Vers l'Auto-Optimisation dans les Systèmes Autonomes.

Christophe Taton

INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de **DOCTEUR DE L'Institut Polytechnique de Grenoble**

Spécialité : « Informatique : Systèmes et Communication »

préparée au laboratoire LIG dans le cadre de **l'École Doctorale « Mathématiques, Sciences et Technologies de l'Information, Informatique »**

préparée et soutenue publiquement par

CHRISTOPHE TATON

le 24 Novembre 2008

Titre :

Vers l'auto-optimisation dans les systèmes autonomes

---

sous la direction de Jacques Mossière et Sara Bouchenak

JURY

Pr Roger Mohr	Président
Pr Christine Morin	Rapporteur
Pr Peter Van Roy	Rapporteur
Pr Jacques Mossière	Directeur de thèse
Dr Sara Bouchenak	Co-encadrant
Pr Marta Patiño-Martínez	Examinateur

Remerciements

Je remercie mon directeur Jacques Mossière et ma co-directrice Sara Bouchenak, qui ont accepté de diriger mes travaux de recherche. Je remercie également Noël de Palma, Daniel Hagimont et Jean-Bernard Stéfani pour leur encadrement. Plus généralement, je remercie tous les membres du projet Sardes qui m'ont accueilli, guidé et conseillé durant ces années d'effort.
Enfin, je remercie ma famille, et particulièrement mes parents à qui je dois beaucoup. Ma situation actuelle résulte également des interactions que j'ai eues avec l'ensemble des personnes qui ont su, tout au long de mon éducation, répondre à mes questions tout en éveillant et en suscitant mon intérêt et ma curiosité.

Résumé

La complexité croissante des systèmes informatiques rend l'administration des systèmes de plus en plus fastidieuse. Une approche à ce problème vise à construire des systèmes autonomes capables de prendre en charge eux-mêmes leur administration et de réagir aux changements de leur état et de leur environnement. Dans le contexte actuel de l'énergie rare et chère, l'optimisation des systèmes informatiques est un domaine d'administration fondamental pour améliorer leurs performances et réduire leur empreinte énergétique. Gros consommateurs d'énergie, les systèmes actuels sont statiquement configurés et réagissent assez mal aux évolutions de leur environnement, et notamment aux variations des charges de travail auxquelles ils sont soumis. L'auto-optimisation offre une réponse prometteuse à ces différents besoins en dotant les systèmes de la faculté d'améliorer leurs performances de manière autonome.

Cette thèse se consacre à l'étude des algorithmes et mécanismes permettant de mettre en œuvre des systèmes autonomes auto-optimisés. Nous étudions plus particulièrement les algorithmes d'auto-optimisation fondés sur l'approvisionnement dynamique des systèmes afin d'en améliorer les performances et de maximiser le rendement des ressources. Dans le cadre du prototype Jade, plate-forme d'administration autonome à base de composants, nous proposons des algorithmes qui améliorent au mieux les performances des systèmes administrés par des adaptations des systèmes en réponse à des variations progressives ou brutales des charges auxquelles ils sont soumis. Nous montrons l'efficacité de ces algorithmes sur des services Internet et des services à messages soumis à des charges variables. Enfin, dans le but de garantir des performances optimales, nous proposons également une politique d'optimisation qui repose sur une modélisation des systèmes administrés servant à l'élaboration de configurations optimales. Cette politique fait l'objet d'une évaluation sur un service de surveillance d'une infrastructure distribuée.

L'implantation de politiques d'administration autonome fait apparaître un certain nombre de défis en induisant diverses contraintes : le système doit être capable d'adaptation dynamique, de s'observer et de se manipuler. En réponse à ces besoins, nous nous appuyons sur le langage Oz et sa plateforme distribuée Mozart pour implanter FructOz, un canevas spécialisé dans la construction et la manipulation de systèmes à architectures distribuées dynamiques complexes, et LactOz, une bibliothèque d'interrogation des architectures dynamiques. En combinant FructOz et LactOz, on montre comment implanter des systèmes dynamiques complexes impliquant des déploiements distribués avec un haut niveau de paramétrage et de synchronisation.

Mots-clés : Auto-optimisation, Administration et systèmes autonomes, Autonomic computing, Adaptation dynamique, Déploiements et systèmes distribués, Administration fondée sur l'architecture.

Abstract

The increasing complexity of computer systems makes their administration even more tedious and error-prone. A general approach to this problem consists in building autonomic systems that are able to manage themselves and to handle changes of their state and their environment. While energy becomes even more scarce and expensive, the optimization of computer systems is an essential management field to improve their performance and to reduce their energetic footprint. As huge energy consumers, current computer systems are usually statically configured and behave badly in response to changes of their environment, and especially to changes of their workload. Self-optimization appears as a promising approach to these problems as it endows these systems with the ability to improve their own performance in an autonomous manner.

This thesis focus on algorithms and techniques to implement self-optimized autonomic systems. We specifically study self-optimization algorithms that rely on dynamic system provisioning in order to improve their performance and their resources' efficency. In the context of the Jade prototype of a component-based autonomic management platform, we propose best-effort algorithms that improve the performance of the managed systems through dynamic adaptations of the systems in response to gradual or sudden changes of their workload. We show the efficiency of these algorithms on Internet services and on messages services submitted to changing workloads. Finally, in order to guarantee optimal performance, we propose an optimization policy relying on the modeling of the managed system so as to generate optimal configurations. This policy is evaluated on a monitoring service for distributed systems.

The implementation of autonomic management policies raised a number of challenges : the system is required to support dynamic adaptions, to observe itself and to take actions on itself. We address these needs with the Oz programming language and its distributed platform Mozart to implement the FructOz framework dedicated to the construction and handling of complex dynamic and distributed architecture-based systems, and the LactOz library specialized in the querying and browsing of dynamic architectures. Combining FructOz and LactOz, we show how to build complex dynamic systems involving distributed deployments as well as high levels of synchronizations and parameters.

Keywords : Self-optimisation, Autonomic management and systems, Autonomic computing, Dynamic adaptation, Distributed systems and deployments, Architecture-based administration.

Table des matières

Chapitre 1

Introduction

1.1 Motivations

1.1.1 Contexte

Les systèmes informatiques sont de plus en plus complexes. Avec des puissances de calcul en constante progression, on construit des applications offrant toujours plus de fonctionnalités, plus sophistiquées, plus complètes, mais également beaucoup plus complexes. Les systèmes s'appuient sur un écosystème logiciel organisé en couches logicielles : système d'exploitation, bibliothèques de fonctions, intergiciels, machines virtuelles, etc. Chacune de ces briques logicielles est paramétrable et configurable, existe en de multiples versions, dépend potentiellement d'autres briques, et peut également exprimer des contraintes sur son environnement matériel ou logiciel.

Les systèmes distribués sont apparus et se sont généralisés grâce au développement des réseaux, et notamment de l'Internet. Ces systèmes informatiques distribués sont constitués d'un ensemble de machines qui sont interconnectées par des réseaux et qui coopèrent pour assurer une fonction. Leur structure distribuée introduit une nouvelle dimension de complexité observable à tous les niveaux, aussi bien dans leur conception et leur mise en œuvre, que dans leur exploitation et leur administration. Cette complexité se manifeste, d'une part, par les dimensions toujours plus grandes de ces systèmes, de l'ordre de plusieurs milliers de nœuds à ce jour dans certaines infrastructures et, d'autre part, par le dynamisme inhérent à leur environnement, lié par exemple aux pannes et aux surcharges qui peuvent survenir. Ce dynamisme rend nécessaire, d'une part la surveillance en continu de l'état du système afin de détecter les problèmes pour pouvoir envisager d'y remédier et, d'autre part, la construction d'applications adaptables, capables de tolérer le dynamisme. De surcroît, les applications se sont adaptées à l'organisation des systèmes distribués, en adoptant des structurations distribuées du type client/server, multi-niveaux, pair-à-pair, etc, desquelles dérivent de nouvelles dépendances et contraintes qui s'ajoutent aux autres.

Toute cette complexité induit un coût d'exploitation des systèmes distribués élevé : coût en ressources humaines, d'une part, car leur exploitation requiert la présence d'administrateurs humains à haut niveau d'expertise pour établir et maintenir l'état du système ; coût en ressources matérielles, d'autre part, car la solution généralement retenue pour limiter l'impact des incidents (panne, surchage, etc.) consiste à dupliquer et à surdimensionner ces systèmes. Non seulement cela conduit à un gaspillage des ressources, généralement sous-utilisées, et donc d'énergie ; mais en plus, cela ne permet toujours pas de gérer correctement les incidents qui viennent perturber ces systèmes. En effet, plusieurs études montrent que l'opérateur humain est l'une des principales causes de panne des systèmes [88]. Il devient aujourd'hui difficile, voire impossible pour un homme d'intégrer l'ensemble des paramètres, contraintes et dépendances des systèmes. Étant faillible par ailleurs, l'opérateur humain risque lui-même d'introduire des erreurs dans la configuration des systèmes, le rendant potentiellement inopérant. De plus, la réactivité d'un opérateur humain est faible en cas d'incident, ce qui le rend particulièrement mal adapté pour une surveillance continuelle du système.

Tant et si bien que l'administration de ces systèmes, opérée jusqu'à présent exclusivement par des administrateurs humains, est devenue un frein majeur à leur développement. Frein duquel il convient de s'affranchir si l'on souhaite pleinement pouvoir tirer parti de la puissance et de la flexibilité offerte par ces systèmes.

1.1.2 Autonomie

L'*administration autonome* est une tentative initiée en 2001 par IBM pour améliorer cet état de fait, et qui vise à construire des systèmes capables de s'auto-gérer, c'est-à-dire de prendre eux-mêmes en charge les tâches relatives à leur administration [63]. L'objectif de ces systèmes dits *autonomes* est de soulager l'opérateur humain en l'assistant dans les tâches d'administration, et, ultimement, de le remplacer partout ou cela est possible. Le rôle de l'opérateur humain est alors réduit à la définition des politiques d'administration qu'il souhaite voir appliquées au système dont il a la charge, tandis que le système autonome prend lui-même en charge la mise en application et le maintien de ces politiques, quoi qu'il arrive et sans requérir l'intervention de l'homme. Les *systèmes autonomes* séduisent essentiellement par les promesses affichées suivantes :

- simplification considérable de leur administration : le système autonome dirige lui-même les opérations d'administration, réduisant ainsi les erreurs de paramétrage et de configuration introduites par des opérateurs humains ;
- meilleure réactivité du système en cas d'incident : le système autonome est capable de maintenir une surveillance étroite et continue de l'intégralité du système et est ainsi capable de réagir à la moindre alerte. Les délais de diagnostic et de réaction sont écourtés au maximum et l'intervention d'un opérateur humain n'est plus nécessaire ;
- meilleure efficacité du système dans sa globalité : en combinant les deux améliorations précédentes, le système voit sa sensibilité aux incidents fortement réduite, car il est alors capable de s'y adapter plus rapidement. D'autre part, par ses fa-

cultés d'adaptation dynamique, le système peut optimiser son fonctionnement, et notamment l'usage des ressources (par exemple en allouant les ressources à la demande).

L'administrateur humain est alors libéré de ces tâches automatisées, souvent laborieuses et répétitives, et peut désormais mettre à profit son expertise en se concentrant sur des tâches d'administration de plus haut niveau.

1.1.3 Optimisation

À l'heure où la maîtrise de notre consommation d'énergie fait l'objet d'une forte attention, l'optimisation des systèmes informatiques, gros consommateurs d'énergie, prend tout son sens. Et cela est d'autant plus vrai que l'omniprésence de ces systèmes s'intensifie dans notre environnement et dans nos modes de vie. La plupart des objets de notre quotidien font intervenir ces systèmes pour leur conception, leur réalisation ou leur usage, tandis qu'une large majorité de nos activités fait désormais intervenir des systèmes informatiques. L'énergie consommée par les systèmes informatiques peut représenter aujourd'hui jusqu'à 60% de la facture énergétique de certaines entreprises.

Une des principales causes de cette facture énergétique résulte de la structure généralement statique des systèmes informatiques. Cela implique que la capacité d'un système est déterminée une fois pour toutes lors de sa conception. Or de nombreux systèmes doivent faire face à des charges de travail variables et parfois imprévisibles. Ces variations proviennent essentiellement des habitudes généralement prévisibles, journalières ou hebdomadaires, des utilisateurs de ces systèmes (consulter son e-mail en arrivant le matin, consulter les sites d'information généralement autour de midi, etc). Et plus rarement, mais aussi de manière plus importante, ces variations sont liées aux événements de grande ampleur, prévisibles ou non, qui génèrent des effets de masse (des pics de charge, ou *flash crowds*). Pour permettre à ces systèmes statiques d'absorber ces variations, la technique généralement adoptée consiste à surdimensionner le système en lui allouant une quantité de ressources supérieure à ses besoins prévisibles maximum estimés. Cette manière de traiter la question du dimensionnement des systèmes est largement imparfaite : même si elle permet de répondre aux besoins normaux du système, à condition de les avoir correctement estimés, cela conduit d'une part à un gaspillage des ressources matérielles et de l'énergie consommée, car le système sera sous-chargé la plupart du temps. Mais en plus, cela n'est pas pleinement satisfaisant dans la mesure où des surcharges pourront toujours survenir de manière imprévisible. L'écroulement des systèmes des chaînes d'information de la BBC par le Web lors des attentats du 11 Septembre 2001, celui des systèmes de déclaration des revenus en ligne en France la première année de sa mise en place ou encore celui des systèmes de vente en ligne de la SNCF lors d'une opération promotionnelle en raison de la fête des 25 ans du TGV sont autant d'exemples démontrant l'imperfection du dimensionnement statique.

L'optimisation de ces systèmes en réponse aux évolutions de leur environnement est désormais incontournable. Plus particulièrement, l'adaptation à la charge est un

domaine d'administration capital pour les développements futurs des systèmes informatiques, afin de les immuniser contre les surcharges, d'en améliorer le rendement et d'en réduire l'empreinte énergétique. Aussi, l'intégration de ce domaine d'administration dans le cadre de la construction de systèmes autonomes auto-optimisés représente un défi majeur que nous abordons dans ce document.

1.2 Définitions et rappels

1.2.1 Administration et systèmes autonomes

L'*administration* des systèmes est un domaine d'activités qui regroupe un ensemble de tâches relatives à la gestion des systèmes. Un administrateur est responsable du bon fonctionnement des systèmes dont il a la charge. Cela implique une surveillance constante de l'état du système. En cas d'incident, l'administrateur doit inspecter, analyser et diagnostiquer le système afin de cerner l'origine et l'ampleur de l'incident. Enfin, afin de restaurer, maintenir ou améliorer l'état du système, l'administrateur peut décider d'intervenir sur le système. Il est alors maître de la mise en œuvre de ces interventions.

L'administration d'un système informatique comporte de multiples domaines d'administration comme, par exemple, la configuration et le déploiement, la fiabilité et la disponibilité, l'optimisation et les performances, la sécurité, etc. Ces domaines d'administration représentent des tâches d'administration de haut niveau qui consistent en la définition de politiques d'administration : quelle politique mettre en œuvre pour quel niveau de fiabilité, de performance, de sécurité ? Dans le cadre de ces domaines d'administration, l'administrateur est amené à effectuer de multiples interventions, parmi lesquelles on peut citer les suivantes : (dés-)installation, (re-)configuration, paramétrage, (re-)déploiement, mise à jour, démarrage, arrêt, etc. Ces tâches sont spécifiques à chaque système et ont des sémantiques, des dépendances et des interactions les unes envers les autres.

Les *systèmes autonomes* sont des systèmes capables d'effectuer des tâches d'administration de manière autonome et sans requérir l'intervention d'un opérateur humain. Le fonctionnement des systèmes autonomes s'inspire largement de celui du système nerveux du corps humain, notamment en tentant de reproduire certains comportements qui lui permettent de se maintenir en bonne condition. À l'instar du corps humain qui intègre divers systèmes de régulation, comme par exemple un système d'optimisation qui régule naturellement le rythme cardiaque en fonction de l'effort qu'il subit, les systèmes autonomes devraient à terme pouvoir offrir des propriétés autonomes telles que :

auto-configuration : capacité à déterminer et appliquer de manière autonome un paramétrage et une configuration acceptable permettant au système de fonctionner ;

auto-réparation : capacité à détecter, diagnostiquer puis compenser ou réparer des pannes survenant dans le système ;

auto-optimisation : capacité à assurer des niveaux de performances, par l'adaptation du paramétrage et de la configuration du système en réponse à la survenue d'événements liés à l'évolution de son état et de son environnement tels que surcharges, sous-charges, etc ;

auto-protection : capacité à protéger le système contre des actions pouvant déstabiliser le système et le rendre inopérant.

La faculté d'auto-administration des systèmes autonomes s'appuie sur le principe de *rétroaction*, lui-même fondé sur les deux mécanismes suivants : l'auto-diagnostic, aptitude du système à observer son état, sa structure, etc ; et l'auto-manipulation, aptitude du système à agir lui-même sur son état, sa structure, etc.

1.2.2 Optimisation, performances et auto-optimisation

Le domaine d'administration relatif aux performances est celui qui nous intéresse principalement dans ce document. Ce domaine comporte diverses activités : établir un contrat de (qualité de) service définissant le niveau de performances attendu du système, dimensionner, calibrer et ajuster correctement le système pour qu'il opère dans des conditions optimales qui lui permettent de respecter les contraintes établies dans un contrat de service, et enfin, surveiller et optimiser les performances du système en temps réel par diverses adaptations, allant du simple changement de paramètre, à une modification plus profonde, par exemple de son dimensionnement, voire de sa structure.

Le *dimensionnement* du système est une tâche attribuée à l'administrateur qui vise à déterminer précisément les quantités de ressources matérielles qui seront allouées à un système. L'impact du dimensionnement sur les performances du système est multiple : de la quantité de ressources allouées dépendent notamment la rapidité et l'efficacité du système ainsi que sa consommation d'énergie. Changer le dimensionnement de certains systèmes actuels est une opération potentiellement complexe, car elle peut nécessiter de nombreuses modifications de paramètres et éventuellement un redémarrage partiel ou complet du système pour prendre en compte ces changements.

L'*optimisation* 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. Les *indices de performances* que l'on peut considérer sont nombreux et variés. La consommation en mémoire et la consommation en énergie sont deux facteurs critiques dans les systèmes embarqués ou contraints tels que les téléphones ou ordinateurs portables. Les systèmes en interaction avec des utilisateurs souhaitent minimiser la latence des traitements perçue par leurs utilisateurs. Les fournisseurs de services souhaitent maximiser le nombre de requêtes traitées par le système, ou encore son débit. Enfin, l'administrateur d'une grappe ou d'une ferme de machines souhaite optimiser le rendement de son infrastructure ; cela se traduit par exemple par la minimisation du nombre de machines utilisées, afin de minimiser le

coût de l'infrastructure et sa consommation en énergie ; de cet objectif découle un second objectif qui est de maximiser l'utilisation des ressources.

Afin de formuler des exigences relatives aux performances d'un système, on établit un contrat de (qualité de) service (*Service Level Agreement*, ou *SLA*) [69]. Ce contract décrit, dans un ensemble de clauses, les indices de performances qui sont considérés ainsi que les contraintes qui leur sont imposées. À charge pour l'administrateur en charge de l'infrastructure de correctement dimensionner et paramétrer celle-ci afin de respecter les termes du contrat.

1.2.3 Administration fondée sur l'architecture

L'adaptation dynamique d'un système est un besoin inhérent à la définition des systèmes autonomes. En effet, les systèmes autonomes doivent prendre eux-mêmes des décisions concernant leur gestion, et les mettre en œuvre par eux-mêmes et sur eux-mêmes. Un moyen pour faciliter la réalisation de ce type de systèmes consiste à les doter d'une représentation manipulable de leur propre organisation : l'*architecture*.

L'administration fondée sur l'architecture est une manière de construire des systèmes adaptables autonomes qui semble très prometteuse. En effet, ces systèmes intègrent une représentation de leur architecture, ce qui leur permet ainsi de s'introspecter et de s'auto-manipuler.

L'architecture d'un système est une représentation de la structure logique de ce système selon une perspective donnée. Plusieurs perspectives peuvent être envisagées pour des systèmes complexes : structuration du système en un ensemble de sous-systèmes qui interagissent ; représentation de la répartition du système sur les machines physiques (perspective orientée déploiement distribué) ; représentation de domaines d'isolation, d'autorisation et de droits d'accès (perspective orientée sécurité) ; etc. L'architecture est une structure de données qui s'appuie généralement sur un modèle de composant. Les éléments de l'architecture du système tels que sa structure, ses paramètres ou sa configuration y sont représentés essentiellement sous forme de composants, interfaces, liaisons et attributs. L'administration fondée sur l'architecture dépend d'une représentation à l'exécution de l'architecture du système pour répondre au besoin d'adaptabilité dynamique autonome en fournissant au système un moyen de définir, d'observer et de manipuler ses points d'adaptations. Elle repose sur le principe général suivant lequel toute forme d'administration du système doit intervenir au moyen de l'architecture. Ceci nécessite la modélisation dans l'architecture de tout élément de configuration du système qui peut faire l'objet d'une adaptation. Toute opération d'administration revêt alors la forme d'une manipulation de l'architecture du système.

Notre étude s'appuie précisément sur l'administration fondée sur l'architecture pour mettre en œuvre des systèmes autonomes auto-optimisés.

1.3 Problématique

La problématique générale soulevée dans cette thèse concerne l'optimisation des systèmes informatiques dans le cadre de la conception de systèmes autonomes. Un système doté d'auto-optimisation peut être vu sous deux angles : d'une part le système fonctionnel administré qui doit être optimisé ; d'autre part, le système de contrôle, qui implante une politique d'administration, c'est-à-dire dans notre contexte une politique d'optimisation. L'interface entre le système fonctionnel et le système de contrôle est matérialisée par deux activités : l'observation de l'état du système fonctionnel à l'exécution, et sa manipulation par des actions dirigées sur lui. La figure 1.1 présente les composantes d'un système autonome et leurs interactions. De cette problématique générale, nous identifions les trois sous-problèmes suivants.

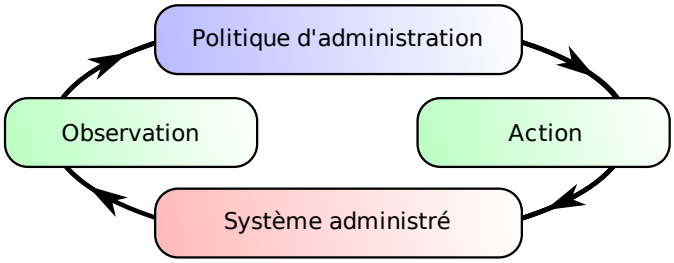


FIG. 1.1 – Structure générale d'un système avec politique d'administration

Politique d'auto-optimisation. Comment optimiser les performances d'un système distribué, en réduire la consommation d'énergie tout en maximisant l'utilisation des ressources (le rendement du système), lorsque l'environnement d'un tel système évolue au cours du temps (charge variable, par exemple) ? Nous nous interrogeons ici, dans un premier temps, sur le moyen d'améliorer les performances du système administré.
Dans un second temps, nous nous interrogeons également sur les moyens de garantir les performances optimales d'un système distribué. La problématique sous-jacente à cette question est celle de la génération et de la sélection d'architectures, configurations et paramétrages garantissant des contraintes de performance.

Action. Quels outils et quels mécanismes sont nécessaires pour mettre en œuvre l'auto-manipulation dans les systèmes autonomes auto-optimisés ? Le système étant représenté et manipulé par son architecture, nous posons ici la question de la manipulation des architectures dynamiques complexes. Agir sur une architecture, structure de données distribuée et dynamique, est une tâche intrinsèquement complexe. Par ailleurs, nous nous interrogeons sur la façon d'intégrer des mécanismes, tels que synchronisation, gestion d'erreur et paramétrage, au cœur des descriptions et des manipulations d'architecture.

Observation. De façon symétrique, quels outils et mécanismes sont nécessaires pour mettre en œuvre l'auto-diagnostic dans les systèmes autonomes auto-optimisés ? De manière analogue, cela pose la question de l'observation des architectures dynamiques complexes. Se repérer dans une architecture distribuée dynamique et en extraire de l'information pertinente est une tâche rendue complexe en raison, notamment, de la nature dynamique de ces architectures. Directement lié à ces questions, se pose le problème de la navigation, de la désignation et du référencement de sous-ensembles pertinents d'une architecture.

1.4 Contributions scientifiques

Les contributions de nos travaux interviennent dans deux domaines différents qui sont, d'une part, le domaine des systèmes distribués et de l'informatique autonome et, d'autre part, le domaine du génie logiciel et des langages. L'objectif de nos travaux est de réunir ici les compétences et les outils issus de ces deux domaines afin de concevoir des systèmes autonomes auto-optimisés. Les principales contributions de nos travaux sont les suivantes.

Politiques d'auto-optimisation. L'originalité de notre approche pour l'auto-optimisation des systèmes distribués est double : (1) elle combine l'*administration fondée sur des modèles architecturaux* à l'*administration fondée sur des modèles comportementaux*, et (2) elle propose une administration fondée sur des *heuristiques* ainsi qu'une administration fondée sur des *modèles mathématiques* [106, 19, 107, 108].

Afin d'optimiser des Services Internet, nous proposons une heuristique d'auto-optimisation fondée sur l'approvisionnement dynamique des ressources et mise en œuvre par une boucle de rétroaction. L'observation d'indicateurs tels que la consommation des ressources physiques ou virtuelles, mais aussi d'indicateurs de niveau applicatif ou utilisateur permet de suivre les évolutions de l'environnement du système et guide ainsi les reconfigurations en s'appuyant sur des seuils. Les reconfigurations visent à mettre en œuvre l'approvisionnement dynamique, c'est-à-dire l'intégration ou le retrait dynamique de ressources pour servir le système auto-optimisé. L'approvisionnement dynamique améliore le rendement des ressources matérielles et contribue ainsi à l'économie d'énergie en collant mieux au besoin du système. Bien qu'approximative et tentant de faire au mieux, cette technique a pour avantages la simplicité de sa mise en œuvre, sa généralité et sa généricité.

Par ailleurs, pour fournir des garanties sur les performances d'un système distribué, nous en avons construit un modèle mathématique, duquel nous extrayons une caractérisation des performances du système en fonction de sa configuration. Cela rend possible la prédiction des performances du système selon sa configuration, et donc le calcul d'une configuration optimale avec garantie de performances.

Action. L'originalité de notre approche pour les actions sur les systèmes distribués réside dans la définition de *nouvelles constructions langage* pour la description d'actions et d'architectures distribuées dynamiques par extension et par intension. Ces constructions langage reposent sur un *modèle d'installation locale* permettant de contrôler précisément le déploiement [102].

Nous assimilons toute manipulation et description d'architecture à une procédure de déploiement distribué. En s'appuyant sur le langage Oz et sa plateforme distribuée Mozart/Oz, nous avons construit le canevas FructOz, qui implante un modèle de composant à la base d'architectures dynamiques. Par divers aspects du langage Oz, FructOz facilite l'intégration d'un haut niveau de paramétrage, de synchronisations et de mécanismes de gestion d'erreurs directement dans les procédures de déploiement distribué, et donc dans les descriptions d'architectures dynamiques. FructOz étend la plateforme Mozart/Oz avec une bibliothèque de primitives de manipulation de composants réutilisables et composables.

Observation. L'originalité de notre approche pour l'observation des systèmes distribués repose sur un *modèle de calcul dynamique distribué* associé à de *nouvelles constructions langage* pour : (1) prendre en compte la nature distribuée de l'état du système administré, (2) décrire l'état dynamique du système administré et (3) naviguer dans l'architecture du système [102].

Nous complétons le canevas à composants FructOz par la bibliothèque LactOz spécialisée dans l'extraction d'informations dans les architectures dynamiques. LactOz intègre des techniques de navigation et de sélection de données dans un environnement de calcul dynamique sur les éléments d'architecture. LactOz facilite notamment l'interrogation des architectures dynamiques, en permettant l'extraction de vues (sous-ensembles) dynamiques de l'architecture, automatiquement mises à jour au gré des évolutions du système. LactOz améliore donc le niveau de compréhension des architectures dynamiques, et simplifie ainsi leur description, leur observation et leur manipulation.

1.5 Principaux résultats

L'évaluation et la validation de nos contributions repose sur une expérimentation diversifiée dans le cadre de plusieurs contextes applicatifs distincts. Nous avons mis en œuvre et validé expérimentalement l'auto-optimisation d'un service à messages en grappe reposant sur le standard JMS et implanté par l'intergiciel JORAM [78, 33]. Nous avons mis en œuvre et validé expérimentalement l'auto-optimisation d'un service Internet de commerce électronique multi-étagé en grappe implantant un service de ventes aux enchères à la eBay [77, 45, 8]. Enfin, nous avons mis en œuvre et validé par des simulations la modélisation d'un service de surveillance réseau de systèmes distribués.

Ces expérimentations ont mobilisé des grappes d'une vingtaine de machines issues de l'environnement expérimental Grid'5000 ainsi que d'une grappe de Mac mini [57].

