04] S. Russell and P. Norvig. *Artificial Intelligence : A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, USA, 2004.

[SBP08] S. Sicard, F. Boyer, and N. De Palma. Using components for architecture-based management : the self-repair case. In *Proc. of the 30th Int. Conf. on Software Engineering*, page 101–110, 2008.

[Sha10] R. Sharrock. *Gestion autonomique de performance, d'énergie et de qualité de service. Application aux réseaux filaires, réseaux de capteurs et grilles de calcul*. PhD thesis, Université de Toulouse, France, 2010.

[SPE] SPEC. Standard performance evaluation corporation. `http://www.spec.org/`. (retrieved on 2011-03-11).

[Szy97] C. A. Szyperski. *Component Software*. ACM Press, New York, 1997.

[Szy02] C. A. Szyperski. *Component Software - Beyond Object-Oriented Programming (Second Edition)*. Addison-Wesley / ACM Press, 2002.

[Tal09] E.-G. Talbi. *Metaheuristics from design to implementation*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2009.

[TBPH08] Ch. Taton, S. Bouchenak, N. De Palma, and D. Hagimont. Self-optimization of internet services with dynamic resource provisioning. Technical Report RR-6575, SARDES, INIRIA Grenoble-Rhône-Alpes, LIG, Montbonnot, France, July 2008.

[Tea09] IBM WPLC Performance Team. IBM websphere portal 6.1.x performance tuning guide. `http://handbook5.com/i/ibm-websphere-portal-6.1.`

`x-performance-tuning-guide-w1796.html`, March 2009. (retrieved 2012-02-17).

[Uni] Berkeley University. The oceanstore project. `http://oceanstore.cs.berkeley.edu`. (retrieved on 2012-01-23).

[WCL+00] A. Wise, A. G. Cass, B. Staudt Lerner, E. K. McCall, L. J. Osterweil, and S. M. Sutton Jr. Using little-jil to coordinate agents in software engineering. In *Proc. of ASE 2000, The Fifteenth IEEE International Conference on Automated Software Engineering*, 2000.

[ZSTF04] Q. Zhang, J. Sun, E. Tsang, and J. Ford. Hybrid estimation of distribution algorithm for global optimization. *Engineering Computations*, 21(1) :91–107, 2004.

Introduction de fonctionnalités d'auto-optimisation dans une architecture de self-benchmarking

El Hachemi BENDAHMANE

Mots-clés : Benchmarking, Optimisation, Auto-optimisation

Le *benchmarking* des systèmes client-serveur implique des infrastructures techniques réparties complexes, dont la gestion nécessite une approche autonomique. Cette gestion s'appuie sur une suite d'étapes, observation, analyse et rétroaction, qui correspond au principe d'une boucle de contrôle autonome. Des travaux antérieurs dans le domaine du test de performances ont montré comment introduire des fonctionnalités de *test autonome* par le biais d'une injection de charge *auto-régulée*. L'objectif de cette thèse est de suivre cette démarche de *calcul autonome* (*autonomic computing*) en y introduisant des fonctionnalités d'*optimisation* autonome. On peut ainsi obtenir automatiquement des résultats de benchmarks fiables et comparables, mettant en œuvre l'ensemble des étapes de *self-benchmarking*.
Notre contribution est double. D'une part, nous proposons un algorithme original pour l'optimisation dans un contexte de test de performance, qui vise à diminuer le nombre de solutions potentielles à tester, moyennant une hypothèse sur la forme de la fonction qui lie la valeur des paramètres à la performance mesurée. Cet algorithme est indépendant du système à optimiser. Il manipule des paramètres entiers, dont les valeurs sont comprises dans un intervalle donné, avec une granularité de valeur donnée. D'autre part, nous montrons une approche architecturale à composants et une organisation du benchmark automatique en plusieurs boucles de contrôle autonomes (détection de saturation, injection de charge, calcul d'optimisation), coordonnées de manière faiblement couplée via un mode de communication asynchrone de type publication-souscription. Complétant un canevas logiciel à composants pour l'injection de charge autorégulée, nous y ajoutons des composants pour reparamétrer et redémarrer automatiquement le système à optimiser.
Deux séries d'expérimentations ont été menées pour valider notre dispositif d'auto-optimisation. La première série concerne une application web de type achat en ligne, déployée sur un serveur d'application Java EE. La seconde série concerne une application à trois tiers effectifs (WEB, métier (EJB JOnAS) et base de données) clusterSample. Les trois tiers sont sur des machines physiques distinctes

Introduction of self-optimization features in a self-benchmarking architecture

El Hachemi BENDAHMANE

Keywords : Benchmarking, Optimization, Self-optimization

Benchmarking client-server systems involves complex, distributed technical infrastructures, whose management deserves an autonomic approach. It also relies on observation, analysis and feedback steps that closely matches the autonomic control loop principle. While previous works in performance testing have shown how to introduce autonomic load testing features through *self-regulated* load injection, the goal of this thesis is to follow this approach of *autonomic computing* to introduce *self-optimization* features in this architecture to obtain reliable and comparable benchmark results. , and to achieve the fully principle of *Self-benchmarking*. Our contribution is twofold. from the algorithmic point of view, we propose an original optimization algorithm in the context of performance testing. This algorithm is divided into two parts. The first one concerns the overall level, i.e. the control of the performance index evolution, based on global parameters setting of the system. The second part concerns the search for the optimum when only one parameter is modified. From the software architecture point of view, we complete the `Fractal` component-based architecture, containing several autonomic control loops (saturation, injection, optimization computing) and we implement the coordination principle between these loops by asynchronous messages according to the publish-subscribe communication paradigm. To apply a given parameters setting on the system under test, we introduced new components to support the setting of parameters before starting the test process. It may also be necessary to restart all or part of the system to optimize to ensure that the new setting is effectively taken into account. We introduced components to cover this need in a specific way for each system. To validate our self-optimization framework, two types of campaigns have been conducted onto the servers of Orange Labs in Meylan and the servers of the LISTIC Laboratory of the University of Savoie in Polytech Annecy-Chambéry (Annecy le Vieux). The first one is a WEB online shopping application deployed on a Java EE application server `JOnAS`. The second one is a three-tiers application (WEB, business (EJB JOnAS) and data base) clusterSample. The three tiers are in three separate machines.