Nous avons conçu et mis en œuvre le gestionnaire d'auto-optimisation de la plateforme d'administration autonome Jade [106, 19, 107, 108]. Ce gestionnaire intègre différentes politiques d'auto-optimisation des systèmes administrés. Dans le contexte de services à messages ou de services Internet multi-étagés, nous avons implanté deux algorithmes fondés sur des heuristiques permettant l'approvisionnement dynamique, afin de traiter des variations graduelles de charge ainsi que des pics de charge. Dans le contexte d'un système de surveillance réseau, nous avons conçu et implanté un modèle mathématique du système de surveillance permettant de fournir des garanties de performances.

Par ailleurs, nous avons conçu et implanté les canevas logiciels FructOz et LactOz qui intègrent de nouvelles constructions langage pour la description d'actions et d'architectures distribuées dynamiques, ainsi que de nouvelles constructions langage pour l'observation des systèmes distribués administrés fondés sur des architectures dynamiques [102]. FructOz est spécialisé dans la construction de systèmes distribués fondés sur des architectures dynamiques, tandis que LactOz est spécialisé dans leur observation. Nous appliquons ces constructions langage dans le cadre de la conception et de l'implantation de services à messages en grappe, de services Internet multi-étagés en grappe et enfin de services de surveillance réseau. L'exploitation de ces nouvelles constructions langage permet d'intégrer à ces différents services des capacités d'adaptation dynamique et, ultimement, des capacités autonomes d'auto-optimisation par approvisionnement dynamique.

1.6 Plan et organisation du document

Le reste du document est organisé comme suit.

Dans le partie I, nous présentons une revue de l'état de l'art. Le chapitre 2 présente les travaux du domaine de l'informatique autonome. Le chapitre 3 étudie les travaux relatifs à l'optimisation des systèmes distribués et à la modélisation de leurs performances. Le chapitre 4 décrit l'état de l'art relatif à la description et au déploiement des systèmes distribués fondés sur des architectures dynamiques. Enfin, le chapitre 5 s'intéresse aux techniques d'observation des systèmes distribués fondés sur des architectures dynamiques.

Nous décrivons dans la partie II nos différentes contributions, leur conception et leur mise en œuvre concrète. Le chapitre 6 présente le contexte technique et applicatif de la mise en œuvre de nos différentes contributions. Le chapitre 7 décrit la conception et l'implantation de politiques d'auto-optimisation fondées sur l'approvisionnement dynamique. Le chapitre 8 expose la mise en œuvre de notre canevas FructOz spécialisé dans la description de systèmes distribués fondés sur des architectures dynamiques. Le chapitre 9 présente finalement notre bibliothèque LactOz spécialisée dans l'observation et la navigation dans les architectures dynamiques.

Ces travaux font l'objet de validations expérimentales, évaluations, simulations et comparaisons présentées à la partie III. Le chapitre 10 présente l'évaluation de nos po-

litiques d'auto-optimisation par approvisionnement dynamique dans le contexte des services à messages et des services Internet, ainsi qu'une évaluation par simulation du modèle de performance d'un service de surveillance de systèmes distribués. Le chapitre 11 évalue notre contribution sur la description de systèmes distribués fondés sur des architectures dynamiques au moyen du canevas FructOz et de la bibliothèque LactOz.

Enfin, le chapitre 12 présente nos conclusions et perspectives sur nos travaux.

Première partie

État de l'art

Chapitre 2

L'informatique autonome

Nous présentons dans ce chapitre l'état de l'art relatif au domaine de l'administration autonome des systèmes informatiques distribués. En premier lieu, nous précisons en quoi consiste l'administration des systèmes et quels sont les outils dont un administrateur dispose à l'heure actuelle pour l'assister dans ces tâches d'administration. Nous introduisons ensuite l'administration autonome, et plus particulièrement le domaine de l'administration fondée sur l'architecture qui définit par la suite le cadre général de notre étude sur l'auto-optimisation.

2.1 L'administration des systèmes

L'administration des systèmes informatiques est un domaine essentiel dans le cycle de vie des systèmes. Elle regroupe un ensemble de tâches dont l'importance et la délicatesse augmentent avec la complexité et la taille croissante des systèmes d'aujourd'hui. Les systèmes déployés reposent à présent sur de nombreux services, intergiciels et applications interdépendants qu'il devient de plus en plus difficile d'administrer avec les techniques manuelles traditionnelles.

Dans une étude comparant les différentes approches pour l'administration des systèmes, Talwar et al. proposent une classification selon le degré d'automatisation des tâches d'administration [104]. Nous détaillons cette classification dans la suite.

Au niveau le plus bas se trouve l'*administration manuelle*, pour laquelle les diagnostics et les tâches sont réalisés et répétés manuellement par un admistrateur humain. Les tâches d'administration à effectuer comportent l'ensemble de toutes les tâches existantes et imaginables en ligne de commande (par exemple, se connecter à une machine distante, copier et distribuer des fichiers, éditer des fichiers de configuration, vérifier l'état des services, etc).

L'*administration fondée sur les scripts* introduit un premier degré d'automatisation en permettant de regrouper des ensembles de tâches dans des scripts. Les tâches répétées peuvent alors être automatisées dans des boucles, ainsi que les déploiements et certaines boucles de commande simples. Les scripts peuvent également déclencher et

réagir à des évènements. Les tâches d'administration se réduisent alors à l'invocation de quelques scripts qui automatisent les processus logiques du déploiement (diffusion d'un logiciel, démarrage du système, vérification de l'état, etc). Cela correspond, par exemple, au fonctionnement d'une partie des systèmes Unix et Linux, dont l'administration repose sur un ensemble de scripts (notamment ceux contenus dans les répertoires `/etc/rc?.d/` et `/etc/init.d/`), ou encore au fonctionnement du serveur de bases de données MySQL qui s'appuie sur le script `mysqld_safe` pour surveiller l'état du démon `mysqld` et le redémarrer automatiquement en cas de panne [1]. Citons encore le système évènementiel plus récent Upstart qui permet de gérer le cycle de vie des services dans les distributions Ubuntu Linux en s'appuyant sur des scripts capables de déclencher et réagir à divers évènements [73].

L'*administration fondée sur un langage* est un autre degré d'administration qui intègre des facultés plus évoluées, comme la description des systèmes administrés ou de leurs dépendances, la gestion de leur déploiement ou de leur cycle de vie. Le système de gestion des services des distributions Gentoo Linux fonctionne sur ce modèle, en s'appuyant sur une extension d'un langage de script permettant de décrire les dépendances entre services, afin d'ordonner et de coordonner leurs déploiements ainsi que les opérations sur leurs cycles de vie respectifs. La plate-forme SmartFrog représente l'exemple emblématique d'administration fondée sur un langage de description et sur une plate-forme de déploiement et de gestion du cycle de vie [56]. Elle propose notamment une bibliothèque de composants de description des flots d'exécution (workflow) autorisant un haut degré de contrôle sur l'exécution des tâches d'administration des systèmes.

Enfin, l'*administration fondée sur les modèles* constitue selon la classification citée le plus haut degré d'automatisation de l'administration des systèmes. Reposant sur l'exploitation de modèles du système, ce type d'administration permet l'élaboration de politiques d'administration de haut niveau décrites selon les concepts représentés par ces modèles. L'utilisation d'un modèle, lorsque celui-ci est maintenu en cohérence avec l'état réel du système permet en outre de mieux capturer et traiter le dynamisme des systèmes administrés.

2.2 L'administration autonome et l'auto-optimisation

L'administration autonome désigne la capacité des systèmes à administrer eux-mêmes les éléments qui les composent, en l'absence d'intervention humaine, pour en assurer le bon fonctionnement [52, 70]. Les systèmes dits autonomes sont dotés de capacités d'administration autonome. À la manière dont fonctionnent les nombreux systèmes de régulation du corps humain, les systèmes autonomes devront à terme intégrer des capacités autonomes telles que l'auto-configuration, l'auto-réparation, l'auto-protection ou encore l'auto-optimisation. Pour l'administrateur humain, les tâches d'administration prennent alors une nouvelle forme, qui consiste à définir des politiques d'administration de haut niveau. Ces politiques seront interprétées et mises en application automatiquement par le système autonome.

Les capacités autonomes d'un système informatique peuvent être mises en relation

avec les différents niveaux d'administration identifiés et présentés à la section précédente [104]. Ainsi, l'administration manuelle n'offre aucune capacité d'administration autonome ; l'administration fondée sur des scripts offre des capacités autonomes primitives reposant sur des réactions entièrement préprogrammées et non adaptables ; l'administration fondée sur un langage apporte des capacités d'adaptation et d'analyse simples, reposant par exemple sur les dépendances ; enfin l'administration fondée sur les modèles offre les capacités autonomes les plus évoluées.

La mise en œuvre des systèmes autonomes repose sur trois principes fondamentaux : (1) l'auto-diagnostic, qui repose sur une connaissance de sa propre structure et permet au système de découvrir et d'évaluer son état par l'*observation* ; (2) la rétroaction par une boucle de commande, qui autorise le système à réagir sur lui-même en réponse à des modifications de son état, selon des *politiques d'administration* ; et enfin (3) l'auto-manipulation par le biais de reconfigurations dynamiques, en vue de modifier la configuration du système administré pendant son fonctionnement pour mettre en œuvre des *actions*. La figure 2.1 illustre la structure fondamentale d'un système autonome telle que nous venons de la décrire.

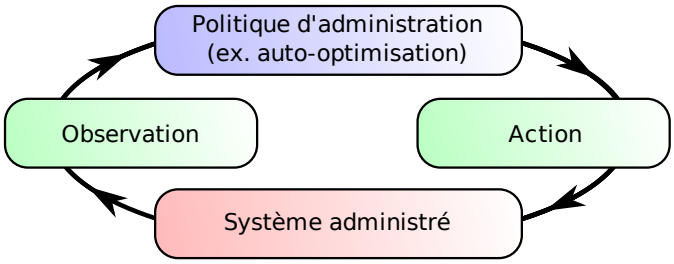


FIG. 2.1 – Structure fondamentale d'un système autonome.

Divers travaux étudient l'administration des systèmes distribués avec comme perspective l'auto-optimisation des systèmes administrés. Ces systèmes reposent généralement sur le principe de la rétroaction mis en œuvre par des boucles de commande. C'est précisément ce qu'implantent Diao et al. pour la régulation d'un serveur Web Apache HTTP [43, 74] ; ou encore Soundararajan et al. pour la régulation d'un serveur de bases de données dupliqué en grappe [100, 101].

Plus généralement, Diao et al. étudient comment intégrer les techniques de régulation issues de la théorie de la commande à la conception de systèmes autonomes et plus particulièrement à la conception de systèmes auto-optimisés [44]. Ce travail conclut sur les défis à venir lié à l'intégration de ces techniques, dont notamment, la conception de modèles fiables des diverses ressources, l'intégration des délais induits par les sondes servant à surveiller le système, l'intégration des latences des opérations agissant sur le système ou encore la mise en œuvre de bancs de mesures permettant d'étalonner et de calibrer les modèles (identification des paramètres et validation du modèle).

2.3 Administration fondée sur les modèles d'architecture

L'*administration fondée sur l'architecture* est une spécialisation de l'administration fondée sur les modèles avec pour modèle principal l'architecture du système administré. Van der Hoek et al. introduisent en effet l'architecture comme modèle unifié pour la conception de systèmes, la gestion de configuration et la (re-)configuration des systèmes distribués [121]. L'architecture repose sur un modèle de composant, pour représenter la structure d'un système à la granularité d'un composant ainsi que les relations entre ces composants en les rendant explicites au moyen de liaisons. L'architecture devient alors le support de la connaissance requise du système administré pour permettre la mise en œuvre des systèmes autonomes.

White et al. identifient et définissent en outre des patrons d'architecture des systèmes autonomes ainsi que différents éléments d'architecture et de conception concourant à leur réalisation [125]. Un système autonome est ainsi conçu comme un assemblage d'éléments autonomes qui interagissent. Chaque élément autonome est responsable de son administration interne, qui est pilotée par un ensemble de politiques d'administration. Les éléments autonomes exposent des interfaces, parmi lesquelles une interface de surveillance permettant d'observer son état, une interface de liaison permettant de contrôler ses relations avec les autres éléments autonomes, une interface de cycle de vie, ou encore une interface de contrôle des politiques d'administration.

Les systèmes fondés sur les architectures reposent généralement sur : (1) un modèle de composant, qui détermine les concepts servant à représenter les éléments d'architectures (composants, ports ou interfaces, connecteurs ou liaisons, etc), et (2) un langage de description d'architecture (*Architecture Description Language*, ou *ADL*).

L'*architecture* est un modèle qui peut être exploité de plusieurs manières. Une utilisation primitive de l'architecture consiste à guider la phase de conception d'un système autonome. C'est notamment l'exploitation qui en est faite par Garlan et al. avec le canevas *Rainbow* spécialisé dans la construction de systèmes autonomes [28, 53]. *Rainbow* permet de modéliser l'architecture des systèmes distribués et permet ainsi d'exprimer des règles sur ces architectures, dont notamment des contraintes et des reconfigurations dynamiques. Les règles et les reconfigurations dynamiques sont intégrées au système à l'exécution, réalisant ainsi un système autonome.

Dans le cadre de l'administration fondée sur l'architecture, l'architecture peut également être réifiée à l'exécution et servir de support à la mise en œuvre des opérations d'administration. C'est, par exemple, ce qu'implante le canevas *Plastik* spécialisé dans la construction de systèmes à architectures dynamiques [66]. *Plastik* repose sur une extension du langage ADL de description d'architecture *ACME/Armani* [54, 85], et sur le modèle de composant et son infrastructure d'exécution *OpenCOM* [37]. *ACME* est un ADL très général qui permet, grâce à son extension *Armani*, l'expression de règles architecturales comme des contraintes ou des invariants, mais n'autorise pas la description de reconfigurations dynamiques de l'architecture. Cet ADL a été étendu dans le cadre du canevas *Plastik* pour permettre la description de telles reconfigurations architecturales sous la forme de règles ECA (Évènement-Condition-Action) du type : on

(prédicat) do (action). Dans *Plastik*, la connexion causale maintenue entre le système à l'exécution et l'architecture, où *architecture réifiée*, est un moyen de manipuler le système à l'exécution par l'intermédiaire de son architecture.

2.4 Synthèse

Nous avons présenté dans ce chapitre les principes de l'administration des systèmes informatiques distribués et nous avons introduit les concepts généraux de l'administration autonome des systèmes. Nous distinguons deux grandes lignes d'approches pour l'administration autonome et l'auto-optimisation des systèmes distribués. Certaines approches guident la conception de systèmes auto-optimisés en partant des comportements attendus des systèmes administrés, exprimés sous forme de niveaux ou contraintes de performance et de qualité de service. D'autres approches conçoivent l'administration autonome des systèmes en raisonnant sur leur structure, à partir de modèles architecturaux des systèmes administrés. C'est l'*administration fondée sur l'architecture*.

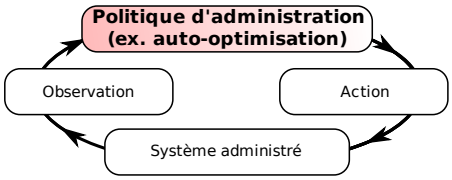
Nous proposons une nouvelle approche, qui combine l'administration fondée sur les modèles architecturaux à l'administration fondée sur des *modèles comportementaux* des systèmes administrés, afin de concevoir des systèmes autonomes auto-optimisés. Il s'agit ici de concevoir des systèmes auto-optimisés fondés sur des architectures qui intègrent directement des modèles et contraintes de performances et de qualité de service des systèmes administrés.

Ceci fait l'objet du chapitre 7.

Nous avons introduit, dans ce chapitre, l'administration et les systèmes autonomes, avec comme objectif la construction de systèmes autonomes auto-optimisés. Dans le chapitre suivant, nous étudions les travaux apparentés d'auto-optimisation des systèmes distribués.

Chapitre 3

Politiques d'auto-optimisation



Ce chapitre est consacré à la présentation des approches existantes pour l'optimisation des performances et de la qualité de service des systèmes distribués. Dans un premier temps, nous rappelons quelques concepts de qualité de service et de gestion des performances. Nous identifions, par la suite, plusieurs défis scientifiques qui découlent de la problématique de l'optimisation des systèmes distribués. Nous présentons ensuite les travaux apparentés d'optimisation des systèmes distribués. Enfin, nous présentons une synthèse de ces travaux.

3.1 Définitions et rappels

3.1.1 Qualité de service

La *qualité de service* (*Quality of Service*, ou *QoS*) correspond littéralement à un niveau de performances d'un service fourni. Il existe une multitude de critères de performance pouvant servir à la définition de niveaux de qualité de service. Par exemple, les performances d'une infrastructure matérielle sont généralement évaluées par son coût de fonctionnement, son rendement énergétique, le taux d'utilisation de ses ressources, sa disponibilité, etc. Les performances d'un service Internet sont évaluées par le débit de requêtes utilisateur traitées par l'infrastructure sous-jacente, par la latence de traitement des requêtes utilisateur, par la disponibilité du service, etc.

Lors de la mise en place d'un service, on établit généralement un contrat de niveau de service (*SLA*, *Service Level Agreement*) qui spécifie précisément les objectifs et les contraintes sur une combinaison de critères de performances [69]. L'analyse de ce

contrat permet de réaliser le dimensionnement du système. Cette tâche consiste à déterminer précisément les ressources (leur quantité et leurs caractéristiques) qui doivent être mobilisées pour permettre au service mis en œuvre de respecter les critères de qualité de service établis. Le système est alors caractérisé par sa capacité (de traitement), qui détermine la quantité de travail qu'il est capable de traiter conformément aux exigences de qualité de service et de performances associées au système. La charge instantanée d'un système est le rapport entre la quantité de travail soumise au système et sa capacité. Une charge très inférieure à 1 indique que le système est sous-utilisé et gaspille des ressources ; une charge supérieure à 1 indique que le système est surchargé, ce qui entraîne généralement une dégradation des performances.

3.1.2 Mécanismes de gestion de performances

La figure 3.1 illustre la mise en œuvre d'un service par un système informatique. Lorsqu'un client soumet un travail au système, le travail est évalué par un contrôleur d'admission et peut être refusé ou bien admis dans le système. Une fois admis dans le système, le travail est classifié en fonction de certaines caractéristiques (comme par exemple sa nature, son coût, etc). Le système distingue alors plusieurs classes de travaux, auxquels sont associés différents niveaux de services (comme par exemple, des priorités absolues). L'ordonnanceur applique cette différenciation de service en choisissant quel travail est effectivement traité par les ressources du système.

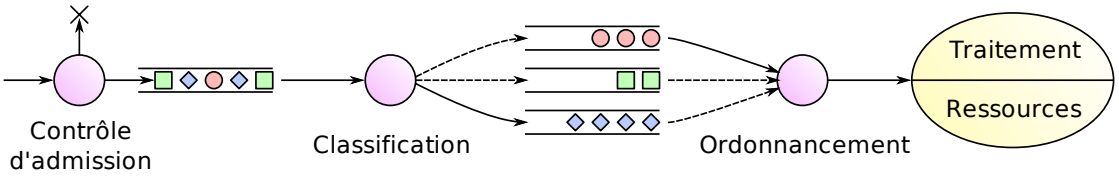


FIG. 3.1 – Fonctionnement d'un système informatique pour réaliser le traitement d'un travail.

Au cours du traitement d'un travail dans le système illustré sur la figure 3.1, nous distinguons six grands mécanismes et outils pour le contrôle des performances :

Contrôle d'admission : Le contrôle d'admission gouverne directement la quantité de travail admise et effectivement traitée dans le système. En refusant un travail, le contrôle d'admission préserve les ressources du système et donc la qualité de service des traitements en cours dans le système. Cela se répercute néanmoins par une baisse immédiate de la disponibilité du service aux clients, fortement dommageable pour la qualité générale du service.

Ordonnancement : L'ordonnanceur applique les règles discriminant les différentes classes de requêtes admises dans le système en choisissant à quel travail attribuer les ressources. Il existe de très nombreuses stratégies d'ordonnancement, certaines étant parfois spécialisées pour des contextes particuliers (priorités strictes, FIFO, Start-time Fair Queuing (SFQ), Hierarchical SFQ (HSFQ), Lottery Scheduling [122],

Locality-aware Request Distribution [90], etc). En modifiant la stratégie d'ordonnancement et/ou son paramétrage, en ajustant notamment par les niveaux de priorités, on contrôle les performances et la qualité de service du système.

Dégradation : La dégradation consiste à adapter le travail accepté dans le service, pour réduire son impact sur les ressources du système, en acceptant une dégradation de la qualité de son traitement. La dégradation permet d'atteindre un compromis entre le refus d'un travail par le contrôleur d'admission, et la baisse de la qualité de service engendrée par l'admission d'une trop grosse quantité de travail dans le système. La dégradation est par exemple implantée de manière visible par le service d'indexation et de recherche bibliographique CiteSeer [29], qui limite le nombre et le niveau de détail des réponses renvoyées au client lorsqu'il est trop chargé.

Dimensionnement et approvisionnement dynamique : Le dimensionnement consiste à déterminer la quantité de ressources composant le système. En agissant sur la quantité de ressource du système, on contrôle sa capacité de traitement. Cela a un impact direct sur les performances des traitements effectués, et donc sur la qualité de service. Lorsque le dimensionnement est ajusté durant le fonctionnement du système, on parle d'approvisionnement dynamique.

Modélisation et évaluation des performances : Dans le contexte de la gestion de performances des systèmes, les modèles mathématiques sont un puissant outil, dont l'exploitation peut intervenir dans chacun des mécanismes de contrôle décrits précédemment. Un modèle peut permettre d'estimer le coût du traitement d'un travail dans le système, aidant ainsi à la prise de décision du contrôleur d'admission ou de l'ordonnanceur. Dans le cadre du dimensionnement du système, un modèle peut permettre d'estimer les performances, la qualité de service du système ou encore sa capacité en fonction de sa configuration. Il est également possible de modéliser la charge de travail soumise au système afin de prévoir et d'adapter à l'avance le dimensionnement du système, par exemple pour suivre les évolutions prévisibles (notamment journalières ou hebdomadaires) de la fréquentation des services Internet.

Modélisation et profils d'exécution : Les profils sont des modèles probabilistes assez simples réalisés expérimentalement, en testant le comportement du système en réaction à des stimulations unitaires isolées. L'exploitation de profils fait l'hypothèse que le comportement du système examiné est statistiquement déterministe. Une utilisation courante des profils consiste à étudier l'impact du traitement d'une requête sur les ressources du système. Cette étude permet de constituer les profils des besoins en ressources des différentes classes de requêtes admises dans le système. Diposant de ces profils, il devient ainsi possible de prévoir à l'avance l'impact du traitement d'une requête avant son admission dans le système, ce qui permet de décider si son admission est acceptable ou non pour les performances du système. A contrario, les profils peuvent guider le dimensionnement ou l'approvisionnement dynamique du système en fonction des requêtes qui lui sont soumises.

En résumé, la gestion des performances des systèmes informatiques est un vaste domaine de recherche faisant appel à des approches techniques (p.ex. le contrôle d'admission), algorithmiques (p.ex. l'ordonnancement) et mathématiques (p.ex. les modélisations).

3.2 Défis scientifiques des politiques d'auto-optimisation des systèmes distribués

Nous identifions et détaillons dans cette section plusieurs défis scientifiques qui découlent de la problématique d'auto-optimisation des systèmes distribués.

Gestion des variations de charge. La gestion des variations de charge est essentielle pour le bon fonctionnement des systèmes distribués administrés. Les variations de charges induisent progressivement ou brutalement des situations de surcharge ou de sous-charge dans le système distribué. Un système distribué en situation de sous-charge gaspille des ressources matérielles et de l'énergie. À l'inverse, un système distribué surchargé est dans l'incapacité de satisfaire correctement l'ensemble des tâches qui lui sont attribuées. Il fonctionne ainsi de manière dégradée et devient potentiellement sujet à l'écroulement.

Garanties de performance. Une caractéristique particulièrement intéressante consiste à pouvoir fournir des garanties sur les performances du système distribué administré dans le cadre d'une politique d'auto-optimisation donnée. Les contrats de qualité de service (*SLA*) reposent principalement sur la capacité à établir des garanties de performance. Par exemple, dans le cas d'un contrat de qualité de service établi entre un hébergeur et un fournisseur de service, l'hébergeur s'engage généralement à garantir les performances de l'infrastructure mise à disposition du fournisseur de service (par exemple, la puissance de calcul ou la bande passante réseau fournie au service).

Gestion des systèmes patrimoniaux. La gestion des systèmes patrimoniaux est un défi important qu'il convient de considérer pour la conception de systèmes auto-optimisés. En effet, de nombreux systèmes actuels sont des systèmes patrimoniaux qui doivent ainsi être assimilés à des boîtes noires. Ceci est dû à l'inaccessibilité de leur code source (code ou licence propriétaire et fermée), ou à l'effort nécessaire trop lourd pour adapter et intégrer ces systèmes à une nouvelle approche d'auto-optimisation. La gestion de ces systèmes patrimoniaux est donc une caractéristique essentielle dont l'absence conduit à ignorer un grand nombre des systèmes existants.

Gestion des oscillations et de la stabilité du système administré. L'auto-optimisation d'un système distribué repose sur des adaptations dynamiques qui visent à en améliorer les performances. Les adaptations dynamiques perturbent la stabilité du système

administré. Lorsque ces instabilités sont mal interprétées, cela peut conduire à des oscillations issues de sur-réactions ou de réactions contraires injustifiées. Les systèmes distribués sujets à des instabilités ou à des oscillations ont des performances dégradées qui peuvent potentiellement écrouler le système administré et le rendre ainsi inopérant. Il est ainsi essentiel de prendre en compte la stabilité du système administré dans la conception d'une politique d'auto-optimisation.

Généralité et réutilisabilité de l'approche. La généralité d'une politique d'auto-optimisation est un critère important. Elle a un impact immédiat sur le degré de réutilisation de la politique dans des contextes et circonstances différents. Les politiques d'auto-optimisation spécialisées pour des systèmes distribués particuliers ou associées à des contraintes de mise en œuvre trop restrictives ont un champ d'application limité.

3.3 Travaux apparentés d'auto-optimisation des systèmes distribués

Nous présentons dans cette section les travaux apparentés d'auto-optimisation des systèmes distribués au regard des défis scientifiques présentés précédemment.

3.3.1 Gestion des variations de charge

Il existe plusieurs catégories de travaux qui s'intéressent à la gestion des variations de charge. Certains travaux se limitent à la gestion de la contention afin de limiter l'effet d'une surcharge, tandis que d'autres travaux traitent effectivement les variations de charge par adaptation à la charge et approvisionnement dynamique.

(a) Gestion de la contention. Une première catégorie de travaux s'attaque spécifiquement à la gestion de la contention. Ces travaux traitent spécifiquement la surcharge en limitant ses effets sur le système administré et notamment en prévenant son écroulement. Certaines des approches minimisent les effets de la contention liée à la surcharge en appliquant des priorités à différentes classes de travaux. Toutefois, aucune des approches dans cette catégorie ne traite les causes des surcharges, à savoir les variations de charge.

i. Dégradation de service. La dégradation de service est un moyen de gérer la contention par adaptation de la qualité et du coût des traitements à la quantité de ressources disponibles. Cette technique permet ainsi de traiter dans une certaine mesure les variations de charge en fournissant des traitements de qualité moindre.

Abdelzaher et al. étudient le contrôle de la dégradation de qualité de service par l'adaptation des contenus afin de préserver les performances du système [3, 4]. À mesure que la charge de travail du système se rapproche de la limite de sa capacité de

traitement, les traitements des requêtes sont dégradés. Cela se traduit pour les utilisateurs du service par des contenus de qualité moindre, comme par exemple, des images dont le taux de compression est plus élevé ou dont la résolution est plus faible, afin de réduire leur taille physique et donc le coût lié à leurs transferts aux utilisateurs. Différents jeux de contenus statiques sont ainsi précalculés avec différents niveaux de dégradation. La dégradation ultime étant le rejet pur et simple des requêtes par contrôle d'admission.

Welsh et al. proposent le canevas *SEDA* qui permet la maîtrise de la dégradation des performances par conception du système administré [124, 123]. *SEDA* est un canevas pour la construction de systèmes étagés à évènements. Cette structuration particulière des systèmes autorise un contrôle fin sur la répartition des ressources entre les différents étages composant le système. Cela améliore la robustesse du système en cas de surcharge, car la structure étagée et évènementielle du système en exclut naturellement l'écroulement. *SEDA* intègre un contrôleur de gestion de la surchage qui optimise par contrôle d'admission les latences de traitement des requêtes, ce qui autorise ainsi le contrôle de la dégradation de la qualité de service lors des surcharges.

ii. Isolation de performance. L'isolation de performance est un outil de gestion de contention en environnement partagé par plusieurs services concurrents. Il s'agit d'empêcher qu'un service surchargé ne perturbe le fonctionnement des autres services avec lesquels il partage l'infrastructure. C'est une technique particulièrement utile pour s'accommoder des surcharges mais qui ne résoud pas le problème de fond des variations de charge.

Aron et al. s'intéressent avec *Cluster Reserves* à l'isolation des performances entre plusieurs services concurrents hébergés sur une même grappe de machines [12]. Ces travaux s'appuyent et étendent les concepts de *Resource Containers* pour l'isolation de performance localement à une machine définis par Banga et al. [14]. *Cluster Reserves* étend ainsi le mécanisme de réservation de ressources à l'échelle d'une grappe de machines, mais permet aussi la redistribution des ressources inutilisées tout en forçant l'équilibrage global des allocations. De cette façon, lorsqu'un service entre en surcharge, cela n'a pas de conséquence sur les performances des autres services hébergés par la plate-forme.

Urgaonkar et al. décrivent une technique fondée sur la sur-réservation contrôlée des ressources (*Resource Overbooking*), afin d'optimiser l'exploitation des ressources du système administré, tout en bornant statistiquement et par construction l'apparition de surcharges [116]. Dans cet objectif, des profils des différents systèmes administrés sont construits expérimentalement hors-ligne, desquels on dérive les besoins en ressources de chacun des systèmes. Ces informations dirigent un algorithme de placement qui implante la sur-réservation des ressources, de façon à maximiser l'exploitation des différentes ressources, tout en autorisant de façon statistiquement limitée les possibilités de surcharges. Les systèmes sont toutefois isolés au moyen d'ordonnanceurs bas niveau qui appliquent les réservations de ressources.
Sharc étend l'étude préliminaire précédente réalisée sur la sur-réservation des ressources,

et intègre des mécanismes d'allocation dynamique des ressources permettant d'arbitrer la répartition des ressources entre de nombreuses applications concurrentes hébergées sur une grappe de machines [114]. *Sharc* implante une politique d'équilibrage des allocations de ressources à l'échelle de la grappe de machines similaire à *Cluster Reserves*. Cette politique assure la redistribution des ressources inutilisées tout en garantissant aux systèmes administrés la récupération des ressources qui leur sont allouées si besoin.

iii. Ordonnancement. L'ordonnancement est une technique permettant de choisir les tâches à traiter en priorité. Lors d'une surcharge, cela permet de maximiser le rendement des ressources saturées en traitant en priorité les tâches les plus rentables.

Shen et al. proposent la plate-forme *Neptune* spécialisée dans l'hébergement de services Internet [98, 97]. La plate-forme *Neptune* contrôle la répartition des ressources d'une grappe de machines au moyen d'un ordonnancement à deux niveaux. Pour cela, différentes classes de requêtes sont considérées. Un premier ordonnanceur équilibre la charge à l'échelle de la grappe de machines et dirige les nouvelles requêtes sur les machines les moins chargées, en assurant une répartition homogène des différentes classes de requêtes sur l'ensemble de la grappe. Un second ordonnanceur travaille localement sur une machine et choisit parmi l'ensemble des requêtes qui lui sont adressées celles à traiter en priorité. *Neptune* identifie les priorités en fonction d'une estimation paramétrable des revenus générés par le traitement de chaque classe de requêtes. Cette caractérisation des revenus générés constitue un critère d'optimisation qui guide l'ordonnanceur. En cas de surcharge, *Neptune* optimise ainsi l'exploitation des ressources afin de générer des revenus maximum.

Patiño et al. ont conçu et implanté l'intergiciel *Middle-R* spécialisé dans l'optimisation des grappes de serveurs de bases de données répliqués [83, 92]. L'intergiciel *Middle-R* gère les transactions et ordonnance les requêtes SQL selon divers algorithmes sur les serveurs de bases de données. L'objectif de l'ordonnanceur est ici d'équilibrer au mieux la charge sur l'ensemble des serveurs afin de maximiser les performances de la grappe de serveurs. *Middle-R* combine cette technique à une optimisation locale de chacun des serveurs de bases de données qui consiste en l'adaptation dynamique du taux de parallélisme (*MultiProgramming Level*, ou *MPL*) de ces serveurs en fonction de la charge qui leur est soumise.

Enfin, Zhu et al. proposent *DDSD* (*Demand-Driven Service Differentiation*), une approche hybride qui mélange gestion de contention et approvisionnement dynamique [128]. *DDSD* implante un partitionnement dynamique d'une grappe de machines par l'ordonnancement des requêtes sur des sous-ensembles disjoints de machines de la grappe. Il s'agit en réalité d'un partitionnement virtuel des machines mis en œuvre par ordonnancement. Le partitionnement permet ainsi d'isoler et d'adapter les capacités de traitement réservées à différentes classes de requêtes, en recalculant périodiquement les nouvelles frontières virtuelles du partitionnement.

(b) Approvisionnement dynamique. Une seconde catégorie d'approches s'appuie sur des techniques d'adaptation à la charge spécifiquement fondées sur l'approvisionnement dynamique pour traiter les variations de charge. L'intérêt de l'approvisionnement dynamique est double : d'une part, il permet de traiter effectivement les variations de charge en mobilisant les ressources permettant d'absorber un niveau de charge donné, et d'autre part il permet d'éviter le gaspillage des ressources et de l'énergie en réaffectant ou en désactivant les ressources inutilisées.

Soundararajan et al. décrivent dans le cadre du projet *Chameleon* l'optimisation par approvisionnement dynamique d'une grappe de serveurs de bases de données dupliqués [100, 101]. Pour répondre à d'éventuels pics de charge, *Chameleon* conserve en permanence un ensemble de machines hébergeant des instances inutilisées et déconnectées des bases de données administrées, mais qui sont régulièrement maintenues à jour. Cela contribue à améliorer sensiblement la latence des opérations d'approvisionnement dynamique, en optimisant notamment la mise en cohérence des nouvelles instances des serveurs de bases de données.

Appleby et al. proposent la plate-forme *Océano* d'hébergement d'un service Internet structuré selon l'architecture multi-étagée J2EE [10]. *Océano* est spécifiquement conçue pour réagir aux variations de charges et implante l'approvisionnement dynamique des services Internet par partitionnement dynamique d'une grappe de machines. L'architecture multi-étagée du service Internet administré guide le partitionnement : une première partition est spécialisée dans la répartition de la charge sur le système ; une seconde partition héberge les serveurs d'applications supposés sans état (les *dauphins*), ce qui permet d'adapter dynamiquement la taille de cette partition ; enfin, une dernière partition fixe et non adaptable héberge les serveurs de bases de données (les *baleines*). Les opérations d'approvisionnement dynamique consistent à allouer puis préparer les machines intégrées à une partition. Les machines sont alors réinstallées à partir d'une image de système d'exploitation. Cela entraîne une latence des opérations d'approvisionnement dynamique particulièrement longue, de l'ordre de 5 à 10 minutes.

Noris et al. réimplantent l'approche suivie dans *Océano* et conçoivent la plate-forme *OnCall* en ciblant spécifiquement l'amélioration de la latence des opérations d'approvisionnement dynamique [89]. L'objectif de la plate-forme *OnCall* est en effet d'accélérer les réactions d'approvisionnement afin de pouvoir réagir face à des pics de charge. Pour cela, *OnCall* exploite des machines virtuelles *VMWare* pour réaliser l'infrastructure hébergeant les services Internet [64]. Les machines virtuelles sont intéressantes pour l'isolation et pour leur vitesse d'activation qui est alors réduit à environ une minute.

Chase et al. présentent la plate-forme d'hébergement *Muse* [26]. Cette approche se différencie de la plate-forme *Océano* en permettant d'accueillir plusieurs systèmes administrés qui partagent un même grappe de machines. *Muse* s'intéresse notamment à l'optimisation de la consommation d'énergie. En particulier, *Muse* cherche ainsi à optimiser l'exploitation et le rendement des ressources en concentrant les systèmes administrés sur un nombre minimal de machines. Moore et al. proposent la plate-forme *Cluster-On-Demand* (*COD*) également largement inspirée par *Océano* [86]. Il s'agit d'une approche hybride entre *Océano* et *Muse* qui permet de partager une grappe de machines

en différentes grappes de machines virtuelles, chaque grappe virtuelle étant réservé à un système administré. Les grappes virtuelles sont mises en œuvre et isolées par des réseaux virtuels à l'aide de switchs réseaux reconfigurables.

Enfin, Urgaonkar et al. proposent la plate-forme pour services Internet *Cataclysm* spécifiquement destinée à la gestion des pics de charge extrêmes [112, 115]. *Cataclysm* se présente comme une extension de la plate-forme *Sharc* présentée précédemment, et introduit l'approvisionnement dynamique par partitionnement dynamique d'une grappe de machines. *Cataclysm* intègre en outre une différenciation de services fondée sur des classes de requêtes, un contrôleur d'admission, un ensemble de répartiteurs de charge dynamiquement approvisionné et enfin un ensemble de serveurs hébergeant le service Internet également dynamiquement approvisionné. En cas de surcharge, la différenciation de services donne la priorité aux requêtes les plus rentables, tandis que le contrôleur d'admission prévient l'écroulement du système, le temps pour le système d'effectuer des opérations d'approvisionnement dynamique afin d'absorber le pic de charge. L'approvisionnement dynamique s'appuie par ailleurs sur un modèle de service Internet simple. La force de *Cataclysm* réside dans la coopération entre le contrôleur d'admission et l'approvisionnement dynamique au sein d'une plate-forme d'administration intégrée. Néanmoins, l'approvisionnement dynamique est similaire au partitionnement dynamique virtuel mis en œuvre dans *DDSD* puisque les différents serveurs sont en réalité déjà préinstallés sur l'ensemble des machines composant le système.

3.3.2 Garanties de performance

Pour une large majorité des travaux relatifs à l'optimisation des systèmes distribués, il est impossible d'établir des garanties sur les performances des systèmes considérés après optimisation. L'amélioration des performances obtenue par l'implantation des différentes approches est certes démontrée grâce à des séries de mesures expérimentales. En outre, ces mesures permettent de constater et d'évaluer les améliorations en comparant les performances des approches traditionnelles avec celles des approches issues de ces travaux. Toutefois, l'environnement et les conditions expérimentales de réalisation des mesures sont généralement particuliers, si bien qu'il est difficile de généraliser avec certitude les résultats obtenus. Il est ainsi impossible d'établir des garanties de performances en se fondant sur ces seuls résultats d'évaluations. Pour ne lister que quelques uns des travaux de cette catégorie, citons l'approche d'Abdelzaher et al. sur la dégradation de contenus, les travaux de Welsh et al. avec le canevas *SEDA* de construction de systèmes étagés, ou encore les plates-formes d'hébergement de services Internet *Neptune* et *DDSD* [3, 4, 124, 123, 98, 97, 16, 128].

L'approche pour la sur-réservation des ressources (*Resource Overbooking*) permet d'établir des garanties statistiques sur la disponibilité des ressources [116]. Ces informations sont obtenues par des mesures expérimentales hors-ligne sur les systèmes distribués qui doivent être hébergés et qui permettent de constituer les profils des besoins en ressources des systèmes administrés. Avec l'hypothèse que les mesures expérimentales,

et donc les profils, sont représentatives du comportement réel du système (c'est-à-dire que les systèmes ont des comportements globalement déterministes), cette approche permet de garantir de manière statistique que le système administré disposera des ressources dont il a besoin pour une moyenne de $x\%$ du temps.

Les travaux qui s'intéressent à l'isolation de performance comme les *Resource Containers*, *Cluster Reserves* ou *Sharc*, permettent de fournir des garanties sur les allocations de ressources [14, 12, 114]. Ces mécanismes de réservation des ressources de bas niveau constituent une brique élémentaire pour l'établissement de garanties de performances des systèmes distribués administrés. En effet, l'accès garanti aux ressources est une condition nécessaire pour envisager de garantir les performances des systèmes administrés. L'isolation de performance supprime ici une source d'incertitude liée au partage des ressources entre des systèmes en concurrence sur une infrastructure partagée. Ce mécanisme requiert généralement des techniques complémentaires pour garantir les performances du système administré dans la mesure où ces performances dépendent d'autres facteurs comme par exemple un niveau de charge variable, etc.

Enfin, certaines approches ciblent précisément la quantification des performances des systèmes administrés [113, 110, 25, 127, 103, 62, 43, 11, 99]. En s'appuyant sur des modélisations mathématiques des systèmes administrés, ces approches permettent d'établir avec précision les performances des systèmes et ainsi d'établir des garanties. Les modélisations mathématiques prennent diverses formes selon la nature des systèmes considérés : certaines approches s'appuient sur des modélisations sous forme de réseaux de files d'attente, certaines approches reposent sur des modélisations issues du domaine de l'automatique et de la théorie de la commande, enfin d'autres approches s'appuient sur des modélisations mathématiques de certaines caractéristiques très spécifiques des systèmes considérés. Nous étudions maintenant plus en détail ces différentes approches.

Modélisation issue de l'automatique et de la théorie de la commande. Hellerstein et al. explorent les moyens d'appliquer les techniques issues de la théorie de la commande pour la régulation des systèmes informatiques [62]. Il s'agit d'établir un modèle sous la forme "contrôles $=$ Modèle(mesures)", donnant ultimement lieu à une boucle de commande par rétroaction. La rétroaction consiste ici à agir sur le système administré à l'aide des paramètres de contrôles dont les nouvelles valeurs sont obtenues grâce au modèle. Ces valeurs sont calculées par application du modèle sur de nouvelles mesures sur le système administré. La boucle de rétroaction est modélisée sous forme d'un contrôleur avec en entrée les mesures issues du système administré, et en sortie les paramètres de contrôle du système administré. Les modèles issus de ces techniques font intervenir plusieurs types de paramètres qui doivent être finement ajustés. Certains paramètres peuvent ou doivent être déterminés lors de la phase de modélisation. D'autres paramètres doivent être calibrés une fois pour toutes par expérimentation sur

le système considéré (par exemple, lors de la phase de validation du modèle). D'autres enfin doivent être réévalués dynamiquement durant le fonctionnement du système.

Diao et al. étudient notamment la régulation d'un serveur Web Apache HTTP [43]. L'étude identifie deux paramètres clés de contrôle du serveur Web, qui sont (1) le nombre maximum de clients acceptés par le serveur (*MaxClient*), et (2) le délai de garde des connexions persistantes (*KeepAlive timeout*). Ces deux paramètres sont en effet directement reliés (1) au niveau de consommation du processeur, et (2) à la consommation mémoire. L'étude compare une double modélisation *SISO* (*Single Input and Single Output*), c'est-à-dire une modélisation séparée des deux paramètres : MaxClient $=$ $M_1(\text{CPU})$ et KeepAlive $= M_2(\text{Mémoire})$; et une modélisation *MIMO* (*Multiple Input and Multiple Output*), c'est-à-dire une modélisation combinée des deux paramètres en fonction des deux mesures : $(\text{MaxClient}, \text{KeepAlive}) = M(\text{CPU}, \text{Mémoire})$. Ces travaux montrent l'intérêt de la modélisation MIMO, capable de capturer les interactions entre les paramètres contrôlés, ce que ne fait pas la double modélisation SISO.

Modélisation par réseaux de files d'attente. Urgaonkar et al. proposent une modélisation des services Internet multi-étagés en pipeline dupliqué sur des grappes de machines [110, 111]. Il s'agit ici d'une modélisation sous forme de réseaux de files d'attente qui capturent la structure étagée en pipeline des services Internet. La modélisation repose sur des probabilités de transitions entre les différents étages lors du traitement d'une requête client. En ajustant précisément les probabilités de transition entre deux étages, le modèle obtenu permet de capturer les comportements du système liés aux traitements de différents types de requêtes. Une extension de ce modèle de référence permet en outre de capturer la duplication et la répartition de charge, le taux de parallélisme des serveurs, la classification des requêtes ou encore l'effet des caches. Les différents paramètres du modèle sont déterminés selon la méthode MVA (*Mean Value Analysis*) [96].
L'application immédiate du modèle de service Internet obtenu permet d'estimer les latences de traitement des requêtes ainsi que le débit des requêtes traitées par le service. Ces estimations sont réalisées étant donné un certain jeu de paramètres du modèles dont les probabilités de transition entre les étages du système, les degrés de duplication de chacun des étages du système ou encore le nombre d'utilisateurs et le délai moyen entre deux requêtes soumises par un même utilisateur (le *think time*). Urgaonkar et al. expliquent comment utiliser le modèle obtenu afin de guider l'approvisionnement dynamique d'un service Internet multi-étagé. Ils en font notamment une démonstration reposant sur la connaissance a priori de l'évolution de la charge de travail au cours du temps.

Arnaud et al. exploitent les travaux d'Urgaonkar et al. cités précédemment et étendent la méthode de calcul MVA afin d'optimiser les performances des services Internet multi-étagés en grappe tout en minimisant leur coût de fonctionnement [11]. Les performances considérées dans cette étude concernent la latence des traitements des requêtes clients. Le coût de fonctionnement est minimisé en réduisant au maximum le nombre de ressources mobilisées pour héberger le service. Dans cet objectif, les performances de

ces services multi-étagés en grappe sont modélisées en fonction de leur architecture, caractérisée ici par les degrés de duplication de chacun des étages composant le système, et aussi en fonction de certains paramètres de configuration des systèmes, comme par exemple, le taux de parallélisme dans les serveurs. Cette modélisation permet de déterminer l'architecture et la configuration du système qui optimisent la latence du service tout en minimisant la quantité de ressources mobilisée.

Autre modélisations mathématiques. So et al. décrivent une approche par modélisation pour concevoir et mettre en œuvre un détecteur de panne dans un système distribué pair-à-pair et exhibant des garanties de performances [99]. Les performances considérées dans cette approche concernent la latence maximale de détection de la panne d'une machine ainsi que la bande passante réseau consommée par les détecteurs. Ces deux métriques de performance s'opposent dans la mesure où tout gain selon l'une des métriques se répercute négativement sur l'autre métrique. Cela conduit naturellement à considérer un compromis, et donc deux formules d'optimisation : (1) ou bien l'on garantie la latence de détection, et le modèle permet de minimiser la bande passante réseau consommée, et (2) ou bien l'on garantie la bande passante réseau consommée et le modèle permet de minimiser la latence de détection. La modélisation du système aboutit ici à une expression analytique immédiate permettant de calculer la configuration optimale des détecteurs selon le compromis choisi. Concrètement, cela permet de calculer explicitement l'intervalle de temps optimal entre deux prises de mesures par un détecteur. Cette modélisation mathématique est totalement spécifique au système de détection de panne considéré dans cette approche.

3.3.3 Gestion des systèmes patrimoniaux

Un certain nombre d'approches nécessitent des adaptations plus ou moins lourdes des systèmes administrés. Par exemple, l'approche reposant sur le canevas *SEDA* de construction de systèmes étagés évènementiels nécessite une réingénierie majeure du système administré afin de réorienter sa construction selon le modèle étagé évènementiel [124, 123]. Cette approche est totalement inapplicable pour des systèmes patrimoniaux, dont le code source est par exemple inaccessible.

C'est également le cas de la plate-forme *Neptune* qui nécessite la coopération des systèmes administrés afin de mettre en place la classification et l'ordonnancement des requêtes [98, 97]. Toutefois, l'approche employée dans *Neptune* a été généralisée dans *Quorum* qui réimplante principalement *Neptune* en levant cette limitation et en permettant ainsi d'appliquer le principe de l'ordonnancement à des services Internet quelconques [16].

De façon analogue, les approches par modélisation sont difficilement applicable pour des systèmes dont le fonctionnement interne n'est pas ouvert [99, 43]. Toutefois, il est possible pour certaines catégories de systèmes de construire une modélisation indépendante des logiciels effectivement utilisés et reposant sur une connaissance extérieure du fonctionnement global du système. C'est ce qu'ont fait Urgaonkar et al.

en modélisant les services Internet multi-étagés en pipeline dupliqué [111, 110]. Cette modélisation s'appuie justement sur la connaissance de la structure multi-étagée et en pipeline dupliqué du système et sur le fonctionnement de ces systèmes, décrits notamment par la norme J2EE.

Enfin, un certain nombre d'approches reposent sur des mécanismes de contrôle extérieurs aux systèmes administrés. C'est généralement le cas des approches fondées sur des intergiciels tels que des proxy ou des équilibreurs de charge comme dans les approches *Chameleon*, *Middle-R* ou *DDSD* [100, 101, 92, 83, 128]. C'est également le cas des approches reposant sur des mécanismes enfouis dans le système d'exploitation, comme les réservations de ressources mises en œuvre dans *Cluster Reserves* ou *Sharc* [12, 114] ; ou encore les approches fondées sur des machines virtuelles ou des grappes virtuelles comme *OnCall* ou *Cluster-on-Demand* [89, 86].

3.3.4 Gestion des oscillations et de la stabilité

L'auto-optimisation des systèmes distribués repose sur des adaptations dynamiques des systèmes administrés en vue d'améliorer leurs performances. Chaque adaptation dynamique d'un système distribué est suivie d'une période plus ou moins longue durant laquelle le système administré est soumis à des instabilités. Ces instabilités résultent principalement des modifications des caractéristiques du système ainsi que des délais d'initialisation des systèmes (comme par exemple, le temps de chauffe d'un serveur lié au chargement des mémoires caches, etc). Une mauvaise interprétation du comportement du système durant ces périodes d'instabilité conduit à des sur-réactions ou à des réactions contraires infondées, qui pénalisent les performances du système administré.

Une grande majorité des approches pour l'optimisation des systèmes par adaptation dynamique n'étudient pas le problème de la stabilité ou des oscillations. En réalité, ces approches font l'hypothèse, parfois implicite, que deux reconfigurations sont suffisamment espacées dans le temps. Autrement dit dans ces approches, la latence d'une reconfiguration (incluant la stabilisation du système après reconfiguration) est supposée petite devant la durée moyenne entre deux reconfigurations.

La plate-forme d'hébergement *Océano* dynamiquement approvisionnée ne propose pas de solution au problème des oscillations qui peuvent survenir lors de l'approvisionnement dynamique [10]. Toutefois, les concepteurs de la plate-forme identifient le problème lié à la latence des opérations d'approvisionnement dynamique. La latence de ces opérations provient dans *Océano* des délais d'allocation et de préparation des machines dans le système. Les machines sont en effet réinstallées à partir d'images du système, d'où des latences longues de l'ordre de 5 à 10 minutes. Ce problème a guidé la conception de la plate-forme *OnCall* fondée sur des machines virtuelles permettant de réduire la latence de ces opérations à environ une minute [89]. Malgré cela, la stabilisation et les oscillations du système administré n'est pas étudié par ces deux approches.

Comme dans la plate-forme *Océano*, Moore et al. observent avec *Cluster on Demand*

des latences importantes pour les opérations d'approvisionnement dynamique [86]. Afin justement de prévenir d'éventuelles oscillations et l'écroulement du système qui pourrait en résulter, les concepteurs de la plate-forme *Cluster on Demand* introduisent un délai minimal entre deux opérations d'approvisionnement dynamique intervenant sur une même grappe virtuelle de machines. Concrètement, ce mécanisme prend la forme d'une fenêtre temporelle glissante pendant laquelle la plate-forme garantit la disponibilité des machines allouées à une grappe virtuelle. L'amplitude de cette fenêtre temporelle correspond ainsi à l'inertie du système d'allocation des ressources.

Soundararajan et al. s'intéressent aux oscillations liées à l'approvisionnement dynamique des serveurs de bases de données en grappe mis en œuvre dans *Chameleon* et décrive une proposition spécialisée originale [100, 101]. La stabilité de l'algorithme d'auto-optimisation s'appuie sur deux éléments. Elle repose d'une part sur la maîtrise des latences des opérations d'approvisionnement dynamique qui est obtenue en s'assurant le contrôle de quelques machines toujours prêtes à être rapidement réintégrées en cas de besoin. L'algorithme intègre d'autre part la connaissance des délais de stabilisation des systèmes administrés. Les opérations de retrait de serveurs de bases de données sont ainsi retardées de manière transparente durant un délai permettant de vérifier la validité des retraits avant d'être rendus effectifs.

Chase et al. évoquent le problème de la stabilité de la boucle de rétroaction pilotant l'approvisionnement dynamique de la plate-forme *Muse* [26]. Toutefois, les auteurs font ici l'hypothèse que les variations de charge interviennent sur une échelle temporelle large, de l'ordre de l'heure. Les instabilités considérées ici sont issues de la forte variabilité des mesures du système, ce qui conduit les auteurs à proposer une technique de filtrage de ces mesures fondée sur un moyenne glissante (c'est-à-dire un filtre passe-bas éliminant le bruit en hautes fréquences) et sur un nouveau filtre appelé *flop-flip* et qui favorise la stabilité du signal généré.

Enfin, dans le domaine de l'automatique et de la théorie de la commande, la stabilité d'une boucle de commande fait partie intégrante du processus d'analyse et de conception du système [62]. En effet, l'automatique modélise la rétroaction mathématiquement sous forme d'équations différentielles. Il est ainsi possible d'étudier analytiquement l'existence de régions garantissant la stabilité du système, ainsi que d'autres facteurs comme par exemple la latence maximum de stabilisation (à l'établissement d'un régime permanent). La rétroaction est mise en œuvre par un contrôleur calculant les nouvelles valeurs des paramètres de contrôle en fonction des mesures faites dans le système. La conception d'un contrôleur est une tâche complexe. Pour cela, les automaticiens ont à leur disposition plusieurs formes de contrôleurs bien étudiés et dont les conditions de stabilité, les délais de stabilisation, etc. sont maîtrisées. Par exemple, Abdelzaher et al. emploient un contrôleur PI (*Proportional-Integral controller*) afin de réguler un serveur Web Apache HTTP [74]. Le contrôleur PI garantit notamment la stabilité de la rétroaction et permet d'estimer le délai de stabilisation du système ainsi régulé.

3.3.5 Généralité et réutilisabilité de l'approche

La généralité et la réutilisabilité des différents travaux d'optimisation des systèmes distribués est très variable. Certaines approches sont très spécifiques à un système particulier donné, comme par exemple l'optimisation de certains paramètres d'un serveur Web Apache HTTP ou encore la modélisation et la conception très spécifique d'un système de détection de panne pair-à-pair optimal [43, 74, 99]. D'autres proposent des approches pour l'optimisation dans le contexte de certaines catégories particulières de systèmes telles que les services Internet, les systèmes J2EE ou encore les serveurs de bases de données dupliqués en grappe, etc [98, 97, 128, 10, 100, 101, 83, 92, 112, 115, 110, 111, 11]. Enfin, certains proposent des approches plutôt générales et qui reposent sur des mécanismes de contrôle transparents [14, 12, 114, 89, 86, 26].

Notons enfin que certaines approches aboutissent à des prototypes non réutilisables, car entièrement spécifiques aux cas d'études considérés, mais apportent une méthodologie générale ou réutilisable. C'est très exactement le cas des approches par modélisation, ou encore celui des approches définissant un cadre de conception, comme par exemple *SEDA* [3, 62, 124, 123].

3.4 Synthèse

L'optimisation des performances des systèmes distribués est un vaste domaine d'étude toujours très actif et qui laisse de nombreux problèmes ouverts. Nous avons présenté un ensemble de travaux représentatifs des différentes approches que nous avons identifiées concernant l'optimisation des systèmes distribués. Notre étude révèle que la gestion des variations de charge a été largement étudiée dans des contextes très variés, et donnant lieu à des approches diverses dont principalement la gestion de contention et l'adaptation à la charge par approvisionnement dynamique. Il existe finalement assez peu d'approches qui s'attachent à établir puis garantir les performances des systèmes optimisés. En réalité, cela doit être relié à la complexité des modélisations sur lesquelles reposent ces approches garantissant les performances des systèmes administrés. La gestion des systèmes en "boîtes noires" comme les systèmes patrimoniaux est une caractéristique assez rare parmi les différents travaux que nous avons considérés. En effet, l'optimisation d'un système repose fréquemment sur l'adaptation ou la connaissance de mécanismes internes aux systèmes administrés, ce qu'il est impossible de réaliser sur des systèmes vus comme des boîtes noires. Enfin, quelques approches reconnaissent les complications liées aux instabilités des systèmes administrés causées par leur adaptation dynamique. À l'exception des techniques issues de la théorie de la commande et du domaine de l'automatique qui intègrent directement ces considérations dans la conception de la boucle de rétroaction par modélisation, très peu d'approches apportent de solution concrète à ce problème.

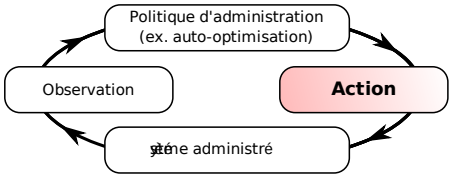
Nous proposons une approche pour l'auto-optimisation fondée sur l'approvisionnement dynamique pour la *gestion des variations dynamiques de charge*. Notre approche per-

met de traiter aussi bien les *variations graduelles* que les *pics de charge*, et s'intéresse également à la gestion de l'énergie en considérant les *sous-charges*. Notre approche intègre à la fois un gestionnaire d'auto-optimisation très général fondé sur une heuristique *générale* et *réutilisable*, ainsi qu'un gestionnaire d'auto-optimisation fondé sur la modélisation mathématique du système administré *spécifique* mais permettant de fournir des *garanties de performance*. Notre approche permet de plus l'administration et l'auto-optimisation de *systèmes patrimoniaux* vus comme des boîtes noires par une technique d'encapsulation qui fournit une vue uniforme des différents systèmes patrimoniaux administrés. Nous proposons enfin une technique fondée sur l'architecture du système administré et permettant la *prévention des oscillations* pouvant résulter de l'approvisionnement dynamique, afin de maintenir la stabilité et les performances du système auto-optimisé. Notre approche est décrite plus en détail au chapitre 7.

Nous avons présenté, dans ce chapitre, les travaux d'optimisation des systèmes distribués. L'auto-optimisation des systèmes distribués implique diverses adaptations dynamiques des systèmes administrés qui prennent la forme d'actions sur ces systèmes distribués administrés. Dans le chapitre suivant, nous étudions les travaux relatifs aux actions sur les systèmes distribués.

Chapitre 4

Actions sur le système administré



Nous étudions, dans ce chapitre, l'état de l'art relatif aux actions sur les systèmes administrés dans le cadre de l'administration fondée sur l'architecture. Dans un premier temps, nous identifions et présentons les défis scientifiques relatifs aux actions sur les systèmes administrés. Nous étudions ensuite, défi par défi, les travaux apparentés d'actions sur les systèmes administrés. Enfin, nous présentons une synthèse critique sur les travaux apparentés.

4.1 Défis scientifiques d'actions sur les systèmes administrés

Cette section présente les défis scientifiques que constituent les actions sur les systèmes distribués administrés. Nous nous intéressons aux moyens d'agir sur les systèmes administrés afin de les adapter dynamiquement au cours de leur fonctionnement.

Description d'architectures paramétrables. Le paramétrage des architectures est une faculté essentielle pour représenter des architectures dynamiques, adaptables et réutilisables. Cela est particulièrement important lorsqu'il s'agit de décrire l'architecture d'un système dont certaines caractéristiques ne sont pas connues avant le déploiement et l'instanciation du système. Par exemple, la répartition exacte des composants formant un système distribué sur un ensemble de machines n'est bien souvent pas connu avant l'instanciation effective du système. Il est donc important de pouvoir abstraire et paramétrer la description du système.

Contrôle du déploiement. L'algorithme de déploiement employé pour exécuter une action distribuée ou instancier une description d'architecture est-il paramétrable, ajustable ou spécialisable ? La flexibilité de l'algorithme de déploiement est essentielle, car elle permet d'intégrer des contraintes de distribution, de coordination et de synchronisation au processus de déploiement. Ces contraintes sont parfois nécessaires pour maintenir la cohérence du système.

Dynamisme de l'architecture. Les systèmes autonomes réagissent et évoluent par le jeu d'adaptations dynamiques dirigées sur leur architecture. Les systèmes autonomes reposent ainsi sur des architectures dynamiques. Il est donc souhaitable de disposer de descriptions d'architecture capables d'intégrer l'expression d'adaptations dynamiques. Ces descriptions doivent par ailleurs être suffisamment expressives pour autoriser des adaptations dynamiques arbitraires.

Généralité de l'approche. La généralité d'une approche pour la mise en œuvre d'actions sur les systèmes administrés est un critère important. Elle détermine le degré de réutilisabilité de l'approche dans des contextes différents.

4.2 Travaux apparentés d'actions sur les systèmes administrés

Nous détaillons dans cette section, défi par défi, les travaux apparentés d'actions sur les systèmes administrés.

4.2.1 Description d'architectures paramétrables

Nous identifions deux manières de décrire l'architecture d'un système distribué : ou bien nous décrivons explicitement l'architecture cible en listant extensivement sa composition, ou bien nous la décrivons implicitement et par intension, en énumérant certaines de ses caractéristiques. Les descriptions extensives autorisent un paramétrage architectural (par exemple, le type ou le nombre de sous-composants). Les descriptions intensives relèvent naturellement d'un plus haut niveau de description en permettant d'intégrer dans la description des contraintes ou paramètres architecturaux ou comportementaux, comme par exemple une description de niveau de qualité de service.

Description par extension. Un grand nombre de travaux étudiant les actions sur les systèmes distribués fondés sur des architectures dynamiques s'appuient sur des descriptions par extension des actions à mettre en œuvre sur l'architecture cible [2, 81, 6, 95, 23, 27, 46, 9, 68, 56].

Abdellatif et al. proposent par exemple une approche reposant sur le langage de description d'architecture (ADL) Fractal décrivent explicitement l'architecture cible par extension en listant tous les éléments d'architectures et leurs relations [2]. Une description ADL Fractal prend la forme d'une description XML listant les éléments constituant

l'architecture. Par exemple, l'ADL Fractal d'un composant composite HelloWorld formé d'un sous-composant client lié à un sous-composant server s'exprime comme illustré ci-dessous. La description d'un composant énumère ses interfaces (balises interface), son contenu (ou bien des sous-composants grâce aux balises component, ou bien la classe Java d'implantation grâce à la balise content) et enfin un ensemble de liaisons (balises binding).

```
<definition name="HelloWorld">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <component name="client">
        <interface name="r" role="server" signature="java.lang.Runnable"/>
        <interface name="s" role="client" signature="Service"/>
        <content class="ClientImpl"/>
    </component>
    <component name="server">
        <interface name="s" role="server" signature="Service"/>
        <content class="ServerImpl"/>
    </component>
    <binding client="this.r" server="client.r"/>
    <binding client="client.s" server="server.s"/>
</definition>
```

L'ADL Fractal intègre en outre des capacités de paramétrage architectural. Toutefois, le niveau de paramétrage autorisé est très faible et se limite à des paramètres très simples (de type macros). L'exemple ci-dessous montre une description de composant paramétrée par trois arguments impl, header et enfin count. L'argument impl permet de paramétrer le nom de la classe Java d'implantation du composant (utilisé dans la balise content). De façon analogue, les arguments header et count paramètrent les valeurs des attributs de même nom du composant (balises attribute).

```
<definition name="GenericServerImpl" arguments="impl,header,count">
    <interface name="s" role="server" signature="Service"/>
    <content class="${impl}"/>
    <attributes signature="ServiceAttributes">
        <attribute name="header" value="${header}"/>
        <attribute name="count" value="${count}"/>
    </attributes>
    <controller desc="primitive"/>
</definition>
```

D'autres approches reposent sur des langages ADL aux propriétés similaires à celle que nous venons de présenter. Il s'agit de descriptions ADL extensives exploitées dans des contextes variés et exhibant des spécificités liées à leur contexte.

Quema et al. proposent un ADL pour décrire des systèmes à composants asynchrones hiérarchiques fondés sur la plate-forme ScalAgent [95].

Mikic-Rakic et al. exploitent un ADL extensif pour décrire et déployer des systèmes distribués en environnements embarqués contraints (comme des assistants personnels PDA) [81]. Cet ADL est fondé sur la plate-forme de conception et d'exécution *Prism* [76, 82].

Caromel et al. présentent un ADL pour la plate-forme *ProActive*, une implantation du modèle de composant Fractal sous forme d'objets asynchrones [23]. Cette approche se focalise sur l'abstraction des descriptions ADL par rapport à l'environnement effectif

d'exécution. En particulier, l'ADL intègre une notion de nœud virtuel et permet de spécialiser le mécanisme d'allocation des nœuds physiques.

Flissi et al. implantent une plate-forme d'exécution de tâches modélisées sous forme de composants Fractal [46]. L'approche repose sur une description ADL Fractal traditionnelle, donc extensive, des tâches et de leurs relations. La modélisation sous forme de composants capture et homogénéise la diversité des tâches devant être exécutées (exécution de scripts ou code Java, invocation d'applications ou de systèmes tiers, etc.).

Akkerman et al. définissent un ADL spécialisé (CDL, *Component Description Language*) dans le cadre d'une infrastructure reposant sur l'architecture J2EE qui consiste ici en une grappe de serveurs JBoss [6]. Les composants modélisés ici exposent leurs dépendances, afin de guider un algorithme de déploiement également ad hoc.

Chazalet et al. présentent un ADL spécifique pour les systèmes construits sur un ensemble de services interdépendants [27]. Cette approche repose sur un service de déploiement ad hoc qui déduit l'ordre d'exécution des actions de déploiement des dépendances entre services, de façon similaire aux usines ADL Fractal.

Enfin, Goldsack et al. proposent la plate-forme de déploiement distribué et de gestion de cycle de vie *SmartFrog* [9, 56]. La plate-forme *SmartFrog* associe un modèle de composant particulier et un langage de description ADL extensif. L'ADL *SmartFrog* est un langage de description spécialisé qui permet basiquement de décrire une structure à composants hiérarchique (à la manière des descriptions ADL Fractal en XML). L'ADL offre un niveau de paramétrage simple reposant sur des références qui sont résolues lors de l'instanciation ou de l'exécution de la structure à composants. Nous procédons en section 11.2 (page 158) à une évaluation plus poussée des fonctionnalités et des limitations de la plate-forme *SmartFrog*, et notamment des capacités de paramétrage de son ADL.

Description par intension. À l'inverse, certains travaux proposent des descriptions par intension de l'architecture cible. Nous discernons deux catégories de travaux suivant cette approche : (1) les travaux qui reposent sur une description par contraintes, associée à un moteur de résolution de contraintes [15, 80, 109] ; et (2) les travaux qui reposent sur un planificateur IA [71, 13].

Résolution de contraintes. Tibermacine et al. proposent des descriptions d'architecture qui combinent des contraintes architecturales et des contraintes de configuration (par exemple, des contraintes sur les ressources ou sur la localisation et la distribution de l'architecture) [109]. L'originalité de cette approche est d'intégrer ces contraintes au système à l'exécution, permettant ainsi de maintenir les décisions et caractéristiques de l'architecture malgré les évolutions du système et de son environnement (comme par exemple la perte de machines). Les contraintes sont exprimées dans un langage ACL (*Architecture Constraint Language*) sous forme d'invariants (inv) faisant intervenir des prédicats de la logique du premier ordre, que nous illustrons ci-dessous.

Contrainte architecturale : la contrainte suivante interdit l'établissement d'une liaison entre deux composants Input_Flow.

```
# This constraint states that for all link instances between architecture instances,
# there should be no link which binds two components which are identified by Input Flow.
context ClientServer : ComponentInstance inv :
ClientServer.subArchitecture.archInstance.linkInstance->select(link | link.endPoint->forAll(p1,p2 |
    p1.anchorOnInterface.componentInstance.id = 'Input_Flow'
    and p2.anchorOnInterface.componentInstance.id <> 'Input_Flow' ))
```

Contraintes sur les ressources : les différentes contraintes qui suivent requièrent 512Mo de mémoire disponible, au moins un processeur dont la vitesse excède 1GHz et enfin une connexion réseau dont la bande passante excède 512Kbits/s.

```
# 1. Free memory >= 512 MB:
context Serv_SVG : Component inv :
Serv_SVG.resource->oclAsType(Memory).free >= 512
# 2. At least one CPU > 1 GHz:
context Serv_SVG : Component inv :
Serv_SVG.resource->oclAsType(CPU).processors->select(cpu : CPU_Model | cpu.speed>1000)->size() >= 1
# 3. Network interface bandwidth >= 512 Kb/s:
context Serv SVG : Component inv :
Serv_SVG.resource->oclAsType(NetworkInterface).tx >= 512
```

Contrainte de distribution et localisation : pour des raisons de fiabilité (redondance), la contrainte suivante exige que chaque instance de composant Serv_SVG soit placée sur une machine distincte.

```
# for reliability reasons (redundancy at the server side),
# all Serv_SVG components have to be located on distinct hosts
context ClientServer : CompositeComponent inv :
ClientServer.subComponent->select(c1,c2 : Component |
    c1.name='Serv_SVG' and c2.name='Serv_SVG' and c1.location.id <> c2.location.id)
```

Dans cette approche, les contraintes sont rendues actives au sens où leur validité et leur cohérence sont maintenues par adaptation dynamique lorsque l'environnement du système évolue. En pratique, seuls certains types de contraintes sont réifiés à l'exécution et sont ainsi rendus actifs. Concrètement, les contraintes sont traduites et mises en œuvre à l'aide de sondes qui permettent au système d'être informé des changements de l'environnement d'exécution. Tout changement détecté déclenche alors une réévaluation du déploiement et de ses contraintes. Ce travail s'appuie sur la bibliothèque Java d'optimisation et de résolution de contraintes *Cream* afin de traiter les contraintes dynamiques [105] ; il s'appuie également sur la plate-forme de surveillance *Draje* (*Distributed Resource-Aware Java Environment*) permettant de découvrir et de monitorer de façon homogène les diverses ressources physiques et systèmes intervenant dans les contraintes [58].

Bastarrica et al. présentent le modèle de calcul *BIP* (*Binary Integer Programming model*) qui permet de modéliser les systèmes distribués et permet également d'exprimer des contraintes sur ces systèmes [15]. Le modèle et ses contraintes sont ensuite exploités par un algorithme de résolution (de type *branch & bound*) qui permet de générer des instances de solution du problème ainsi modélisé. L'étude démontre l'utilisation de contraintes sur la structure d'un système distribué par rapport aux ressources du système, et considère spécifiquement des contraintes sur la consommation mémoire et la consommation de bande passante réseau.

Mikic-Rakic et al. intègrent la résolution de contraintes au cycle de vie du système administré [80]. Ce travail propose l'algorithme *Avala* qui adapte dynamiquement la structure du système dans l'objectif d'améliorer sa disponibilité en cas de perte de connectivité de certains de ses nœuds, en fonction d'observations réalisées à l'exécution sur le système. Ce travail s'appuie sur une modélisation très spécifique des systèmes distribués, et sur l'algorithme *Avala* également spécifique à la résolution du problème de la disponibilité.

Notons toutefois que les approches faisant intervenir des moteurs de résolution de contraintes souffrent de coûts d'exploitation relativement élevés. En effet, la complexité des algorithmes mis en jeu pour la résolution des contraintes est élevée (par exemple, les algorithmes de type *branch and bound*), si bien que ces approches ne sont pas envisageables sur des systèmes de grande taille. Par exemple, en dépit des efforts d'optimisation particulièrement poussés entrepris par Mikic-Rakic et al. sur l'algorithme *Avala* présenté à l'instant, la complexité de l'algorithme obtenu reste exponentielle, ce qui rend son exploitation relativement limitée pour des systèmes de grande taille.

Planification IA. Kichkaylo et al. propose un planificateur qui cible spécifiquement le problème du placement d'ensembles disjoints de composants sur un ensemble de ressources [71]. Le planificateur s'appuie sur une description qui intègre des indications de qualité de service (concernant ici les bandes passantes réseau et les niveaux d'utilisation du processeur) guidant la planification dans l'élaboration d'un placement optimal des composants sur les ressources.

Arshad et al. implantent la plate-forme de (re-)déploiement *Planit* qui repose sur un planificateur (ici le planificateur LPG, *Local search on Planning Graph*) [13]. *Planit* reçoit une description de l'état cible du système que l'utilisateur souhaite déployer. L'état courant du système est par ailleurs activement surveillé par diverses sondes, ce qui permet au planificateur d'en conserver une connaissance précise et actualisée. À partir de l'état courant et l'état cible du système, le planificateur génère des plans de tâches permettant d'atteindre l'état cible. Ces différents plans de tâches peuvent être évalués selon différents critères afin d'obtenir un plan optimal. Le critère utilisé ici est le nombre d'étapes composant le plan. Enfin, le plan retenu est exécuté. Lorsqu'un problème survient, celui-ci est détecté grâce à la surveillance opérée sur le système par un gestionnaire de problèmes. Ce dernier évalue l'ampleur du problème et reconstruit l'état courant du système, à partir duquel le planificateur établit un nouveau plan de tâches destiné à ramener le système dans son état cible.

La communauté spécialisée dans l'étude de la planification IA a standardisé un langage de description. Il s'agit du langage PDDL qui permet de décrire l'état initial ainsi que l'état cible à atteindre [72]. Cette description s'inscrit dans un cadre plus large définissant un domaine décrivant les objets manipulés ainsi que leurs relations, et les différents actions qui peuvent être réalisées. Le descripteur d'une action indique ses paramètres, sa durée ainsi que des pré- et post-conditions qui rendent la planification possible. Le planificateur IA génère à partir de ces descriptions un ensemble d'actions permettant d'atteindre l'état cible. Les actions peuvent ensuite être traduites et exécu-

tées, par exemple, sous forme d'une séquence de commandes shell. L'extrait de description PDDL ci-dessous illustre l'expressivité des approches reposant sur des planificateurs IA. Cet extrait inclut un fragment de la description de l'état initial, la description de l'état cible visé associé à un critère d'optimisation. L'extrait présente également la description d'une action.

```
;; initial state description
(:init
    ...
    ;; machine description (currently running)
    (machine-working leone)
    (application-available sms leone)
    (application-available rubbos leone)
    (application-available webcal leone)
    (apache-installation-available apache leone leone)
    (tomcat-installation-available tomcat leone leone)

    ;; running apache server (hosted on the machine named leone)
    (apache-working apache leone leone)
    (apache-has-configuration-file apache leone httpd)
    (apache-has-module-installation apache leone jsp leone)
    (apache-module-provides-service apache leone jsp jsp service)
    (apache-has-installed-module apache leone ssl leone)
    (apache-providing-service apache leone leone ssl service)
    (connector-available-in-apache mod jk apache leone leone)
    (connector-available-in-apache mod proxy apache leone leone)
    (connector-available-in-apache mod jk2 apache leone leone)
    ...
    )

;; target state description
(:goal (and (application-ready-1 rubbos)))

;; optimization metric
(:metric minimize (time-to-start-apache))

;; actions description
...
(:durative-action start-apache
    :parameters (?ma - machine ?ap - apache ?app - application)
    :duration
        (= ?duration (time-to-start-apache ?ap))
    :condition
    (and
        (at start (apache-configured ?ap ?ma))
        (at start (not (apache-working ?ap ?ma)))
        (at start (machine-working ?ma))
        (at start (application-available-in-apache ?app ?ap ?ma))
        (at start (> (- (max-machine-load ?ma) (current-machine-load ?ma))
                (apache-load ?ap) ))
        )
    :effect
    (and
        (at end (apache-working ?ap ?ma))
        (at end (application-ready-in-apache ?app ?ap ?ma))
        (at start (increase (current-machine-load ?ma) (apache-load ?ap)))
        )
    )
```

4.2.2 Contrôle du déploiement

Contrôle implicite et limité des déploiements. Une grande majorité des travaux présentés précédemment repose sur des algorithmes de déploiement prédéterminés et peu flexibles [2, 81, 6, 95, 23, 27, 46, 109, 15, 80, 71, 13]. En effet, dans ces approches, les algorithmes de déploiement font partie de la plate-forme de déploiement et sont externes aux descriptions d'architectures. Les descriptions d'architectures peuvent alors contenir des annotations destinées à l'algorithme de déploiement. Ces approches offrent un contrôle limité sur la distribution, la coordination ou la synchronisation des tâches générées et exécutées.

Plusieurs approches reposent sur un algorithme de déploiement entièrement piloté par l'architecture du système à instancier et sans offrir aucune capacité de contrôle de la distribution, de synchronisation ou de coordination sur le processus de déploiement. L'approche suivie par Quema et al. repose sur un algorithme de déploiement hiérarchique ad hoc directement piloté par la description hiérarchique ADL des systèmes à déployer [95]. Akkerman et al. proposent un algorithme spécialisé dans le déploiement d'infrastructures reposant sur l'architecture J2EE [6]. L'approche modélise sous forme de composants les éléments de l'architecture J2EE et exposent ainsi les dépendances entre ces éléments. Cette connaissance guide l'algorithme de déploiement de façon analogue à ce que l'on trouve dans les usines ADL Fractal. De façon similaire, Chazalet et al. proposent un algorithme de déploiement de systèmes distribués construits comme un assemblage de services interdépendants [27]. Cette approche prend la forme d'un service de déploiement ad hoc qui infère depuis la description ADL l'ordre d'exécution des actions de déploiement de service.

Abdellatif et al. s'appuient sur l'algorithme de déploiement codé dans les usines ADL Fractal pour générer et exécuter les plans de déploiements qui correspondent aux descriptions d'architectures qui doivent être instanciées [2]. Les usines ADL Fractal sont effectivement les entités chargées d'instancier les descriptions ADL Fractal [21]. Une usine ADL Fractal analyse un descripteur d'architecture Fractal XML, le traduit en un ensemble de tâches (création de composant, ajout d'un sous-composant, création d'une liaison entre deux interfaces, etc) reliées entre elles par des dépendances (par exemple, la création de deux composants doit intervenir avant l'établissement d'une liaison entre ces composants). Enfin, l'usine ADL ordonne puis exécute ces tâches selon un schéma prédéterminé qui respecte les dépendances générées. La génération, l'ordonnancement et l'exécution des différentes tâches font partie du fonctionnement des usines ADL et n'offrent a priori aucun moyen de contrôle ou de paramétrage. Pour en contrôler plus précisément le fonctionnement, il faudrait étudier comment modifier l'ADL Fractal pour y inclure des informations relatives aux tâches générées et modifier les usines ADL pour prendre en compte ces informations.

La plate-forme distribuée d'exécution de tâches implantée par Flissi et al. repose justement sur l'algorithme de déploiement codé dans les usines ADL Fractal [46]. Les tâches y sont représentées sous forme de composants Fractal. Le modèle d'exécution des tâches repose alors (1) sur les dépendances entre les tâches, modélisées par les liaisons entre composants ; et (2) sur un cycle de vie spécifique des composants (deploy,

start, stop, undeploy). En pratique dans cette approche, il faut considérer les tâches-composants comme les instructions d'un bytecode dont la machine virtuelle d'exécution serait l'usine ADL Fractal. L'exécution des tâches est ainsi ordonnée par les usines ADL Fractal. Cette approche souffre donc des limitations des usines ADL Fractal, et n'intègre donc pas de mécanisme pour coordonner, synchroniser ou organiser la distribution des tâches.

Mikic-Rakic et al. proposent deux algorithmes de déploiement distribué pour la plate-forme *Prism* [81]. Le premier est construit sur un contrôleur central qui dirige les opérations de déploiement. Le second est construit sur un ensemble de contrôleurs distribués sur chaque machine composant le système distribué ; le déploiement est ainsi distribué et parallélisé, et chaque contrôleur réalise une partie du déploiement localement sur une machine. L'approche offre ainsi deux alternatives prédéfinies pour la distribution du processus de déploiement en choisissant l'un ou l'autre des deux algorithmes de déploiements. Toutefois, aucun de ces algorithmes n'intègre de mécanisme de synchronisation ou de coordination.

L'algorithme de déploiement mis en œuvre par Caromel et al. pour la plate-forme *ProActive* offre un paramétrage du mécanisme d'allocation des machines physiques sur lesquelles a lieu le déploiement [23]. L'approche est extensible et propose d'origine plusieurs allocateurs de machines reposant sur des systèmes existants (SSH, OAR, PBS, etc) dont notamment une technique originale d'allocation pair-à-pair sur une grille de machines. Cette approche s'est focalisée précisément sur l'abstraction de l'environnement d'exécution effectif, mais souffre par ailleurs des mêmes limitations que les approches fondées sur les usines ADL Fractal.

Enfin, les travaux qui s'appuient sur des descriptions par intension des architectures à instancier reposent également sur des algorithmes externes pour réaliser le déploiement (par exemple un moteur de résolution de contraintes ou un planificateur IA) et n'offrent pour l'instant pas de moyen de contrôler ces déploiements [109, 15, 80, 71, 13]. Cela est d'autant plus complexe que l'on ne connait généralement pas à l'avance la forme de l'architecture résultant du processus de résolution de contraintes ou de planification IA. L'expression de structures de contrôle sur ces déploiements guidés par intension constituent encore à ce jour un défi non résolu.

Gestion explicite des synchronisations et coordinations. Quelques travaux intègrent des structures de contrôle explicites pour la distribution, la coordination ou la synchronisation d'actions [9, 56, 68, 118, 67].

Goldsack et al. décrivent la plate-forme de déploiement et de gestion du cycle de vie *SmartFrog* déjà présentée précédemment et qui intègre la bibliothèque de composants *SmartFlow* permettant de distribuer, synchroniser et coordonner des ensembles de tâches [9, 56]. Le modèle d'exécution des tâches est similaire à celui utilisé dans l'approche de Flissi et al. [46]. Les actions sont représentées sous forme de composants, et le modèle d'exécution repose sur le cycle de vie en trois étapes des composants *SmartFrog* (*deploy*, *start* et *terminate*). La création (respectivement le déploiement, le démarrage ou la terminaison) d'un composant implique la création (respectivement le déploiement, le

démarrage ou la terminaison) de ses sous-composants. Le démarrage d'un composant correspond au démarrage de la tâche qu'il représente. La terminaison de l'exécution d'une tâche déclenche la terminaison du composant qui la représente, ce qui permet d'établir des synchronisations sur la terminaison des tâches. La bibliothèque *SmartFlow* propose un ensemble de composants dont l'objectif est de coordonner et de synchroniser les tâches représentées par leurs sous-composants. Par exemple, un composant Parallel permet d'exécuter en parallèle les tâches décrites en tant que sous-composants du composant Parallel. L'exécution se synchronise naturellement sur la terminaison de l'ensemble des sous-composants. La bibliothèque de composants *SmartFlow* propose un ensemble assez complet et extensible de composants de synchronisation et de coordination. Il s'agit principalement des structures de *workflow* usuelles comme celles identifiées par Van der Aalst et al. [120]. Enfin, chaque composant peut être annoté par un attribut sfProcessHost indiquant sur quelle machine il doit être déployé et exécuté. Cela permet de contrôler précisément la distribution de chacune des tâches sur un ensemble de machines. En résumé, la plate-forme *SmartFrog* combinée à sa bibliothèque *SmartFlow* autorisent la description d'un ensemble de tâches distribuées, en permettant le contrôle précis de la distribution, de la coordination et des synchronisations entre ces tâches. Cela vaut pour des tâches représentées sous forme de composants *SmartFrog* et respectant le cycle de vie d'un composant correspondant à une tâche. Mais cela exclut toutefois le contrôle du déploiement d'un système à composants *SmartFrog*, car la tâche correspondant à ce déploiement n'est pas représentée sous forme d'un composant *SmartFrog*. Nous donnons plus de détails sur l'intérêt et les limitations de cette bibliothèque dans notre évaluation comparative de la plate-forme *SmartFrog* à la section 11.2 (page 158).

Keller et al. décrivent l'intégration du langage *BPEL* (*Business Process Language Execution*) à la plate-forme d'administration *IBM Tivoli* [68]. Cette intégration a lieu dans le cadre de la mise un œuvre du système d'administration *CHAMPS* (*Change Management with Planning and Scheduling*) spécialisé pour les services Web. *BPEL* est un langage dérivé de *XML* et spécialisé dans la coordination et la synchronisation d'actions. *CHAMPS* repose ainsi (1) sur le moteur de déploiement *TIO* (*Tivoli Intelligent Orchestrator*) et (2) sur le moteur de coordination *BPWS4J* (Web Services *BPEL* for Java) permettant d'exécuter des scripts *BPEL* faisant intervenir des services Web pour piloter les déploiements. Cette approche s'appuie également sur l'intégration et la couture des langages *BPEL* et Java permettant notamment l'intégration d'extraits de code Java et d'invocations de services Web depuis les descripteurs *BPEL*. *BPEL* intègre des constructions de coordination et de synchronisation équivalentes à celles offertes dans *SmartFrog*. L'objet des scripts *BPEL* est précisément de contrôler les synchronisations et les coordinations d'actions, à l'instar de la bibliothèque *SmartFlow*. Néanmoins, le contrôle de la distribution des actions est implicite et repose sur le découpage des actions en différents scripts distribués manuellement.

Enfin, Valetto et al. développent *Workflakes*, un environnement d'exécution pour les adaptations des systèmes administrés au sein de la plate-forme autonome *Kinesthetics eXtreme* [119, 117, 118, 67]. *Workflakes* est construit comme une extension de la plate-

forme à agents mobiles *Cougaar* [36]. *Workflakes* repose (1) sur une plate-forme à agents mobiles qui exécutent des programmes (*worklets*) réalisant des adaptations locales sur les systèmes administrés, et (2) sur un moteur de coordination des agents qui permet de contrôler les exécutions des différentes adaptations. *Workflakes* maintient ainsi une séparation franche entre les programmes d'adaptation des systèmes et les programmes servant à la coordination de ces adaptations, suivant ainsi le principe de séparation des préoccupations. L'exécution d'un (*worklet*) est contrôlée en y associant des informations de coordination (*jackets*), qui prennent la forme de priorités, de pré- ou post-conditions, de répétitions, etc. Le contrôle de la distribution dans cette approche est de même nature que celui obtenu dans l'approche précédente avec *BPEL*. La distribution d'un ensemble de tâches est obtenue en concevant ces tâches dans des *worklets* séparés. Enfin, les *jackets* introduisent des capacités de synchronisation, mais qui sont toutefois limitées en comparaison aux deux approches précédentes.

4.2.3 Dynamisme de l'architecture

Classification. Bradbury et al. ont récemment conduit une étude sur les différentes techniques de description de systèmes à architectures (ADL) dynamiques [20]. Cette étude révèle que le domaine des systèmes distribués fondés sur des descriptions d'architectures dynamiques est largement étudié. L'étude se focalise spécifiquement sur l'expressivité des ADL dynamiques et identifie trois niveaux d'expressivité du dynamisme. Nous distinguons avant tout, en dehors de cette classification, les approches statiques qui ne permettent pas de décrire le dynamisme du système. (1) Au plus faible niveau d'expressivité, l'ADL dynamique permet simplement d'exprimer des adaptations statiquement prédéterminées avant l'exécution du système. (2) Avec une expressivité intermédiaire, l'ADL dynamique permet de choisir à l'exécution parmi un ensemble statiquement prédéterminé d'adaptations. Pour ces deux premiers niveaux d'expressivité, les descriptions d'architectures spécifient entièrement les différentes possibilités d'adaptations dynamiques. Enfin, (3) l'étude conclut sur l'incapacité des différentes approches considérées à permettre l'expression d'évolutions arbitraires et non contraintes, qui représentent selon les termes de cette étude le plus haut degré d'expressivité. Concrètement, cela signifie qu'aucun des systèmes considérés dans cette étude n'est capable d'intégrer durant son fonctionnement des modifications qui n'ont pas été entièrement spécifiées dans leur description initiale. Cette constatation peut être reliée au fait qu'aucune des approches considérées n'intègre de capacité d'expression d'ordre supérieur.

Approches statiques. Un certain nombre des approches que nous avons citées précédemment n'intègre aucune capacité d'expression du dynamisme des systèmes dans les descriptions ADL [95, 6, 81, 23, 46, 2]. Ces approches reposent en effet sur des descriptions ADL statiques. Ces descriptions correspondent alors tout simplement à l'architecture initiale pure des systèmes à déployer.

En revanche, certaines approches reposent également sur des descriptions ADL statiques, mais permettent toutefois de mettre en œuvre des réactions dynamiques. C'est par exemple le cas de la plate-forme de déploiement *SmartFrog* [56]. L'ADL *SmartFrog* est effectivement statique et ne permet pas d'exprimer le dynamisme des systèmes dans leurs descriptions. Toutefois, en combinant cet ADL statique avec le code d'implantation Java des composants *SmartFrog*, il est alors possible de commander dynamiquement des déploiements, depuis le code d'implantation Java, au moteur de déploiement de la plate-forme *SmartFrog*.

Dynamisme spécifié. Certaines des approches permettent d'exprimer directement dans les descriptions ADL les adaptations dynamiques du système. Dans ces approches, les adaptations possibles des systèmes administrés sont entièrement spécifiées dans les descriptions.

Safran. Pierre-Charles David propose notamment dans sa thèse le canevas *Safran* spécialisé dans la construction de systèmes adaptatifs [39]. Ce canevas associe et étend le modèle de composant Fractal avec le langage *FPath* d'interrogation des systèmes à composants, le langage de script *FScript* spécialisé dans les reconfigurations des systèmes à composants et enfin avec le système de surveillance *WildCAT* [40]. Un système fondé sur le canevas *Safran* embarque une politique composée d'un ensemble de règles. Une règle *Safran* correspond à une réaction à un évènement, dont la description prend la forme suivante :

```
rule <rule-name> { when <event> if <cond> do <action> };
```

Les évènements considérés dans *Safran* sont liés ou bien à des modifications de l'architecture Fractal (par exemple, l'ajout ou le retrait d'un sous-composant ou d'une liaison, etc) ou bien à des changements dans l'environnement d'exécution, rapportés par le service de surveillance *WildCAT*. Par exemple, l'évènement *WildCAT* suivant est déclenché lorsque la mémoire disponible tombe en dessous de 10 millions d'octets :

```
realized(sys://storage/memory#free < 10_000_000)
```

Enfin, l'action associée à une règle *Safran* repose sur un script *FScript* permettant d'agir sur la structure à composants Fractal. Les scripts *FScript* font un usage intensif du langage d'interrogation *FPath* pour naviguer et désigner des éléments de l'architecture que l'on souhaite manipuler. Par exemple, le script *FScript* suivant permet modifier la stratégie de cache d'un système en ajustant la liaison du composant cache vers un nouveau composant strat implantant la stratégie voulue.

```
action select-strategy(cache, strat) = {
    if ($cache/binding::strategy != $strat) then {
        itf := $cache/interface::strategy; /* select the 'strategy' interface of component 'cache' */
        if (bound($itf)) then {
            /* unbind and disable the former cache strategy component */
            previous := $cache/binding::strategy;
            unbind($itf);
            stop($previous);
```

```
            }
            /* bind and start the new cache strategy */
            bind($itf, $strat/interface::strategy);
            start($strat);
        }
}
```

Plastik. Joolia et al. proposent *Plastik*, un canevas de construction de systèmes à architectures dynamiques [66]. *Plastik* repose sur une extension du langage ADL de description d'architecture *ACME/Armani* [54, 85]. Les architectures *Plastik* sont mises en œuvre sur le modèle de composant et son infrastructure d'exécution *OpenCOM* [37]. *ACME* est un ADL très général qui permet grâce à son extension *Armani* l'expression de règles architecturales comme des contraintes ou des invariants, mais n'autorise pas la description de reconfigurations dynamiques de l'architecture. Cet ADL a été étendu dans le cadre du canevas *Plastik* pour permettre la description de telles reconfigurations architecturales sous la forme de règles ECA (Évènement-Condition-Action) du type : on (prédicat) do (action). Dans cet objectif, *Plastik* maintient une connexion causale partielle durant l'exécution entre l'architecture représentant la structure du système et le système réel en cours d'exécution. En particulier, les modifications réalisées au niveau de l'ADL sont systématiquement répercutées sur le système à l'exécution ; en revanche, l'inverse n'est pas nécessairement vrai, ce qui peut introduire des incohérences entre la représentation architecturale du système et son état réel à l'exécution. *Plastik* identifie en effet trois niveaux de description des reconfigurations dynamiques : (1) les reconfigurations de niveau architectural, qui s'expriment directement au niveau de l'ADL, (2) les scripts de reconfiguration qui s'expriment comme des séquences d'opérations *OpenCOM* et (3) les reconfigurations au niveau d'*OpenCOM*, qui s'expriment directement par des appels spécifiques aux procédures de l'interface de programmation *OpenCOM*. Les scripts de reconfigurations sont exprimés dans le langage de script *Lua*. Les reconfigurations de niveau architectural sont exprimées selon l'extension d'*ACME/Armani* intégrée à *Plastik*, et sont compilés en scripts *Lua* avant d'être exécutés.

Dynamisme arbitraire. Enfin, parmi l'ensemble des travaux que nous avons étudiés, aucune des approches proposées n'offre la capacité d'exprimer des évolutions libres et non contraintes. Plus précisément, toutes les approches que nous avons étudiées et qui intègrent des capacités d'évolutions dynamiques des systèmes sont limitées parce que les reconfigurations qu'elles autorisent sont nécessairement entièrement décrites et spécifiées dans les descriptions initiales des systèmes administrés. Autrement dit, les plates-formes implantées par ces approches sont incapables d'accueillir des systèmes déclenchant des évolutions arbitraires et non initialement prévues, par exemple sous la forme d'un *plug-in*.

Paluska et al. présentent une approche permettant justement de concevoir et spécifier des systèmes capables d'intégrer des adaptations non initialement prévues [91]. Cette approche repose sur les concepts de buts et de techniques contribuant à un but

(*Goals* et *Techniques*). Cette approche s'appuie sur un planificateur chargé de réaliser un ensemble de buts en déterminant les techniques appropriées. Chaque but y est décrit comme un problème de décision ouvert reposant sur un ensemble de techniques ou de sous-buts. L'originalité de cette approche est qu'elle autorise l'intégration dynamique de nouvelles techniques qui peuvent remettre en cause les choix précédents du planificateur.

4.2.4 Généralité de l'approche

Certaines des approches sont spécifiques à des domaines d'application bien précis comme par exemple les services Internet J2EE, les services Web ou encore les systèmes embarqués en environnements contraints Web [2, 95, 6, 23, 27, 68, 81].

Certains proposent également des approches qui répondent à des problèmes spécifiques, comme par exemple la description et l'exécution de tâches réparties [46, 68] ; ou encore l'expression et la mise en œuvre de contraintes architecturales sur des systèmes bien particuliers [109, 15, 80, 71, 13]. La transposition ou la généralisation des principes de ces approches à d'autre contextes semble réutilisable, mais reste tout de même à étudier.

Enfin, plusieurs approches proposent des solutions assez générales qui reposent sur des principes de conception réutilisables, bien que leur implantation soit spécifique à des environnements de développement donnés (OpenCOM, Fractal ou SmartFrog) [56, 66, 39].

4.3 Synthèse

Cette présente étude des actions sur les systèmes distribués fondés sur des architectures fait apparaître plusieurs défis. Le paramétrage des architectures est une capacité fondamentale proposée seulement par un petit nombre d'approches et souvent avec de fortes limitations. Ces limitations sont principalement liées à l'absence d'approches reposant sur un langage ADL d'ordre supérieur. Toutefois, les approches reposant sur des ADL décrivant l'architecture des systèmes par intension exhibent des capacités de paramétrage de très haut niveau et qui semblent très prometteuses. La plupart des approches pour le déploiement de systèmes distribués reposent sur des algorithmes de déploiement externes peu flexibles et qui n'offrent pas de moyen simple pour contrôler la distribution, les synchronisations et la coordination des processus de déploiements, ou plus généralement des actions sur les systèmes distribués. D'autres approches comme *SmartFrog* et *BPEL* ciblent spécifiquement le contrôle de la synchronisation et de la coordination des actions. Ces approches souffrent toutefois de limitations qui empêchent, par exemple, de contrôler directement et simplement le processus de déploiement des systèmes distribués administrés. Enfin, l'expression du dynamisme des systèmes administrés a été spécifiquement étudiée dans des approches comme *Safran* ou *Plastik*. Aucune des approches que nous avons étudiées n'apparaît capable d'intégrer des évolutions des systèmes administrées non prévues initialement. Encore une

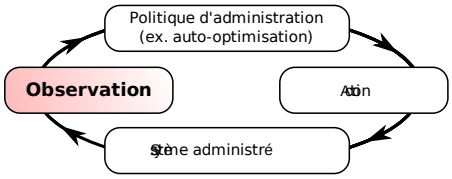
fois, cette limitation est liée au fait qu'aucune de ces approches ne propose un ADL dynamique d'ordre supérieur.

Nous proposons une approche qui repose sur une nouvelle construction langage spécialisée pour la représentation de systèmes distribués à architectures dynamiques. Nous concevons ainsi un *langage ADL dynamique d'ordre supérieur*. Cet ADL permet la description d'architectures et de procédures de déploiement entièrement *paramétrables*. L'architecture et son déploiement y sont décrits de façon procédurale, ce qui autorise le *contrôle de la distribution, de la coordination et des synchronisations* de l'architecture et de son processus de déploiement. Notre langage ADL est construit par extension d'un langage de programmation généraliste et multi-paradigme. Cela permet d'intégrer directement des reconfigurations dynamiques arbitraires en décrivant ainsi des *architectures dynamiques*. Ces travaux sont décrits plus en détail dans le chapitre 8.

Nous avons étudié, dans ce chapitre, les défis et travaux apparentés d'actions sur les systèmes à architectures distribuées. La mise en œuvre d'actions sur les systèmes distribués conduit à la construction de systèmes distribués fondés sur des architectures dynamiques. Le chapitre suivant s'intéresse précisément aux travaux connexes à l'observation des systèmes distribués fondés sur des architectures dynamiques.

Chapitre 5

Observation du système administré



Ce chapitre étudie l'état de l'art relatif à l'observation des systèmes administrés, dans le cadre de l'administration fondée sur l'architecture. Nous identifions tout d'abord les défis relatifs à l'observation des systèmes administrés fondés sur des architectures dynamiques. Nous étudions ensuite pour chaque défi les travaux apparentés d'observation des systèmes administrés. Enfin, nous présentons une synthèse des différents travaux présentés, puis nous positionnons notre contribution au regard des lacunes que nous identifions.

5.1 Défis scientifiques d'observation des systèmes administrés

Nous concevons l'architecture d'un système distribué comme une structure de données distribuée, dynamique et naturellement assimilable à un graphe. Cette section identifie les défis scientifiques concernant l'observation des systèmes administrés fondés sur des architectures dynamiques.

Distribution de l'état de l'architecture observée. L'architecture est une structure de données distribuée sur un ensemble de machines a priori non déterminé. L'état global du système n'existe pas de manière centralisée et directement accessible à un processus simple et non distribué. Autrement dit, l'architecture d'un système n'est pas immédiatement connaissable ni explorable par un processus centralisé.

Dynamisme de l'état de l'architecture observée. Dans le contexte de la construction de systèmes autonomes, l'architecture d'un système administré est soumise à des adap-

tations dynamiques. L'état de l'architecture du système administré subit donc constamment des évolutions dynamiques. Calculer et travailler à partir d'une telle structure dynamique implique de maîtriser ce dynamisme, sans quoi chaque observation réalisée peut immédiatement être invalidée en raison de nouvelles évolutions du système.

Navigation. L'architecture d'un système distribué a naturellement une structure de graphe définie, par exemple, par les relations de composition et les liaisons entre interfaces. La navigation entre les différents éléments d'une architecture est une capacité particulièrement adaptée à son analyse et à l'extraction d'informations.

5.2 Travaux apparentés d'observation des systèmes administrés

Nous présentons différentes approches qui apportent des éléments de réponses aux trois défis énoncés précédemment. Nous nous intéressons plus particulièrement aux travaux qui touchent à la représentation et à l'extraction d'informations depuis une structure de données distribuée et dynamique ainsi qu'à la navigation dans les structures de données ayant une structure de graphe.

5.2.1 Distribution de l'état

Une partie des approches pour l'observation et l'extraction d'information dans des structures de données distribuées dynamiques ne gère pas la distribution de l'état sur un ensemble de machines a priori non maîtrisé. Parmi ces approches, nous pouvons citer les structures de données hiérarchiques fondées sur la norme XML/XPath qui ne fait pas référence à la distribution de l'état de la structure [35]. De façon analogue, les approches fondées sur des bases de données (relationnelles et/ou orientées objet, ou encore les bases de données actives [93]) reposent sur une gestion contrôlée de l'état des données qui sont ainsi centralisées sur un sous-ensemble bien déterminé de machines.

Les autres approches intègrent naturellement la distribution de l'état. C'est notamment le cas des travaux de P.-C. David sur le canevas *Safran* relatifs aux architectures Fractal en association avec le langage d'interrogation FPath dérivé du langage XPath [22, 34, 39]. L'implantation Java de référence du modèle de composant Fractal, Julia, intègre la couche de distribution Fractal/RMI qui permet la construction et l'exploitation transparente d'architectures Fractal distribuées. Fractal/FPath permet ainsi l'extraction d'informations depuis l'architecture dont l'état est effectivement distribué.

5.2.2 Dynamisme de l'état

Une première approche pour la gestion de structures de données dynamiques repose sur les bases de données actives. La gestion des données dynamiques repose notamment sur le principe des règles ECA (évènement, condition, action) mis en œuvre dans le cadre des bases de données actives sous forme de *triggers* [93]. Les *triggers* sont

à l'origine des vues de données (les vues SQL dans les bases relationnelles, ou les vues OQL dans les bases orientées objet) qui constituent un exemple de calcul capable de capturer le dynamisme des données [59].

Par exemple, une vue OQL représente une collection d'objets définie par intension et dont le contenu effectif reste cohérent avec l'état actuel des données. L'exemple suivant montre comment créer deux vues OQL. La première représente le sous-ensemble des films produits par Disney parmi une base de données de films. La seconde en extrait les films dont la durée est inférieure à 60 minutes. L'état effectif de ces vues OQL est mis à jour et reste ainsi valide lorsque la base de données évolue.

```
define DisneyMovies as select m from m in Movies where m.ownedBy.name = 'Disney';
define DisneyShortMovies as select m from DisneyMovies where m.length < 60;
```

Les travaux de Nagapraveen et al. sur Fractal/ECA s'inspirent directement des mécanismes proposés par les bases de données actives pour intégrer le dynamisme dans les architectures distribuées Fractal [87]. En particulier, Fractal/ECA permet de construire des structures équivalentes aux *triggers* des bases de données sous forme de règles ECA. Toutefois, les travaux sur Fractal/ECA se limitent à ce niveau et ne fournissent pas l'équivalent des vues sur les architectures permettant de capturer le dynamisme des architectures Fractal.

5.2.3 Navigation

Les approches XML/XPath et Fractal/FPath qui s'inspire directement du langage XPath intègre directement un langage spécialisé dans la navigation dans les structures de données hiérarchique ou de graphe.

De façon analogue, les bases de données orientées objet (et plus généralement l'ensemble des langages orientés objet) intègrent directement le principe de la navigation dans le langage d'interrogation OQL. Il est ainsi possible de naviguer simplement d'objet en objet selon les relations qui existent entre eux. L'opérateur "." permet précisément de suivre une relation comme dans `m.ownedBy.name = 'Disney'`. À l'inverse, les bases de données relationnelles fondées sur le langage d'interrogation SQL ne sont pas optimisées pour la navigation dans des structures à objets, qui implique alors des jointures coûteuses.

5.3 Synthèse

L'observation des systèmes distribués fondés sur des architectures dynamiques fait apparaître trois défis. Aucun des travaux apparentés d'observation des systèmes ne répond à ces trois défis simultanément en proposant une solution permettant d'intégrer à la fois la distribution de l'état, le dynamisme de l'état et des capacités de navigation dans la structure de graphe que forme l'architecture.

Le canevas *Safran* pour les architectures Fractal en environnement distribué au moyen de Fractal/RMI, en combinaison avec le langage de navigation et d'interrogation FPath

et étendu avec le système Fractal/ECA permettant de définir des réactions sous forme de chaînes d'évènements (*triggers*) constitue un début de réponse prometteur aux trois défis que nous avons identifiés. D'une part, cette approche s'affranchit de la distribution effective de l'état sur un ensemble de machines non déterminé grâce à Fractal/RMI. Cette approche permet la navigation dans les architectures au moyen du langage d'interrogation et de navigation FPath inspiré du langage XPath pour XML. Enfin, Fractal/ECA implante un système de réaction évènementiel. Cela fournit les moyens techniques de mettre en œuvre des calculs dynamiques, et cela permettrait ainsi de capturer le dynamisme des architectures.

Nous proposons de nouvelles constructions langages fondées sur la combinaison d'un *modèle de calcul dynamique* et d'une *modèle de calcul distribué*. Nos constructions langages permettent de raisonner et de calculer indépendamment de la nature distribuée et dynamique des éléments considérés. Ces constructions langages sont mises en œuvre pour la construction d'une *bibliothèque d'interrogation et de navigation* spécialisée pour les systèmes distribués fondés sur des architectures dynamiques. Ces travaux sont présentés plus en détail dans le chapitre 9.

Nous avons présenté, au cours de cette première partie, notre vision de l'état de l'art sur les domaines qui sont pertinents pour la mise en œuvre de systèmes distribués auto-optimisés. Cela comporte une étude de l'état de l'art en matière d'informatique autonome, d'optimisation des systèmes distribués, d'action sur les systèmes distribués et enfin d'observation des systèmes distribués. Cette étude nous a amené à émettre trois propositions qui constituent notre contribution dans les domaines des politiques d'auto-optimisation des systèmes distribués, des actions sur les systèmes distribués et enfin des observations des systèmes distribués. Nous présentons justement dans la partie suivante les trois volets de notre contribution pour la construction de systèmes distribués auto-optimisés fondés sur des architectures dynamiques.

Deuxième partie

Contributions scientifiques

Chapitre 6

Contexte

Nous présentons, dans ce chapitre, le contexte de notre étude. Dans un premier temps, nous présentons les contextes techniques ayant servi à la mise en œuvre de nos prototypes. Nous présentons ensuite les différents contextes applicatifs qui ont servi à l'expérimentation et l'évaluation de nos propositions.

6.1 Contexte technique

Ce section regroupe une description de la plate-forme d'administration autonome Jade, une présentation du modèle de composant Fractal et enfin une introduction au langage Oz et à sa plate-forme distribuée Mozart/Oz.

6.1.1 Plate-forme d'administration autonome Jade

Notre travail s'intègre dans le cadre de la plate-forme d'administration autonome dénommée Jade [17, 19, 41]. La plate-forme Jade cherche à simplifier l'administration des systèmes distribués complexes. Une part importante de la complexité des tâches d'administration provient de l'hétérogénéité des systèmes que l'on souhaite administrer ; cela est d'autant plus marqué lorsque l'on parle des systèmes patrimoniaux.

Systèmes patrimoniaux. Un besoin important pour l'administration des infrastructures logicielles est la capacité de gérer des logiciels patrimoniaux ayant des interfaces d'administration très hétérogènes. Pour faire face à cette hétérogénéité de boîtes noires, Jade définit une interface minimale et uniforme d'administration. Jade encapsule chaque boîte noire à administrer dans un composant logiciel qui fournit l'interface uniforme d'administration. Ainsi, du point de vue de la plate-forme d'administration Jade, l'encapsulation de systèmes patrimoniaux hétérogènes fournit l'illusion de systèmes administrables homogènes. La figure 6.1 illustre le cas d'un système distribué constitué de quatre systèmes patrimoniaux. Chacun des systèmes est encapsulé dans un composant logiciel Jade fournissant l'interface d'administration uniforme et explicitant ses liaisons avec les autres systèmes patrimoniaux. Ainsi, la plate-forme Jade

représente l'architecture du système distribué administré sous la forme d'un graphe de composants d'encapsulation, et permet leur manipulation au moyen de l'interface uniforme d'administration. Les composants d'encapsulation sont causalement connectés aux systèmes patrimoniaux qu'ils représentent. En particulier, les composants d'encapsulation répercutent les opérations d'administration sur les systèmes patrimoniaux en traduisant les opérations de l'interface uniforme vers les interfaces d'administration hétérogènes des systèmes patrimoniaux.

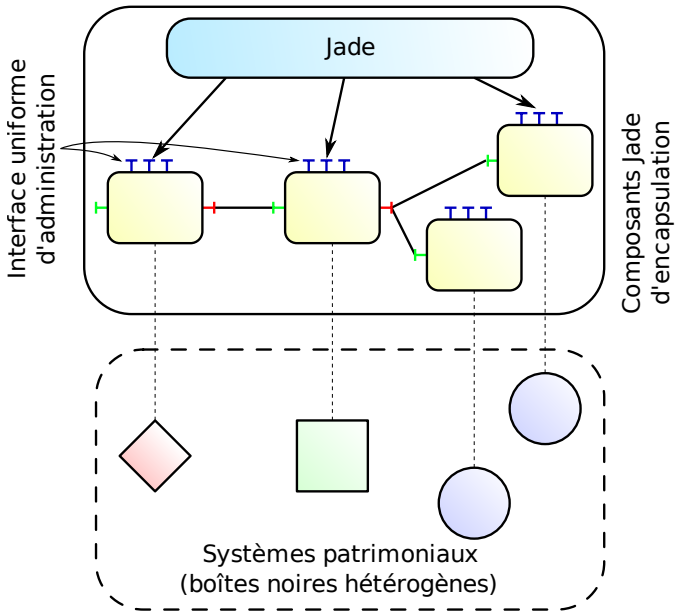


FIG. 6.1 – Jade et l'administration de systèmes patrimoniaux hétérogènes

Jade fournit une bibliothèque de politiques d'administration autonome. Cette bibliothèque prend la forme de gestionnaires autonomes adaptables, comme par exemple un gestionnaire d'auto-réparation chargé d'assurer la pérennité et la disponibilité d'un système administré, un gestionnaire d'auto-protection qui cherche à protéger le système administré contre des activités malveillantes, et enfin un gestionnaire d'auto-optimisation, l'objet de notre étude, chargé d'améliorer ou de garantir les performances du système administré.

Les gestionnaires autonomes de la plate-forme Jade sont organisés sous la forme de boucles de commande qui observent le système administré grâce à diverses sondes, et qui réagissent en manipulant le système administré.

Architecture de la plate-forme Jade. La plate-forme Jade repose sur plusieurs services généraux lui permettant d'administrer des systèmes distribués. Parmi les divers services contribuant au fonctionnement de la plate-forme, comme illustré sur la figure 6.2, on peut citer : l'allocateur de ressources, qui combine un service de découverte des ressources chargé de répertorier les machines disponibles dans le système, et

un service de gestion qui met à disposition ces ressources disponibles ; ou encore, le service de déploiement qui prend en charge l'orchestration des tâches de (re-)déploiement commandées à la plate-forme.

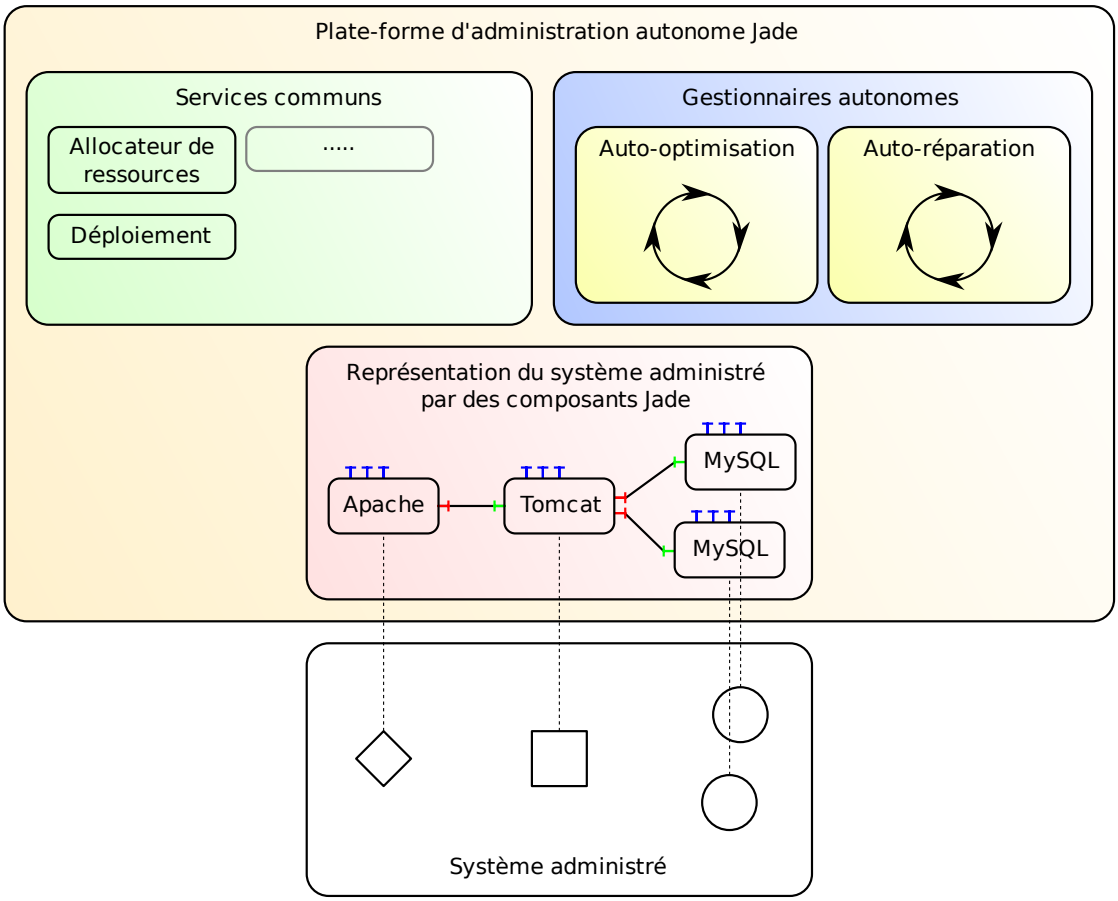


FIG. 6.2 – Architecture générale de la plate-forme d'administration autonome Jade

Notre travail consiste précisément à doter la plate-forme Jade de la faculté d'auto-optimiser les systèmes administrés. Cela revient à proposer dans Jade un gestionnaire d'administration autonome spécialisé pour l'optimisation des systèmes administrés.

La plate-forme Jade repose sur l'encapsulation des systèmes patrimoniaux dans des composants logiciels pour représenter l'architecture des systèmes administrés. Ces composants logiciels d'encapsulation sont construits selon le modèle de composant Fractal que nous décrivons précisément dans la section qui suit.

6.1.2 Modèle de composant Fractal

Nous décrivons dans cette section le modèle de composant Fractal [22, 21]. Fractal est un modèle général qui reprend les concepts issus des principaux travaux relatifs aux architectures (dont par exemple ACME [54]). Fractal n'est spécifique à aucun langage de programmation ni à aucun domaine d'application particulier, ce qui le rend spécialement flexible et adaptable. Le modèle de composant Fractal est un outil de génie logiciel permettant d'organiser logiquement un système en décrivant son architecture. C'est également un outil pour la construction et la configuration dynamique de systèmes distribués. Le modèle Fractal permet de construire des architectures qui reposent sur deux concepts : d'une part, la *composition* de composants qui représente une notion d'encapsulation et d'isolation, et d'autre part, la *liaison* entre composants qui représente les chemins de communication autorisés entre les composants.

Un composant Fractal représente et implante un ensemble de fonctionnalités. Un composant est organisé en deux parties logiques : (1) la *membrane* du composant, qui représente une frontière isolant le composant du reste du système et qui abrite la *partie contrôle* du composant ; et (2) le *contenu* du composant, qui désigne la *partie fonctionnelle* du composant, interne à celui-ci. La partie fonctionnelle met en œuvre les fonctionnalités représentées par le composant, tandis que la partie contrôle implante diverses opérations de contrôle sur le composant (comme par exemple la gestion de son cycle de vie).

Composition. Le modèle Fractal distingue deux types de composants : les composants *primitifs* dont la partie fonctionnelle a la forme d'une boîte noire, et les composants *composites* dont la partie fonctionnelle est observable et est elle-même organisée sous forme de composants. La composition Fractal autorise un composant à être *partagé* par plusieurs composites. Le partage de composant est particulièrement adapté à la modélisation des ressources.

Liaisons. Un composant est équipé d'*interfaces* qui lui permettent d'interagir avec d'autres composants : les interfaces *serveur* représentent des services fournis ou exportés par le composant ; les interfaces *client* représentent des services requis ou importés par le composant. La communication entre composants n'est possible qu'au moyen d'une *liaison* entre une interface client et une interface serveur.

Diverses propriétés peuvent être associées aux interfaces des composants. Par exemple, une interface client peut être déclarée *nécessaire*, indiquant que le composant ne pourra fonctionner si le service requis représenté par l'interface client n'est pas pourvu ; à l'inverse, une interface client peut être déclarée *optionnelle*, indiquant que le composant pourra fonctionner en l'absence du service requis exprimé.

Contrôle. La partie contrôle d'un composant implante diverses capacités non fonctionnelles du composant, comme par exemple le contrôle du cycle de vie du composant, de ses liaisons ou encore de son contenu. Ces capacités de contrôle sont implantées

par des *contrôleurs* appartenant à la membrane du composant, et accessibles au moyen d'*interfaces de contrôle*.

Les contrôles exercés sur le composant sont entièrement libres et non contraints par Fractal. En revanche, Fractal caractérise différents niveaux de contrôle correspondant à des degrés croissants de capacités de contrôle sur le composant. Le niveau de contrôle le plus faible correspond à l'absence totale de capacité de contrôle sur le composant apparaissant alors comme une boîte noire. À l'opposé du spectre des capacités de contrôle, le niveau de contrôle le plus élevé caractérisé par Fractal intègre des capacités d'introspection et d'intercession, associées à différents contrôleurs spécifiés par Fractal.

Le contrôleur d'attributs permet de consulter et de modifier les attributs configurables associés au composant.

Le contrôleur de liaisons contrôle l'introspection, l'établissement et la rupture des liaisons des interfaces du composant.

Le contrôleur de contenu gère l'introspection, l'ajout et le retrait des sous-composants du composant.

Le contrôleur de cycle de vie contrôle l'état du composant, son démarrage et son extinction.

La figure 6.3 présente un composant composite contenant un unique sous-composant primitif. Le composant composite présente deux interfaces serveur et deux interfaces client. Deux liaisons architecturales permettent d'associer l'une des interfaces serveurs et l'une des interfaces client au sous-composant. Le composite présente également une part d'implantation, laquelle est également reliée à une interface serveur et une interface client au moyen de liaisons d'implantation. Enfin, le composite expose trois interfaces de contrôle (on peut par exemple imaginer que ces interfaces implantent les contrôleurs de liaison, de cycle de vie et d'attributs du composant).

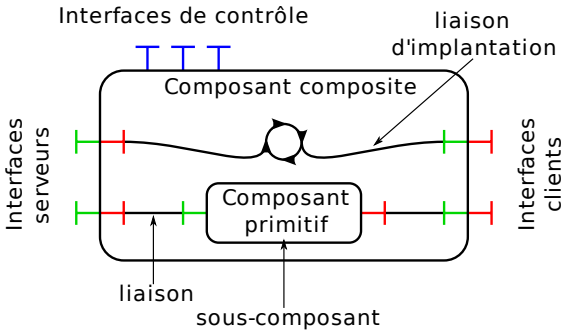


FIG. 6.3 – Structure générale d'un composant Fractal.

Le modèle de composant Fractal est le support que nous adoptons pour décrire l'architecture des systèmes distribués dynamiquement reconfigurés. Dans le cadre de notre proposition relative à la conception et à l'implantation de tels systèmes distribués, nous

mettons en œuvre le canevas FructOz spécialisé dans la construction de systèmes distribués dynamiques. Cette réalisation repose sur le langage Oz et sa plate-forme distribuée Mozart/Oz que nous introduisons maintenant.

6.1.3 Plate-forme Mozart/Oz

Oz est un langage de programmation qui intègre de façon homogène la programmation déclarative (impérative et fonctionnelle), la programmation concurrente, la programmation objet et la programmation par contraintes [31, 60]. À cela, la plate-forme Mozart ajoute une couche de gestion transparente de la distribution qui facilite la construction de systèmes distribués. Le langage Oz présente de nombreuses caractéristiques intéressantes que l'on décrit dans la suite.

Variables. Les variables Oz sont des variables logiques, qui sont soit définies et liées à leur valeur définitive (non mutable), soit non définies (ou non *liées*). La synchronisation est le comportement par défaut sur les variables non liées : toute opération tentant d'utiliser une variable non liée suspend l'exécution du thread à l'origine de l'opération et synchronise son redémarrage sur la liaison de cette variable. Les threads ainsi bloqués reprendront leur exécution dès que la variable à l'origine de leur suspension sera liée (c'est-à-dire définie). Par analogie, ce comportement remplace le "NullPointerException" du langage Java, ou encore le "Segmentation Fault" lors de l'accès à un pointeur nul en C ou C++. La cohérence de ce comportement est assurée par la caractéristique monotonique des variables Oz : une variable Oz ne peut être liée qu'une seule fois ; une fois liée, la valeur d'une variable n'est plus modifiable. La liaison d'une variable s'effectue par l'opérateur d'unification : `X = Y`. Les noms de variables commencent par une lettre majuscule.

Valeurs. Les valeurs Oz peuvent être des valeurs primitives de type entier, nombre à virgule flottante, chaîne de caractères (délimitée par des guillemets doubles), atome (identificateur débutant par une lettre minuscule ou bien délimité à l'aide de guillemets simples), nom unique non forgeable. La forme générale d'une valeur composée est celle d'un enregistrement (*record*) noté `label(f1:V1 f2:V2 ... fN:VN)`. Les champs de l'enregistrement `f1`, `f2`, etc, sont associés respectivement aux valeurs `V1`, `V2`, etc. L'étiquette (*label*) ainsi que les champs de l'enregistrement sont des littéraux (entiers, atomes ou noms uniques). L'utilisation d'un enregistrement s'effectue principalement par l'opérateur de sélection "." qui permet de consulter un champ particulier comme dans `Y = X.f1` qui permet de sélectionner ou de définir la valeur associée au champ `f1` de l'enregistrement `X`. Les enregistrements peuvent faire l'objet de filtrage au moyen de la structure **case** `X` **of** `p1` **then** `S1` **elseof** `p2` **then** `S2` ... **end**, qui teste l'enregistrement `X` successivement avec les patrons `p1`, puis `p2`, etc., et exécute la déclaration (`S1`, ou `S2`, etc) correspondant au premier patron pour lequel le test est positif. Les listes sont mises en œuvre au moyen d'enregistrements récursifs particuliers ayant la forme `'|'(Head Tail)`, aussi noté `Head | Tail`,

ou Head désigne le contenu d'une cellule de la liste et Tail est soit une liste, soit l'élement nil indiquant la fin de la liste.

Procédures et fonctions. Une procédure est déclarée par : **proc** {MyProc X1 X2 ...} S **end**, qui définit la procédure MyProc ayant les paramètres X1, X2, etc., et le corps S. Les paramètres Xi peuvent être utilisés en entrée ou en sortie grâce au variables non liées : un paramètre de sortie est un paramètre auquel on associe une variable non liée et qui sera liée par l'exécution de la procédure. Une invocation de procédure prend la forme : {MyProc V1 V2 ...}. Une fonction se déclare de manière similaire : **fun** {MyFun X1 X2 ...} E **end** et correspond à une procédure dotée d'un paramètre de sortie implicite.

```
% Recursive function computing the fibonacci sequence
fun {Fibo N}
    if (N < 2) then 1 else {Fibo (N−1)} + {Fibo (N−2)} end
end
```

Threads. La concurrence est grandement facilitée, d'une part par le très faible coût lié à l'utilisation des threads, et d'autre part par la simplicité de mise en œuvre des synchronisations qu'offre le langage Oz. Un thread est créé par la déclaration **thread** S **end**, qui indique que la déclaration S doit être exécutée en concurrence dans un nouveau thread. Par exemple, le calcul de $f(x)$ dans Y = {F X} peut être effectué en parallèle très simplement : Y = **thread** {F X} **end**.

```
% Parallel computation of the fibonacci sequence
fun {ParallelFibo N}
    if (N < 2) then 1 else thread {ParallelFibo (N−1)} end + thread {ParallelFibo (N−2)} end end
end
```

Objets et classes. Oz intègre également la programmation orientée objet. Très sommairement, une classe se définie ainsi :

```
class MyClass
    % features are unmutable fields
    feat feat1 feat2 ...
    % attributes are mutable fields
    attr attr1 attr2 ...
    % a method
    meth myMethod(Param1 Param2 ...)
        ...
    end
    ...
end
```

L'instanciation d'objets et l'invocation de méthode sur un objet s'effectuent comme suit :

```
% Instantiate an object of class MyClass
Object = {New MyClass initMethod(Param1 ...)}
% A method call
{Object myMethod(Param1 Param2 ...)}
```

Ports. Les variables non liées sont le support de nombreux mécanismes et constructions, et notamment les constructions infinies dont font partie les *streams*. Un stream est une liste dont la queue n'est jamais définie, c'est-à-dire dont la queue est une variable non liée. L'ajout d'un élément au stream s'effectue en liant la queue de la liste à un couple Head | Tail pour lequel la nouvelle queue (Tail) n'est pas définie. Les *ports* Oz implantent ce comportement.

Synchronisation. Oz intègre plusieurs mécanismes de synchronisation : (1) la synchronisation sur la liaison de variables, (2) les verrous traditionnels (*lock*) et (3) la synchronisation par exécution paresseuse (*lazy*).

1. L'exemple suivant montre comment utiliser la liaison de variables comme outil de synchronisation :

```
%% Declare X (unbound yet)
X
thread
   ... % do something
   {Wait X} % synchronize on X definition: suspend the thread until X is defined
   ... % so something
end
thread
   ... % do something
   X = ok % send synchronization signal: define X
   ... % do something
end
```

   Il est également possible de tester l'état lié ou non lié d'une variable avec la fonction {IsDet X}.

2. Oz offre également un mécanisme de synchronisation fondé sur les verrous traditionnels. Un verrou est créé par L = {NewLock}, et est ensuite exploité au moyen de la construction **lock** L **then** S **end** qui acquiert le verrou L pour exécuter la séquence d'instructions S.

3. Enfin, Oz intègre des mécanismes d'évaluation paresseuse. La déclaration paresseuse X = {ByNeed **fun** {$} E **end**} déclenchera l'évaluation de l'expression E uniquement lorsque cela sera nécessaire à la poursuite de l'exécution. Il est alors possible d'établir une synchronisation sur la nécessité d'évaluer une variable paresseuse avec la primitive : {WaitNeeded X}, qui bloque le thread courant jusqu'à ce que la variable X soit effectivement rendue nécessaire pour la poursuite des opérations.

Foncteurs et modules. Oz permet de structurer logiquement les programmes en modules. Un module est déclaré par la structure de foncteur (*functor*), lequel explicite les modules qu'il importe et les valeurs qu'il exporte. Dans l'exemple suivant, nous déclarons un foncteur MyFunctor qui importe le module System, qui définit la fonction Fibo, une valeur X qui est ensuite affichée sur la console, et qui exporte enfin Fibo et X.

```
functor MyFunctor
```

```
export x:X fibo:Fibo
import System
define
    %% This code will be executed when instantiating the module
    fun {Fibo N} ... end
    X = {Fibo 15} % compute X=fibonacci(15)
    {System.show X} % display X
end
```

Le foncteur est une valeur inerte du langage Oz (à l'instar d'un fichier de classe Java), mais qui peut néanmoins être directement manipulée dans le langage. L'instanciation d'un foncteur produit un module qui est une valeur active (à l'image d'une classe Java chargée depuis un fichier de classe). L'instanciation d'un foncteur en un module est effectuée par un gestionnaire de modules (ModuleManager) chargé de résoudre les dépendances (imports) des modules à charger :

```
% Create a new module manager and instantiate the module
ModuleManager = {New Module.manager init}
MyModule = {ModuleManager apply(MyFunctor $)}
% Now we can use what the module exports
{System.show MyModule.x} % display the value X exported by the module
{System.show {MyModule.fibo 15}} % invoke the function Fibo and display the result
```

Distribution. La plate-forme Mozart intègre un environnement distribué permettant d'exécuter et de faire interagir des programmes sur différentes machines. La plate-forme Mozart rend la programmation sur l'environnement distribué quasiment transparente. L'exécution de code à distance s'effectue à l'aide de proxy de gestionnaires de modules présents sur les machines distantes, et permettant d'instancier des modules à distance.

```
%% Remote code execution
% First, bootstrap a virtual machine on the remote host
RemoteModuleManager = {New Remote.manager init(host:'hostname' fork:ssh)}
% Apply the functor on the remote host
RemoteModule = {RemoteModuleManager apply(MyFunctor $)}
% The result corresponds to the export section of the functor
{System.show RemoteModule.x} % display the value X exported by the remote module
{System.show {RemoteModule.fibo 15}} % invoke the function Fibo exported by the remote module
```

6.2 Contexte applicatif

Pour la validation de nos solutions d'auto-optimisation dans la plate-forme Jade, nous avons considéré différents domaines applicatifs : les services à messages, les services Internet multi-étagés en pipeline, et enfin les services de surveillance des systèmes informatiques.

6.2.1 Services à messages

La multiplication des systèmes distribués stimulée par le développement de l'Internet a fait apparaître le besoin d'intégration des systèmes distribués, afin notamment d'améliorer leur interopérabilité, leur portabilité et leur flexibilité. Les services à messages (Message-oriented Middleware, ou MOM) répondent à ce besoin en proposant une technologie d'intégration des systèmes distribués fondée sur l'échange asynchrone de messages. Dans les services à messages, le message constitue la structure fondamentale sur laquelle repose toute communication, coordination et synchronisation [18]. De nombreux systèmes reposent aujourd'hui sur les services à messages, comme par exemple des services de messagerie tels que des forums ou des services de votes électroniques, ou encore des services de surveillance tels que celui mis en œuvre par la RATP (métro parisien) pour contrôler l'état des stations de métro (escaliers roulants, etc.). L'un des avantages des services à messages provient de leur capacité à fournir des garanties de robustesse et de performance, par exemple au moyen de files d'attente persistantes, et ce de manière orthogonale au cœur fonctionnel du service qu'ils implantent.

La communauté Java a normalisé une interface de programmation pour les services à messages dénommée JMS (*Java Message Service*) [78]. Le standard JMS comporte un modèle de communication point-à-point et un modèle de communication de groupe. Dans le modèle de communication point-à-point, un *producteur* adresse un message directement à une file d'attente identifiée, et un *consommateur* extrait puis traite le message depuis cette même file d'attente. Dans le modèle de communication de groupe, un *émetteur* (*publisher*) adresse un message à un groupe de diffusion identifié (*topic*) ; symétriquement, un client communique son intérêt pour un groupe de diffusion et reçoit alors les messages adressés à ce groupe en tant qu'*abonné* (*subscriber*).

La figure 6.4 présente l'architecture d'un service à messages. Dans cet exemple, le service à messages est implanté par un serveur JORAM (*Java Open Reliable Asynchronous Messaging*), qui héberge deux files d'attente et un groupe de diffusion [33]. Le service possède deux clients, l'un est un système producteur de messages, tandis que l'autre est un système consommateur de messages.

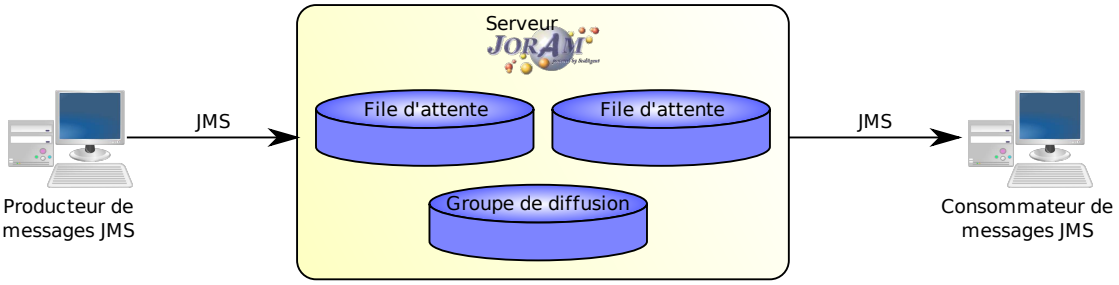


FIG. 6.4 – Architecture d'un service à messages (JMS/JORAM)

JORAM est une implantation de JMS intéressante parce qu'elle propose une fonctionnalité supplémentaire permettant l'agrégation des files d'attente en grappe afin

de tirer parti d'une plus grande puissance de traitement. Une file d'attente en grappe (*clustered queue*) est mise en œuvre par un ensemble de files d'attente standards interconnectées et qui sont capables d'échanger des messages en fonction de leurs niveaux de charge respectifs. Du point de vue de l'utilisateur, une file d'attente en grappe est conforme à l'interface JMS et offre la même interface qu'une file d'attente standard, ce qui donne l'illusion de manipuler une file d'attente unique et rend ainsi son utilisation transparente. Toutefois, ce mode de fonctionnement ne garantit pas l'ordre de délivrance des messages émis par des producteurs différents. L'extension proposée par JORAM intègre un mécanisme de répartition de la charge sur un ensemble de machines réalisant la file d'attente en grappe. Le mécanisme de répartition de charge distribue les connexions des différents clients sur l'ensemble des serveurs composant la grappe. La figure 6.5 présente un service à messages fondé sur une file d'attente en grappe composée de deux machines hébergeant chacune un serveur JORAM. Le service est utilisé par 5 clients, dont 3 producteurs de messages et 2 consommateurs de messages. Les connexions des différents clients sont réparties sur les deux serveurs composant le service afin d'équilibrer la charge imposée au service.

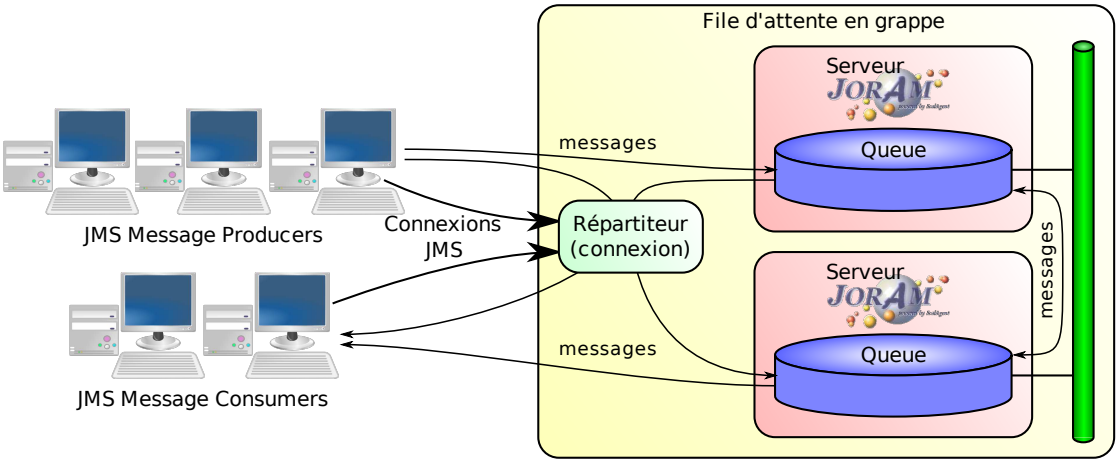


FIG. 6.5 – Architecture d'un service à messages (JMS/JORAM) en grappe

Les services à messages reposant sur des files d'attente en grappe sont des systèmes distribués dupliqués en grappe et interagissant au moyen de communications asynchrones. L'exploitation des services à messages fondés sur cette interface dans le contexte de l'Internet a démontré la nécessité d'optimiser ces services pour en assurer le passage à l'échelle. L'optimisation d'un service à messages consiste à déterminer la bonne configuration architecturale du système afin de satisfaire certains critères de performances. Cela revient à déterminer précisément le nombre approprié de files d'attente constituant ce système distribué dupliqué.

6.2.2 Services Internet

Les services Internet constituent une catégorie désormais incontournable de systèmes distribués reposant sur l'exploitation des réseaux de machines. Il existe de très nombreux services Internet ayant diverses formes et usages, parmi lesquels les services de commerce électronique tels que le site d'enchères en ligne *eBay.com* [45] ou encore la boutique en ligne *Amazon.com* [7], les services de diffusion de contenus multimédia tels que les sites de partage de vidéos *YouTube* [126] ou *Dailymotion* [38], etc.

Les services Internet reposent classiquement sur un fonctionnement organisé sur le modèle client/serveur. Un *client* sollicite un service au moyen d'une requête client. La requête est acheminée par le réseau vers le système *serveur*, qui va traiter la requête émise par le client, puis renvoyer au client une réponse correspondant au résultat du traitement de la requête. HTTP, FTP, SMTP, POP ou encore IMAP sont autant de protocoles utilisés par des services Internet fonctionnant sur le modèle client/serveur.

Le modèle client/serveur classique repose sur un serveur constitué d'un unique système de traitement des requêtes client. Pour des raisons de fiabilité, de performances et de passage à l'échelle des services Internet, le modèle multi-étagé étend le modèle client/serveur classique pour permettre la répartition de la charge sur plusieurs systèmes de traitements. Un service dont l'architecture est multi-étagée est organisé selon un pipeline reliant les systèmes de traitement constituant les différents étages du service. Le traitement d'une requête par un service multi-étagé implique alors de multiples traitements sur les différents étages composant le service, et se propage d'étage en étage dans le pipeline de systèmes formant le service multi-étagé.

Une importante catégorie de Services Internet est organisée selon l'architecture J2EE [77] (*Java 2 Enterprise Edition*, récemment rebaptisé JavaEE [79]) qui constitue un standard de fait pour la réalisation de services multi-étagés. J2EE est une spécification qui propose un cadre pour la conception de services Internet, et qui est largement acceptée et reconnue dans les milieux industriels, par ses apports en termes de standardisation et de séparation des préoccupations pour la conception et l'exploitation des services Internet. La spécification J2EE repose sur le langage Java et sur diverses technologies et interfaces (XML, JDBC, e-mail, JMS, etc), dont certaines sont spécifiques à J2EE, comme les Servlets, les JSP (JavaServer Pages) ou encore les EJB (Enterprise JavaBean).

Plus concrètement, les systèmes J2EE sont généralement mis en œuvre au moyen des quatre systèmes suivants, et organisés en pipeline comme illustré sur la Figure 6.6 :

- Le serveur Web, ici réalisé par un serveur Apache HTTPD [47], reçoit les requêtes HTTP des clients. Certaines requêtes peuvent être entièrement traitées par le serveur Web lorsqu'il s'agit d'accéder à un document statique (par exemple, une page Web statique) ; les autres requêtes sont transmises au conteneur de Servlets.
- Le conteneur de Servlets, par exemple un serveur Apache Tomcat [49], exécute des programmes Java (les Servlets ou JSP) qui commandent des traitements puis génèrent les documents réponses à retourner aux clients (pages Web dynamiques). Les traitements commandés par les Servlets sont exécutés par des EJB.
- Le conteneur d'EJB, par exemple un serveur d'applications JOnAS [32] ou JBoss [61], exécute le code fonctionnel de l'application, qui calcule les données demandées

par les Servlets. Le conteneur d'EJB gère des objets Java persistants qui sont conservés de façon pérenne dans un serveur de base de données.
– Le serveur de base de données, par exemple un serveur MySQL [1] ou PostgreSQL [94], gère l'accès, le maintien de la cohérence et la persistance des données de l'application.

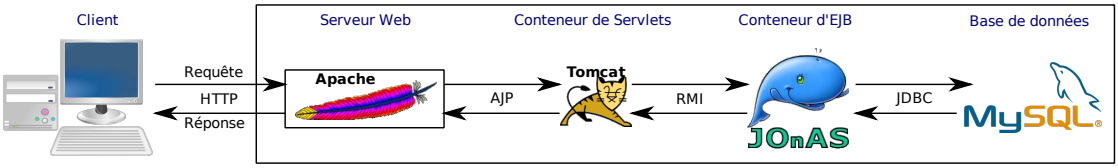


FIG. 6.6 – Architecture d'un service Internet multi-étagé (J2EE)

Pour permettre aux services Internet de traiter de plus grandes quantités de requêtes et de tolérer l'occurrence de pannes logicielles ou matérielles, et ainsi assurer un meilleur passage à l'échelle et une meilleure disponibilité de ces services, les étages de l'architecture J2EE peuvent être dupliqués et distribués sur diverses machines. Comme les éléments du service sont dupliqués sur des machines séparées, l'occurrence d'une panne sur l'un des éléments est absorbée en reportant les traitements sur les autres éléments dupliqués ; par ailleurs, le service bénéficie alors d'une capacité de traitement fonction du nombre de machines et du nombre d'éléments dupliqués. Cela conduit à la construction de systèmes J2EE en grappe comme celui illustré sur la figure 6.7. L'utilisation conjointe des différentes instances des serveurs dupliqués est rendue possible grâce à des répartiteurs de charge (*load-balancer*) qui répartissent la charge sur l'ensemble des serveurs dupliqués et assurent la cohérence entre ces serveurs. Les répartiteurs de charge sont spécifiques à chaque type de serveur.

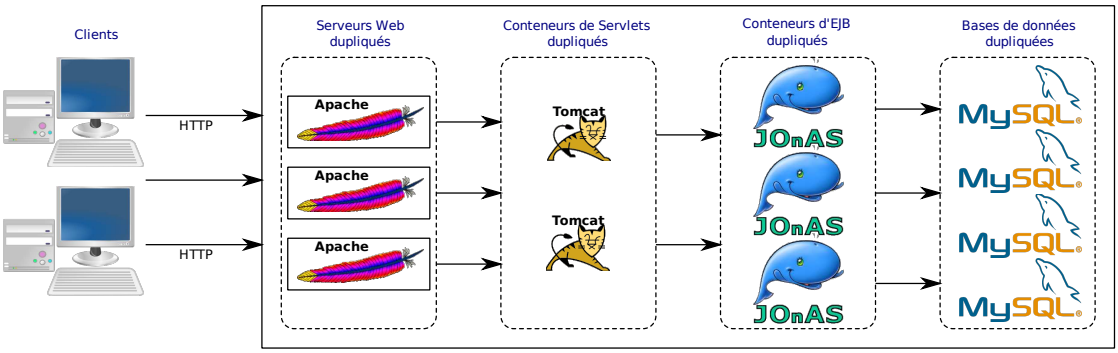


FIG. 6.7 – Architecture d'un service Internet multi-étagé (J2EE) en grappe

Les services Internet sont des systèmes distribués dont l'architecture est complexe, organisée selon un pipeline de systèmes en grappes qui interagissent au moyen de communications synchrones. L'optimisation d'un service Internet consiste à déterminer la configuration architecturale optimale de ce système distribué. Cela revient à déterminer

précisément les nombres de serveurs convenables pour chacun des étages constituant le pipeline du système distribué.

6.2.3 Services de surveillance de systèmes distribués

Lorsque l'on est amené à intervenir sur un système informatique, l'une des premières étapes nécessaire avant toute prise de décision et avant toute opération sur un système administré consiste à s'assurer de son état effectif, à en effectuer un diagnostic précis. Par exemple, il est commun de s'intéresser aux niveaux actuels de consommation des diverses ressources du système (processeur, mémoire, bande passante réseau, etc). La construction de systèmes toujours plus complexes et la perspective des systèmes autonomes ont stimulé le besoin en outils de diagnostic et de surveillance de l'état de ces systèmes administrés. Il s'agit principalement de surveiller, à l'échelle du système global, l'état de l'ensemble des machines physiques (sont-elles disponibles et fonctionnelles ou hors service, fortement ou faiblement utilisées ?), l'état des ressources physiques du système, leur niveau de consommation et leur disponibilité, et aussi l'état des ressources applicatives et des différents services hébergés par ces machines. Toutes ces informations brutes ne sont pas toujours utilisables en l'état et nécessitent généralement des traitements afin d'en extraire des indicateurs synthétiques agrégés qui facilitent le diagnostic.

Les services de surveillance des systèmes distribués ont généralement une structure hiérarchique comme illustré sur la figure 6.8. Chaque nœud de la structure hiérarchique collecte des informations de surveillance, les agrège pour synthétiser une nouvelle information qui est ensuite transmise au nœud hiérarchiquement supérieur. Les informations brutes concernant l'état des ressources physiques sont collectées dans les environnements Linux grâce au système de fichier ProcFS en accédant au contenu du répertoire `/proc`, ainsi que procèdent la plupart des sondes des services bien connus comme SysStat [55], Ganglia [75, 84] ou encore Nagios [50]. Les informations de niveau applicatif sont généralement obtenues de manière spécifique aux applications surveillées (par instrumentation de l'application, par test de disponibilité ou de latence de l'application, etc). Les informations récoltées sont ensuite agrégées puis propagées en remontant dans la hiérarchie des sondes jusqu'à la machine racine du système de surveillance qui dispose alors de l'information à l'échelle du système administré dans sa globalité.

Deux critères de performance nous importent ici lors de la configuration d'un système de surveillance. Le premier est l'impact du système de surveillance sur le système administré, c'est-à-dire son empreinte en termes de consommation des ressources, comme par exemple la consommation en bande passante réseau liée aux échanges de messages entre les sondes. La seconde concerne la qualité de la surveillance obtenue, que nous mesurons ici par la réactivité, c'est-à-dire la rapidité (la latence) avec laquelle les changements d'états des machines sont acheminés et deviennent accessibles sur la machine racine du système de surveillance.

Le service de surveillance est caractérisé par son architecture et par son paramé-

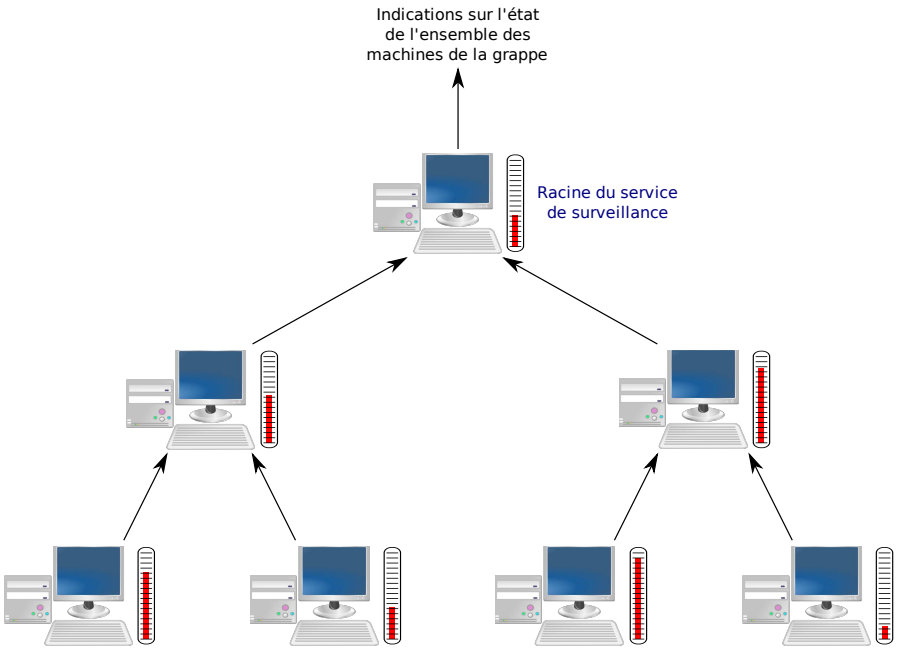


FIG. 6.8 – Structure hiérarchique d'un service de surveillance de système distribué. Dans cet exemple, la structure hiérarchique est d'arité 2 (arbre binaire) et s'étend sur 7 machines.

trage. L'architecture du service de surveillance a ici une structure hiérarchique sous forme d'un arbre, que l'on peut décrire par sa profondeur et son arité (le nombre de branches issues d'un nœud). La profondeur de l'arbre influe directement sur la latence des mises à jour des informations de surveillance au niveau du nœud racine, tandis que l'arité de l'arbre pèse sur la consommation de la bande passante réseau, car elle contrôle la quantité de messages qui transitent par un nœud. Et le paramétrage du service de surveillance intègre notamment un paramètre contrôlant la fréquence de mise à jour des informations par chaque sonde.

Les services de surveillance des systèmes distribués sont des systèmes distribués organisés selon des architectures hiérarchiques. L'optimisation de leurs performances consiste à déterminer la configuration architecturale optimale de ces systèmes, c'est-à-dire la forme de l'arbre hiérarchique qui interconnecte les multiples sondes qui les composent ainsi que le paramétrage local de ces sondes.

6.3 Synthèse

Le contexte de nos travaux est caractérisé par une diversité à plusieurs niveaux.

Diversité des contextes applicatifs. Dans le cadre de notre étude, nous considérons plusieurs contextes applicatifs différents et bien connus : les services à messages, les

services Internet multi-étagés et enfin les services de surveillance des systèmes distribués.

Diversité des architectures des systèmes distribués. Ces contextes applicatifs nous permettent de couvrir différents types d'architectures des systèmes distribués et différents modèles de communication. Les services à messages ont des architectures simples composées d'un système unique potentiellement dupliqué et reposant sur des communications asynchrones. Les services Internet multi-étagés ont des architectures en pipeline de systèmes dupliqués synchrones. Enfin, les services de surveillance des systèmes distribués sont des systèmes distribués asynchrones organisés selon des architectures hiérarchiques.

Diversité des contextes techniques. Notre contribution repose sur des contextes techniques variés. Nous implantons d'une part des systèmes autonomes auto-optimisés au moyen d'un gestionnaire spécialisé pour l'auto-optimisation des systèmes distribués au sein de la plate-forme d'administration autonome Jade. D'autre part, nous implantons un canevas spécialisé pour la construction de systèmes distribués fondés sur des architectures dynamiques dans le cadre de la plate-forme Mozart/Oz. Ces deux aspects de notre contribution reposent tout deux sur le modèle de composant Fractal.

Les gestionnaires d'auto-optimisation implantent diverses politiques d'optimisation des configurations des systèmes distribués administrés. Ces politiques visent donc à déterminer les configurations optimales des systèmes administrés, comme par exemple le degré de duplication du système, de chacun des étages qui le composent, ou encore les paramètres de l'arbre hiérarchique de son architecture (profondeur, arité, etc), etc.

Dans le chapitre suivant, nous proposons différentes politiques d'optimisation des systèmes distribués.

Chapitre 7

Politiques d'auto-optimisation

Nous proposons, dans ce chapitre, différentes politiques d'optimisation des systèmes distribués. Dans un premier temps, nous nous concentrons sur des politiques fondées sur des heuristiques qui améliorent le système administré au mieux. Ces heuristiques ne fournissent pas de garanties quant aux performances réelles obtenues. Nous nous intéressons ensuite aux politiques reposant sur des modélisations du système distribué administré et permettant ainsi d'établir des garanties de performances.

7.1 Heuristiques pour optimisation au mieux

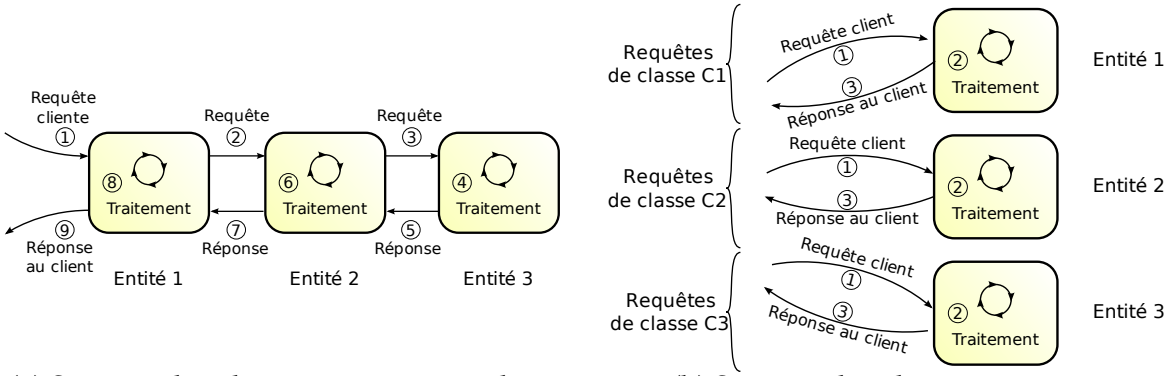
7.1.1 Contexte et hypothèses

Nous nous plaçons dans cette section dans le cadre de l'administration de systèmes distribués sur une grappe de machines interconnectées par un réseau local (LAN). Nous supposons que les machines composant la grappe sont homogènes et ont des caractéristiques physiques et logicielles identiques (mêmes composants physiques et mêmes systèmes d'exploitation), comme c'est typiquement le cas dans des grappes de machines en réseau local. Un système distribué est un système informatique fonctionnant sur un ensemble de machines interconnectées. Un système distribué peut dynamiquement intégrer ou relâcher des machines depuis un ensemble global de machines (par exemple, une grappe de machines). Une machine est alors ou bien disponible, ou bien exclusivement utilisée par un système distribué. En d'autres termes, deux systèmes distribués n'utilisent jamais les mêmes machines à un instant donné. Nous considérons en particulier des systèmes distribués implantant des services Internet ou des services à messages comme, par exemple, des services de messagerie, de commerce en ligne ou encore de diffusion de contenus, etc.

Les systèmes distribués peuvent être considérés ou bien comme des entités monolithiques, ou bien comme des assemblages de plusieurs entités. Par exemple, un service de messagerie peut être observé comme une entité monolithique représentant le serveur d'e-mail. À l'inverse, un service Internet multi-étagé peut être observé comme l'association de trois entités : l'entité représentant le serveur Web, l'entité représentant

le serveur d'applications et enfin l'entité représentant le serveur de bases de données. Ici, une entité désigne toute partie d'un système distribué pouvant être hébergée sur une machine distincte.

De plus, l'architecture des systèmes distribués, c'est-à-dire la manière avec laquelle les entités s'organisent, peut prendre différentes formes. Nous distinguons d'une part les systèmes organisés en pipeline, et d'autre part les systèmes partitionnés. Dans un système distribué en pipeline, les entités s'organisent en série (c.f. figure 7.1-(a)), et chacune des entités peut prendre part au traitement de la requête d'un client. Par exemple, dans un service Internet multi-étagé en trois étages (le serveur Web, le serveur d'applications et le serveur de bases de données), les trois entités coopèrent au traitement et à la construction de la réponse renvoyée au client : le serveur de bases de données exécute des requêtes sur les bases de données abritant l'état du service Internet, le serveur d'applications s'appuie sur ces opérations pour implanter la logique applicative du service, et enfin le serveur Web construit les pages HTML qui sont renvoyées au client. À l'inverse, dans un système distribué partitionné, les entités qui composent le système s'organisent en parallèle (c.f. figure 7.1-(b)) et n'interagissent pas les unes avec les autres. Par exemple, dans le cas d'un service composé de N entités, chacune responsable du traitement d'une classe particulière de requêtes C_i ($1 \leq i \leq N$), les requêtes des clients sont réparties en fonction de leur classe entre les différentes entités, et chaque requête est traitée par une entité unique. Les systèmes distribués peuvent être construits en combinant plusieurs systèmes en pipeline et plusieurs systèmes partitionnés.

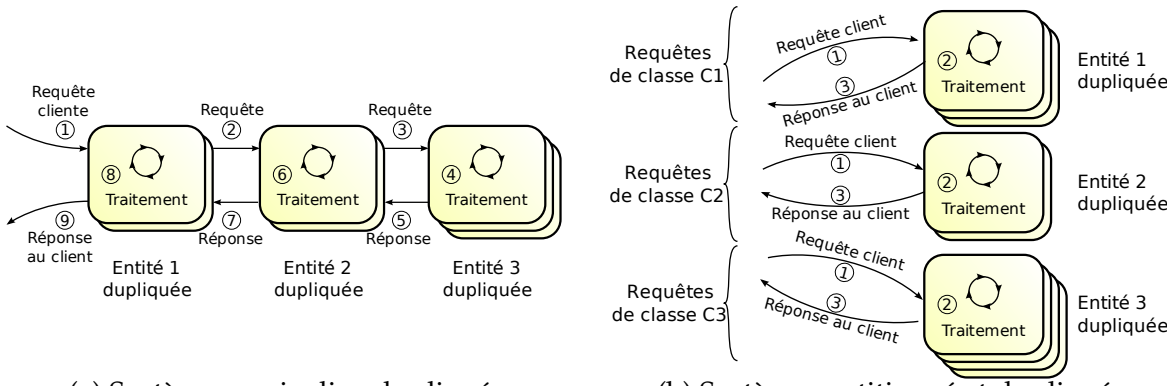


(a) Système distribué organisé en pipeline. (b) Système distribué partitionné.

FIG. 7.1 – Architectures des systèmes distribués en pipeline ou partitionnés.

Enfin, pour des raisons de passage à l'échelle, afin notamment de traiter toujours plus de requêtes clients, les entités peuvent être dupliquées sur plusieurs machines (c.f. figure 7.2). La duplication est associée à l'utilisation d'un répartiteur de charge dont le rôle est de répartir la charge de travail soumise au système sur les différents exemplaires de l'entité dupliquée qui le composent. Il existe de nombreux algorithmes de répartition de charge qui peuvent être utilisés ici [42, 48, 24]. Nous considérons dans la suite que l'algorithme de répartition de charge utilisé répartit équitablement la charge (en termes de niveaux de consommation des ressources) sur l'ensemble des machines

d'un système dupliqué.



(a) Système en pipeline dupliqué. (b) Système partitionné et dupliqué.

FIG. 7.2 – Architectures des systèmes avec duplication.

7.1.2 Principes de conception

Nous proposons une technique d'optimisation des systèmes qui repose sur l'approvisionnement dynamique des systèmes administrés. Cela consiste à modifier dynamiquement la quantité de ressources allouées au système afin de l'adapter aux besoins instantanés et futurs du système.

Gestionnaire d'auto-optimisation

L'auto-optimisation prend la forme d'un gestionnaire chargé d'appliquer une politique d'auto-optimisation sur un système administré donné. Dans notre contexte, l'auto-optimisation repose sur l'approvisionnement dynamique d'un système dupliqué sur une grappe de machines. L'approvisionnement dynamique consiste à adapter dynamiquement la quantité de ressources allouées au système administré, c'est-à-dire à ajouter ou retirer des machines au système durant son exécution.

Nous fondons notre approche pour la réalisation des gestionnaires d'auto-optimisation sur l'architecture du système administré. Nous modélisons l'architecture des systèmes distribués administrés afin d'identifier différents patrons autorisant des optimisations dont, par exemple, les systèmes en pipeline ou partitionnés. En particulier, nous identifions et modélisons les systèmes distribués dupliqués en grappe pour lesquels nous implantons l'auto-optimisation par approvisionnement dynamique.

Nous concevons le gestionnaire d'auto-optimisation en trois parties : (1) l'observation du système distribué administré à l'exécution afin de détecter les violations des niveaux de qualité de service, (2) la planification architecturale de la capacité optimale du système administré en fonction de son état actuel, et enfin (3) la reconfiguration dynamique du système administré par approvisionnement dynamique pour rendre effective la nouvelle capacité planifiée du système.

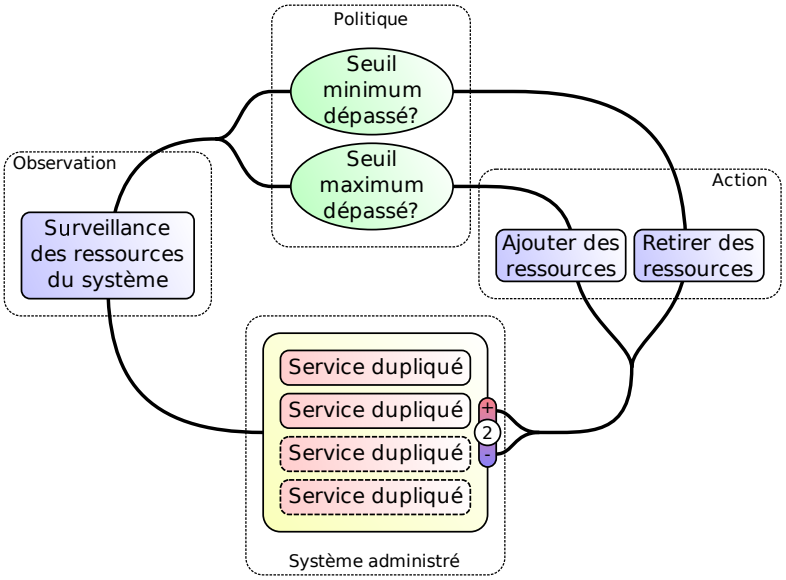


FIG. 7.3 – Auto-optimisation par dimensionnement dynamique.

La figure 7.3 illustre l'organisation générale d'un système auto-optimisé. Le système distribué administré à auto-optimiser est modélisé et expose une interface d'approvisionnement dynamique. L'état de chaque ressource composant le système distribué est surveillé afin de déterminer si le système administré est surchargé ou sous-chargé. En appliquant des seuils sur les niveaux de consommation des ressources, le gestionnaire décide d'ajouter ou de retirer une ou plusieurs machines au système administré. Ces ajouts ou retraits de ressources sont enfin mis en œuvre en exploitant l'interface d'approvisionnement dynamique du système administré. Nous détaillons maintenant la modélisation du système administré ainsi que les trois parties du gestionnaire d'auto-optimisation.

Système administré modélisé sous forme d'un composant redimensionnable. On modélise le système administré, que l'on suppose capable de tirer parti d'une quantité de ressource arbitraire, sous forme d'un composant générique dit *redimensionnable*. La figure 7.4 présente l'architecture générale du composant redimensionnable qui est principalement composé des différents exemplaires du système dupliqué. Le nombre d'exemplaires du système dupliqué représente la taille du système, et donc indirectement sa capacité de traitement. Le composant redimensionnable peut éventuellement faire apparaître sa logique de répartition de charge et de gestion de cohérence, afin par exemple d'exposer des capacités de reconfiguration étendues. Enfin, le composant redimensionnable expose une interface de contrôle de dimensionnement permettant d'adapter la taille du système, c'est-à-dire de contrôler le nombre d'exemplaires du système dupliqué.

L'interface de contrôle du dimensionnement repose essentiellement sur les primi-

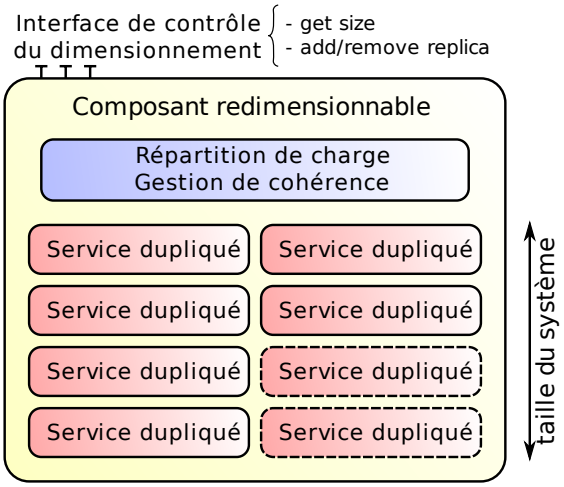


FIG. 7.4 – Composant redimensionnable.

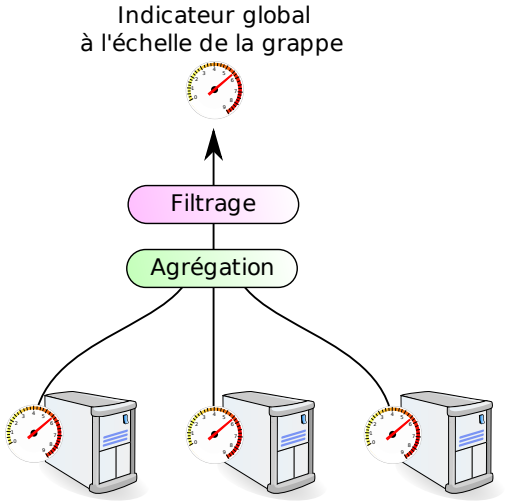
tives addReplica et removeReplica permettant l'ajout et le retrait d'un ou de plusieurs exemplaires du système dupliqué. La forme Java de cette interface est la suivante :

```
public interface ResizableComponent {
    int getSize();

    void addReplica(Component comp);
    void addReplicas(Component[] comps);

    Component removeReplica();
    Component[] removeReplicas(int n);
}
```

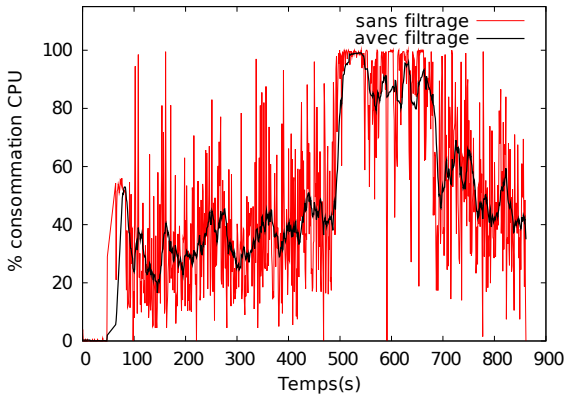
Avec les hypothèses formulées concernant les systèmes distribués administrés, l'interface de contrôle du dimensionnement du composant permet de contrôler directement la quantité de ressources mobilisées, et donc la capacité de traitement du système redimensionnable. L'interface de contrôle du dimensionnement est générique et peut être mise en œuvre pour tout système satisfaisant les hypothèses précédentes.

Observation du système. Il s'agit ici de construire un système permettant de surveiller l'état d'un composant redimensionnable. Rapporté aux hypothèses que nous avons formulées au sujet des systèmes modélisés sous forme de composants redimensionnables, cela revient à surveiller l'état de la grappe des machines qui hébergent les instances du service dupliqué, afin de déterminer le niveau de charge du composant redimensionnable. Aussi, comme illustré dans la figure 7.5-(a), le système que l'on met en œuvre consiste à observer le niveau de consommation des ressources matérielles (processeur, mémoire, bande passante réseau ou disque, etc) de chacune des machines hébergeant une instance du service dupliqué. Ces observations peuvent prendre la forme d'une surveillance en ligne de l'état des ressources du système, ou bien de prédictions

sur l'utilisation future des ressources du système. Il s'agit d'auto-optimisation *réactive* dans le premier cas, et d'auto-optimisation *pro-active* dans le second cas. Pour obtenir une vue globale de la consommation des ressources dans un composant redimensionnable, ces informations sont ensuite agrégées. Cette agrégation est valide avec l'hypothèse d'homogénéité des machines et de répartition équitable de la charge de travail soumise au composant redimensionnable sur l'ensemble des instances du service dupliqué et avec l'hypothèse.



(a) Structure du service de surveillance du système administré redimensionnable.



(b) Illustration de l'importance du filtrage des indicateurs pour la surveillance du système.

FIG. 7.5 – Service de surveillance du système administré.

L'observation du système est une tâche sensible, car c'est elle qui va essentiellement déterminer la qualité de la boucle de commande, sa sensibilité et sa réactivité aux changements d'état du système administré. Le niveau de charge global fait l'objet d'un filtrage dont l'objectif est de minimiser les artéfacts et d'améliorer la qualité de la détection des situations de surcharges ou de sous-charge. La figure 7.5-(b) illustre l'intérêt du filtrage sur le niveau de consommation du processeur. En effet, l'information brute d'utilisation d'un processeur est une indication extrêmement variable et fugace, de par la nature même du fonctionnement du processeur. Le filtrage utilisé dans cet exemple est une moyenne glissante pondérée par des exponentielles (exponentially weighted moving average, ou EWMA [65]). L'opérateur EWMA présente l'avantage d'être peu coûteux et d'être simple à mettre en œuvre et à paramétrer (au moyen d'un unique coefficient déterminant l'importance relative donnée aux informations en fonction de leur âge). Ainsi, l'EWMA permet de minimiser l'impact des artéfacts en donnant un plus faible poids aux informations récentes, limitant ainsi les erreurs de détection. Toutefois, cela se traduit par une perte de réactivité de l'opérateur en cas de changement réel brusque. Il s'agit donc lors du paramétrage de l'opérateur, de trouver le bon com-

promis entre fausse détection et réactivité.

Politique d'auto-optimisation. Le cœur de l'auto-optimisation est réalisé par un algorithme qui implante une politique d'auto-optimisation. L'algorithme s'appuie sur des informations relatives aux taux d'utilisation des ressources et rapportées par le service de surveillance. Par application de seuils sur ces informations, l'algorithme décide de la conformité ou non de l'état du système par rapport à ses contraintes de performances, et déclenche alors d'éventuelles actions à mettre en œuvre pour ramener le système dans un état conforme à ses contraintes de performances. Pour chacune des ressources matérielles observées (processeur, mémoire, etc), deux seuils sont fixés : un seuil minimum de consommation, en deçà duquel la machine est sous-exploitée pour cette ressource, et un seuil maximum de consommation au-delà duquel la machine est surchargée pour cette ressource.

Algorithme 1 Politique d'approvisionnement dynamique basée sur des seuils.

on receive(event : MonitoringEvent) :
if (event.consumption$_1$ > max_threshold$_1$)
  or (event.consumption$_2$ > max_threshold$_2$)
  or ...
  or (event.consumption$_r$ > max_threshold$_r$) **then**
  // One of the hardware resources is overloaded, thus the system is overloaded and needs more replicas
  n := CapacityPlanning() *// Determine how many new replicas are needed*
  AddReplicas(n) *// Deploy n new replicas*
else if (event.consumption$_1$ < min_threshold$_1$)
  and (event.consumption$_2$ < min_threshold$_2$)
  and ...
  and (event.consumption$_r$ < min_threshold$_r$) **then**
  // All hardware resources are under-loaded, thus the system is under-loaded and wastes replicas
  n := CapacityPlanning() *// Determine how many replicas are wasted*
  RemoveReplicas(n) *// Undeploy n replicas*
end if

L'algorithme 1 en présente la forme générale. L'algorithme décide que le système est surchargé lorsque les informations qui lui sont rapportées indiquent que l'une des ressources du système a un taux d'utilisation qui excède un seuil maximum prédéterminé. Symétriquement, l'algorithme décide que le système est sous-chargé lorsque toutes les ressources sont indiquées avec un taux d'utilisation inférieur à un seuil minimum prédéterminé.

Action sur le système. Les actions sur le système consistent à mettre en pratique l'approvisionnement dynamique, c'est-à-dire à exploiter l'interface de dimensionnement

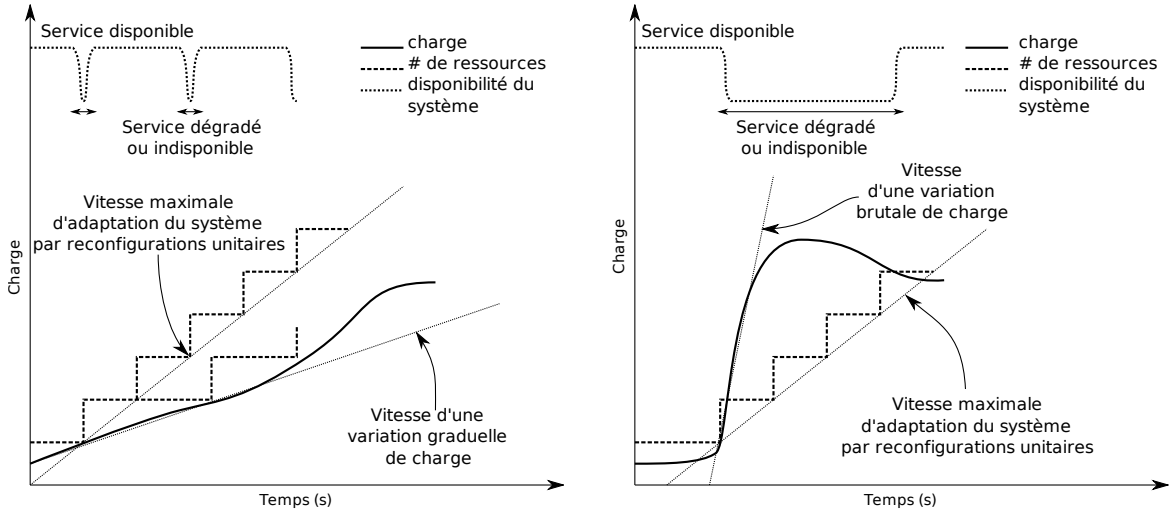
du composant redimensionnable. Par exemple, nous mettons à disposition de l'algorithme d'auto-optimisation des actions unitaires qui déclenchent l'ajout ou le retrait d'une machine au sein du système dupliqué. D'autres actions permettent l'ajout et le retrait de plusieurs machines simultanément en une opération.

Notons que le contrôle du dimensionnement du système administré ne permet pas, a priori et sans plus d'information, de déterminer à l'avance la capacité de traitement du système, mais indique simplement le nombre d'instances du service dupliqué, c'est-à-dire le nombre de machines allouées au système. Avec nos hypothèses, cela garantit toutefois que l'ajout (resp. le retrait) d'un exemplaire du système dupliqué accroit (resp. réduit) la capacité de traitement. Le lien exact entre la taille du système redimensionnable et sa capacité de traitement effective nécessite des informations supplémentaires concernant d'une part les caractéristiques physiques des machines et d'autre part le comportement du système applicatif, et notamment sa réponse au passage à l'échelle. Ces informations peuvent être obtenues au moyen d'une modélisation du système, ou encore au moyen d'une campagne de mesures réalisées sur un banc d'essai et permettant, par exemple, d'en construire des profils.

Variation dynamique de charge

Les situations de surcharge ou de sous-charge sont déclenchées par des variations dans la charge de travail appliquée au système administré. Ces variations peuvent intervenir selon deux scénarios. Ou bien la charge de travail évolue lentement et faiblement, instaurant progressivement une situation de surcharge ou de sous-charge dans le système, comme illustré sur la figure 7.6-(a). Ce type de variations est le plus fréquent et se produit principalement par l'influence des habitudes et comportements cycliques (journaliers, hebdomadaires, etc) des utilisateurs des systèmes, comme par exemple, la consultation matinale des e-mails. Ou bien la charge de travail évolue brutalement, à la manière d'un « pic de charge » plaçant instantanément le système en situation de forte surcharge ou sous-charge, comme illustré sur la figure 7.6-(b). Les variations de ce type, aussi appellées «effet *Slashdot*», sont plus rares et imprévisibles, et sont essentiellement liées à de grands évènements comme des exploits sportifs, des lancements commerciaux ou encore des élections ou attentats, etc [5].

Le critère de distinction que nous adoptons pour différencier une variation graduelle de charge d'un pic de charge fait intervenir la vitesse maximale d'adaptation du système administré par reconfigurations unitaires. Cette vitesse maximale est déterminée par la latence des opérations de reconfiguration unitaires, qui consistent à ajouter ou enlever une ressource à la fois au système administré. Les variations de charge qui surviennent plus lentement que la vitesse maximale d'adaptation du système administré sont graduelles, car l'optimisation du système par des adaptations unitaires suffit à en maintenir l'état. Symétriquement, les variations de charge qui surviennent plus rapidement que cette vitesse maximale sont brutales et sont assimilées à des pics de charge. Pour traiter un pic de charge, nous nous appuyons sur des heuristiques pour



(a) Variation graduelle de charge (b) Variation brutale de charge

FIG. 7.6 – Caractérisation d'une variation progressive ou brutale de la charge par rapport à la vitesse maximale d'adaptation d'un système par reconfigurations unitaires.

déterminer les besoins en ressources du système administré, afin d'accélérer le processus d'adaptation du dimensionnement du système.

Variation graduelle de charge. Lorsque le système administré est sujet à des variations graduelles de charge, de simples ajustements du dimensionnement du système par ajouts et retraits unitaires de ressources suffisent pour absorber les changements progressifs de l'état du système. L'hypothèse d'une variation graduelle de la charge garantit en effet que les ajustements unitaires du dimensionnement du système administré auront lieu plus rapidement que les variations de charge, permettant ainsi d'adapter la capacité de traitement du système à ses besoins réels. Nous concevons un gestionnaire d'auto-optimisation capable de gérer les variations graduelles de charge. Ce gestionnaire s'appuie sur la surveillance des niveaux de consommation des ressources physiques du système administré (comme le processeur, la mémoire, la bande passante réseau ou disque, etc). Sur la base de ces observations, le gestionnaire peut alors décider d'un ajout ou d'un retrait de ressource unitaire, c'est-à-dire d'approvisionner ou de désapprovisionner le système administré d'une machine au moyen des primitives unitaires de l'interface de dimensionnement.

Pic de charge Lorsque le système administré est soumis à des variations brutales de charge, l'ajustement du dimensionnement du système peut nécessiter l'ajout ou le retrait immédiat de plusieurs ressources afin d'absorber le pic de charge. Il devient alors intéressant de factoriser et de paralléliser les ajouts ou les retraits de ressources du système administré afin de réduire la latence totale des opérations de reconfiguration du système. Le défi consiste alors à déterminer aussi précisément que possible le di-

mensionnement optimal du système, c'est-à-dire le nombre de machines à ajouter ou à retirer au système administré. Nous concevons un gestionnaire d'auto-optimisation fondé sur ce principe et chargé de traiter les variations brutales de la charge du système administré. Ce gestionnaire s'appuie sur la surveillance des ressources physiques, mais aussi sur une surveillance de niveau applicatif (comme par exemple le nombre de transactions actuellement en exécution en concurrence dans le serveur). Cette surveillance applicative nous permet d'établir une fonction heuristique capable de déterminer le dimensionnement optimal du système administré selon l'observation de son état. En particulier, nous identifions une fonction heuristique satisfaisante déterminant le dimensionnement optimal du système administré proportionnellement au nombre de transactions en cours de traitement. Grâce à cette estimation, les machines sont ajoutées ou retirées du système en parallèle.

Oscillations du système

L'auto-optimisation du système administré peut induire des périodes d'instabilité durant lesquelles les informations rapportées par les sondes réalisant la surveillance du système ne sont pas pertinentes. L'interprétation de ces informations non pertinentes peut conduire à des décisions d'optimisation erronées.

Latence des reconfigurations et stabilisation. Les reconfigurations dynamiques opérées par le gestionnaire d'auto-optimisation peuvent induire des périodes d'instabilité pendant lesquelles le comportement du système distribué administré est transitoirement erratique. Par exemple, une reconfiguration dynamique ayant pour objectif l'installation et la mise en service d'un serveur de bases de données sera suivie d'une période durant laquelle les performances du serveur de bases de données seront diminuées, notamment en raison du temps de chauffe du serveur pendant lequel celui-ci charge ses mémoires caches. Durant ces périodes transitoires, les informations rapportées par le service de surveillance du système administré ne sont pas pertinentes pour l'auto-optimisation du système dans la mesure où elles représentent un état instable passager. Il est donc nécessaire de coordonner l'observation du système administré avec les opérations de reconfigurations, afin de prévenir les reconfigurations en cascade qui peuvent être déclenchées par l'interprétation des états transitoires du système.

Reconfigurations concurrentes et coordination. Par ailleurs, l'approvisionnement dynamique d'un service Internet peut occasionner de multiples reconfigurations concurrentes en réalité non nécessaires sur différentes parties du système, et dont l'exécution vient au final perturber les performances générales du système administré. Par exemple, dans le cas d'un service Internet multi-étagé organisé en un pipeline comportant un serveur Web frontal et un serveur de bases de données, le serveur de base de données peut devenir le facteur limitant du système et induire alors une situation de sous-charge sur le serveur Web (qui sera alors bloqué en attente des réponses

aux requêtes soumises au serveur de bases de données) ; dans un pareil cas, l'auto-optimisation peut décider d'accroître les ressources allouées au serveur de bases de données et en même temps de réduire celles allouées au serveur Web frontal. Cette dernière réduction de ressources sur le serveur Web frontal est une conséquence de la dépendance entre le serveur Web frontal et le serveur de bases de données, qui conduit à des opérations inutiles d'approvisionnement dynamique, et donc à des oscillations.

Gestion des oscillations. Nous proposons une technique qui s'appuie sur la connaissance de l'architecture logicielle du système distribué administré et des inter-dépendances entre les différentes parties du système administré pour prévenir l'occurence de telles oscillations dans les systèmes administrés. La gestion des oscillations repose sur une description du système qui permet au gestionnaire de déterminer les dépendances entre les différentes parties du système administré. Plus précisément, le gestionnaire exploite l'information contenue dans l'architecture des systèmes administrés pour en extraire les dépendances fonctionnelles qui caractérisent les inter-dépendances entre les parties des systèmes administrés. L'architecture du système permet notamment de détecter les parties organisées en pipeline caractérisant leurs dépendances, et les parties organisées en partitions caractérisant leur indépendance.

Le gestionnaire d'auto-optimisation est alors capable d'identifier les parties inter-dépendantes des systèmes administrés. Afin d'éviter les phénomènes d'oscillations, le gestionnaire s'assure qu'au cours d'une opération d'auto-optimisation ayant lieu sur un système donné, l'auto-optimisation est inhibée sur toutes les parties inter-dépendantes du système en cours d'optimisation pendant un délai prédéterminé. Une fois le délai d'inhibition expiré, les opérations d'auto-optimisation sont de nouveau autorisées sur ces parties du système administré.

7.1.3 Mises en œuvre et validation dans des systèmes réels

Cette section décrit les mises en œuvre concrètes des différentes politiques d'auto-optimisation présentées au cours de la section précédente, et ce dans le cadre de la construction de services auto-optimisés reposant sur les contextes applicatifs suivants : les services à messages et les services Internet multi-étagés.

Auto-optimisation d'un service à messages

Nous présentons ici les détails de la réalisation d'un service à messages auto-optimisé reposant sur le standard JMS et tirant parti de l'extension offerte par les serveurs JORAM permettant d'assembler des grappes de files d'attente, comme illustré sur la figure 7.7 [33]. Le système administré que nous considérons ici est donc un service à messages en grappe que l'on dote d'auto-optimisation par approvisionnement dynamique. Il s'agit ici d'un système distribué monolithique, dupliqué en grappe et reposant sur des communications asynchrones. L'architecture de ce système distribué correspond donc précisément à celle d'un composant redimensionnable, et son optimisation par appro-

visionnement dynamique vise ainsi à ajuster précisément le nombre des files d'attente qui participent au composant redimensionnable.

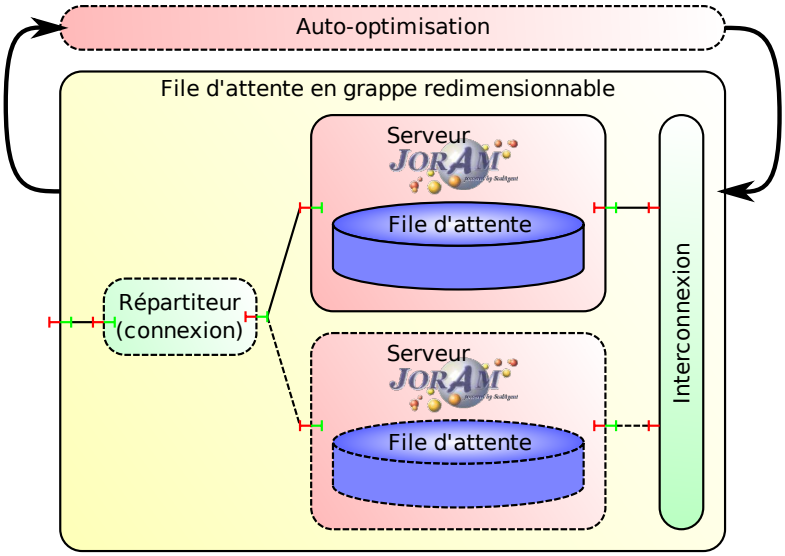


FIG. 7.7 – Implantation d'un service à messages dynamique en grappe.

Le service à messages administré est soumis à une surveillance continue permettant d'observer les niveaux de consommation des ressources matérielles (processeur, mémoire, etc) de chacune des machines hébergeant les files d'attente composant la file d'attente en grappe. En pratique, nous considérons ici uniquement la ressource processeur, car il apparaît que c'est en réalité la seule ressource matérielle susceptible de devenir un facteur limitant de ce système distribué. Les niveaux de consommation des processeurs des différentes machines sont agrégés et moyennés (EWMA) afin de synthétiser un niveau de consommation global de la ressource processeur sur l'ensemble du composant redimensionnable qui modélise le service à messages administré.

Nous implantons un gestionnaire d'auto-optimisation traitant les variations graduelles de charge par approvisionnement dynamique du composant redimensionnable. L'approvisionnement dynamique consiste en des ajouts ou des retraits dynamiques et unitaires de files d'attente dans la grappe de files d'attente.

Nous implantons la gestion des oscillations du système au moyen d'un délai de garde interdisant toute nouvelle reconfiguration directement après une reconfiguration. Une fois le délai de garde expiré, les reconfigurations par approvisionnement dynamique sont à nouveau autorisées. Cette politique est conforme à la technique de prévention des oscillations établie précédemment et découle de l'architecture du service à messages en grappe qui est composée d'un système unique dupliqué et sans dépendances.

Auto-optimisation d'un service Internet multi-étagé

De façon similaire, nous implantons les politiques d'auto-optimisation détaillées précédemment dans le cadre de la construction d'un service Internet multi-étagé dynamiquement approvisionné. Le service Internet que nous mettons en œuvre est un service fondé sur l'architecture J2EE et se compose de deux étages comme illustré sur la figure 7.8 : le premier étage est constitué de serveurs de Servlets Tomcat en grappe qui réalisent la partie Web et application du service Internet [49] ; le second étage est contitué de serveurs de bases de données MySQL en grappe qui gèrent la persistence des données du service Internet [1]. Il s'agit donc d'optimiser un système distribué organisé en pipeline de systèmes dupliqués en grappes, et reposant sur des communications synchrones. Plus concrètement, le système est formé d'un pipeline de composants redimensionnables, et nous implantons un gestionnaire d'auto-optimisation par approvisionnement dynamique pour chacun des deux composants redimensionnables constituant ce service Internet.

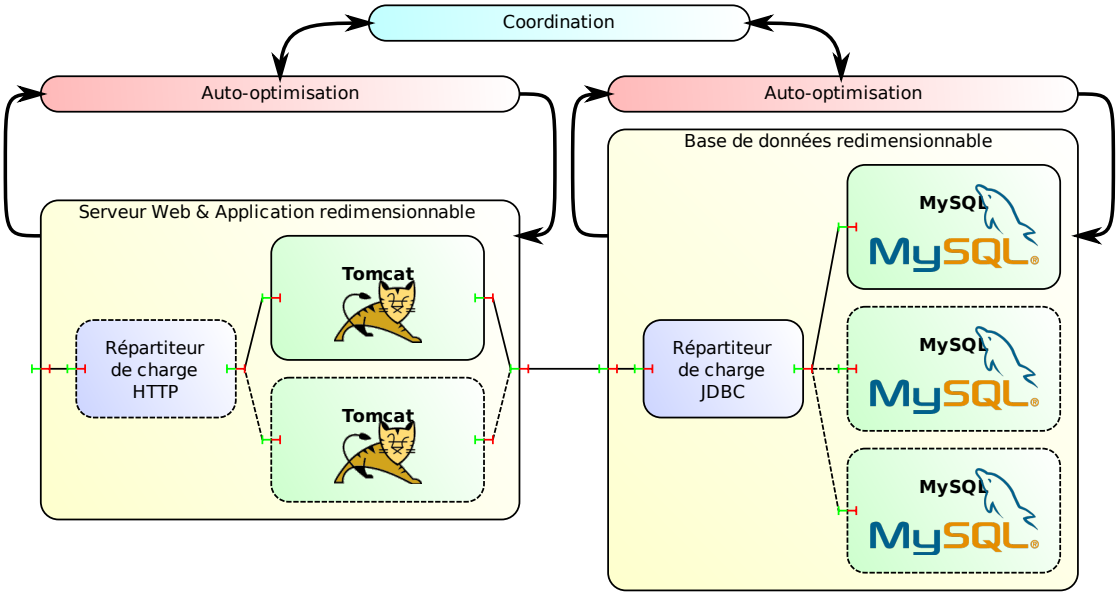


FIG. 7.8 – Implantation d'un service Internet dynamique en grappe.

Chaque composant redimensionnable représentant un étage du service Internet fait l'objet d'une surveillance. Nous observons les niveaux de consommation des ressources matérielles des machines hébergeant les instances des systèmes dupliqués, et pour les mêmes raisons que précédemment, nous nous limitons à l'observation de la ressource processeur. Les niveaux de consommation des processeurs des différentes machines sont agrégés et moyennés (EWMA) afin de synthétiser un niveau de consommation global de la ressource processeur de chacun des composants redimensionnables. Nous implantons également une sonde de niveau applicatif, spécifique à l'étage correspondant au serveur de bases de données dupliqué. Cette sonde permet de surveiller en

continu le nombre de transactions qui s'exécutent à un instant donné dans le serveur de bases de données. Cette information complète le niveau de consommation des processeurs des serveurs de bases de données et permet d'obtenir une indication plus précise sur le niveau de charge effectif de l'étage bases de données du service Internet administré.

Nous mettons en œuvre deux politiques d'auto-optimisation. La première vise à traiter les variations dynamiques graduelles de la charge soumise au service Internet. Elle est fondée sur un seuil minimal et un seuil maximal du niveau de consommation de la ressource processeur des composants redimensionnables, et déclenche des ajouts ou retraits dynamiques unitaires des serveurs Tomcat ou MySQL dans les composants redimensionnables.
La seconde politique d'auto-optimisation cible la gestion des pics de charge. Elle est fondée sur l'observation de la ressource processeur ainsi que du nombre de transactions en cours d'exécution dans le serveur de bases de données du service Internet. Cette dernière information couplée à une fonction heuristique permet de déterminer instantanément le dimensionnement optimal de l'étage bases de données. En cas de pic de charge, cette politique d'auto-optimisation déclenche directement et en une seule fois l'ajout du nombre nécessaire et suffisant de serveurs MySQL permettant de satisfaire les requêtes clients et d'absorber ainsi le pic de charge.

L'architecture du service Internet est formée d'un pipeline de deux composants redimensionnables, équipés chacun d'un gestionnaire d'auto-optimisation. Cette architecture montre donc une dépendance entre les deux composants redimensionnables. Ainsi, conformément avec la technique de prévention des oscillations présentée précédemment, toute reconfiguration intervenant sur l'un des composants redimensionnables déclenche un délai de garde interdisant toute nouvelle reconfiguration sur lui-même ainsi que sur l'autre composant redimensionnable. Après expiration du délai de garde, les reconfigurations sont à nouveau autorisées sur les deux composants.

Nous avons présenté différentes politiques d'auto-optimisation permettant l'amélioration des performances des systèmes distribués administrés en s'appuyant sur des algorithmes heuristiques. Ces algorithmes sont intéressants pour leur simplicité et leur généralité. Toutefois, ils ne permettent pas d'établir des garanties sur les performances du système administré. Nous étudions dans la section qui suit une technique fondée sur la modélisation du système administré et qui permet d'assurer des garanties de performances optimales.

7.2 Modélisation pour garantie d'optimalité

Notre précédente approche améliore les performances du système en s'appuyant sur des techniques à base d'heuristiques. Notre objectif est ici de proposer une technique pour configurer le système administré afin de pouvoir en garantir les performances.

7.2.1 Principes de conception

Nous fondons notre approche sur l'utilisation d'une modélisation de performance du système administré. Cette modélisation prend en compte les caractéristiques architecturales du système ainsi que son paramétrage. Par inversion du modèle obtenu, nous déterminons une architecture optimale qui réalise les contraintes de performance formulées sur le système administré.

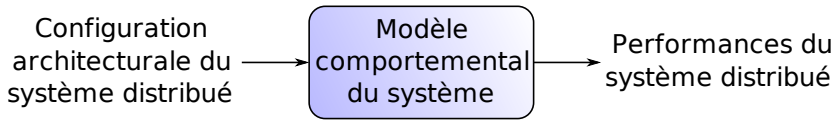


FIG. 7.9 – Modélisation des performances d'un système en fonction de la configuration (architecture et/ou paramétrage local).

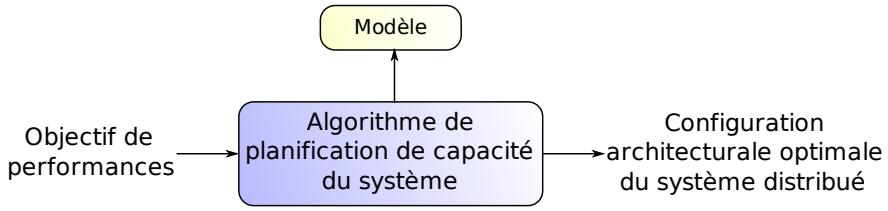


FIG. 7.10 – Inversion du modèle de performance.

7.2.2 Réalisations

Système distribué réel considéré

Notre service de surveillance s'appuie sur des sondes systèmes présentes sur chacune des machines qui composent le système administré et qui récoltent des informations concernant l'état des machines et notamment les niveaux d'utilisation des ressources matérielles (processeur, mémoire, réseau, disque dur, etc). Le service de surveillance est chargé d'extraire certaines informations à l'échelle de la grappe de machines (comme par exemple les taux minimal, maximal et moyen d'utilisation des processeurs sur l'ensemble des machines), et qui sont agrégées selon une architecture hiérarchique des sondes (c.f. figure 7.11). Les sondes fonctionnent indépendamment les unes des autres et selon un schéma asynchrone. Une sonde récolte périodiquement les informations sur la machine qui l'abrite. La sonde récolte également de manière asynchrone les comptes-rendus d'autres sondes avec lesquelles elle est connectée selon l'architecture hiérarchique du service de surveillance. Enfin, périodiquement, la sonde agrège toutes les informations dont elle dispose à un instant donné sous forme d'un compte-rendu qu'elle transmet à la sonde qui lui est hiérarchiquement supérieure.

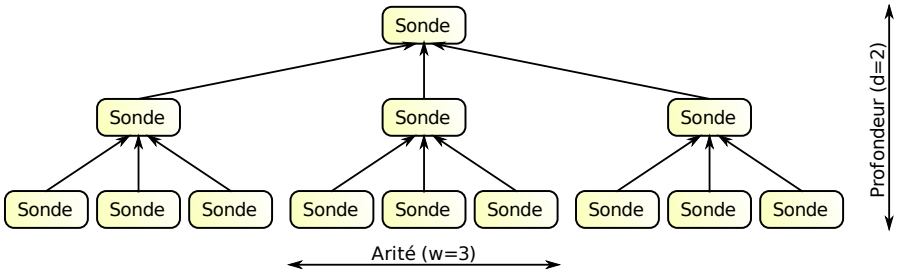


FIG. 7.11 – Exemple d'architecture hiérarchique du service de surveillance réel considéré.

Caractérisation de la configuration architecturale du système distribué

L'architecture du système distribué a une forme hiérarchique arborescente comme celle donnée en exemple sur la figure 7.11. L'arbre des sondes qui composent ce système est entièrement caractérisé par les trois paramètres suivants :

w l'arité de la structure d'arbre selon laquelle sont organisées les sondes ;

d la profondeur de l'arbre ;

N le nombre de nœuds de l'arbre (constante donnée) ;

Caractérisation de la configuration locale du système distribué

Chaque sonde mesure périodiquement l'état des différentes ressources sur la machine qui l'héberge, puis transmet un message contenant un compte-rendu à une autre sonde. Une sonde est ainsi localement caractérisée par les paramètres suivants :

τ l'intervalle de temps séparant deux mises à jour consécutives des mesures par la sonde ;

s la taille des messages émis par la sonde.

Caractérisation des performances du système distribué

Enfin, nous considérons deux critères de performances du service de surveillance :

L la latence maximale avec laquelle une information est disponible à la racine hiérarchique du service de surveillance ;

B la bande passante réseau consommée par le service de surveillance sur les machines du système distribué ;

Nous considérons dans la suite deux objectifs d'optimisation distincts, lesquels guident la recherche de configurations optimales du service de surveillance. Nous mettons en œuvre cette technique pour optimiser notre service de surveillance distribué des systèmes administrés et des grappes de machines qui les hébergent. Nous modélisons ce système en capturant ses différentes caractéristiques du service de surveillance au

moyen des informations suivantes. Notre objectif est de fournir une garantie de qualité de service du service de surveillance, en calculant une architecture et un paramétrage optimal afin d'atteindre l'un des deux objectifs d'optimisation suivants.

Configuration architecturale optimale

Sans entrer dans les détails, nous présentons ici les résultats de la modélisation, qui nous permettent de satisfaire les objectifs formulés précédemment de deux manières : ou bien nous formulons une contrainte sur les paramètres du système, c'est-à-dire dans notre cas l'intervalle de rafraîchissement des mesures τ, et le modèle nous permet de déduire une architecture optimale caractérisée par ses paramètres w et d ; ou bien nous formulons une contrainte architecturale sur w et d, et le modèle nous permet d'en déduire un paramétrage optimal τ.

Garantie relative à la consommation de bande passante réseau : Optimiser la réactivité en minimisant la latence de la disponibilité des informations L^* au nœud central, tout en respectant une contrainte de consommation de la bande passante réseau maximale $B_{\max}$.

$(L^*, d^*, w^*) =$ **modèle**$(N, \tau, B_{\max})$: Lorsque nous ajoutons une contrainte sur le paramètre τ (intervalle de rafraîchissement), le modèle nous permet d'extraire l'architecture optimale caractérisée par w et d.

$$w^* = \left\lfloor \frac{\tau}{s} B_{\max} - 1 \right\rfloor$$

Nous déduisons alors d en fonction de N et de w ; et la latence obtenue est bornée par $L^* = (d+1) \times \tau$.

$(L^*, \tau^*) =$ **modèle**$(N, d, w, B_{\max})$: Lorsque nous ajoutons une contrainte architecturale (sur w et d), le modèle nous permet d'extraire le paramétrage optimal τ.

$$\tau^* = \frac{(w+1)s}{B_{\max}}$$

La latence obtenue est bornée par $L^* = (d^*+1) \times \tau^*$.

Garantie relative à la latence de la disponibilité des informations : Optimiser l'empreinte du système de surveillance en minimisant la bande passante réseau consommée B^*, tout en respectant une contrainte de latence maximale de la disponibilité des informations sur le nœud central $L_{\max}$.

$(B^*, d^*, w^*) =$ **modèle**$(N, \tau, L_{\max})$: Lorsque nous ajoutons une contrainte sur le paramètre τ (intervalle de rafraîchissement), le modèle nous permet d'extraire l'architecture optimale caractérisée par w et d.

$$d^* = \left\lfloor \frac{L_{\max}}{\tau} - 1 \right\rfloor$$

Nous déduisons w^* en fonction de N et de d^* ; et la bande passante consommée est bornée par $B^* = (w+1)\frac{s}{\tau}$.

$(B^*, \tau^*) =$ **modèle**$(N, d, w, L_{\max})$: Lorsque nous ajoutons une contrainte architecturale (sur w et d), le modèle nous permet d'extraire le paramétrage optimal τ.

$$\tau^* = \frac{L_{\max}}{d+1}$$

La bande passante consommée est alors bornée par $B^* = (w+1)\frac{s}{\tau}$.

7.2.3 Reconfiguration dynamique

Le nombre de machines N devant être couverts par le système de surveillance évolue au cours de l'exécution du système administré. L'évolution de N doit être prise en compte pour adapter l'architecture et le paramétrage du service de surveillance afin d'une part d'intégrer les machines entrantes et les machines sortantes de l'architecture du système administré, et d'autre part d'assurer les garanties de performances exigées du service de surveillance.

7.3 Synthèse

Dans ce chapitre, nous avons présenté différentes politiques d'auto-optimisation qui répondent à l'ensemble des défis que nous avons identifiés pour l'auto-optimisation des systèmes distribués. Nous nous appuyons sur des boucles de commande implantant l'approvisionnement dynamique du système administré pour traiter les variations de charge. Un premier ensemble de politiques est fondé sur des algorithmes heuristiques permettant d'améliorer au mieux les performances des systèmes administrés. Ces politiques sont générales et réutilisables, mais ne fournissent pas de garantie sur les performances du système réellement obtenues. Afin d'établir des garanties de performance, nous présentons des politiques qui reposent sur des modélisations mathématiques des systèmes administrés. Par ailleurs, nous implantons un mécanisme de gestion des oscillations par l'analyse des dépendances dans l'architecture du système administré et reposant sur des délais d'inhibitions des reconfigurations dynamiques. Enfin, notre approche tolère les systèmes patrimoniaux par encapsulation des systèmes administrés dans des composants Jade exhibant une interface d'administration uniforme.

Les techniques fondées sur des heuristiques sont particulièrement intéressantes pour leur simplicité de mise en œuvre et pour leur grande généralité. Ces deux caractéristiques facilitent l'adaptation et l'implantation de ces politiques sur un grand nombre de systèmes distribués pourvu que ces systèmes soient représentés par des composants d'encapsulation Jade. Les politiques d'auto-optimisation par approvisionnement dynamique sont réutilisables dès lors que le système est modélisable sous forme d'un composant redimensionnable. La politique visant à réguler les variations de charge

graduelle est directement applicable lorsque les latences des reconfigurations sont supposées courtes devant l'intervalle de temps entre deux reconfigurations. La politique chargée d'absorber les pics de charge nécessite quant à elle la définition d'une fonction heuristique calculant la quantité de ressources nécessaire en fonction de la charge mesurée du système. D'autres politiques d'auto-optimisation peuvent être mises en œuvre en identifiant l'interface de reconfiguration adaptée. Par exemple, l'auto-optimisation par reconfiguration d'un paramètre local du système peut être obtenue en représentant ce paramètre local sous forme d'un attribut du composant d'encapsulation du système administré, et en exploitant l'interface de contrôle des attributs afin de commander les reconfigurations. Enfin, la complexité de ces politiques d'auto-optimisation par algorithmes heuristiques provient essentiellement de l'identification des sources d'information qui servent à piloter la politique (quelle ressource faut-il mesurer et surveiller ?) ainsi que du paramétrage des seuils déclenchant les reconfigurations.

Les techniques d'auto-optimisation fondées sur la modélisation du système administré permettent d'établir des garanties sur les performances du système administré. Ces techniques reposent sur un travail de modélisation du système administré, généralement complexe et très spécifique et qui doit être reconduit pour chaque système que l'on souhaite administrer. Toutefois, l'usage d'un modèle du système offre une grande précision sur le contrôle du système.

Une évaluation expérimentale des politiques d'auto-optimisation proposées suivra au chapitre 10.

Les mises en œuvre de ces diverses politiques d'auto-optimisation reposent sur des reconfigurations dynamiques des systèmes administrés permettant leur approvisionnement dynamique. Cela nous conduit à construire des systèmes dont l'architecture est dynamique et commandant des actions de reconfiguration potentiellement complexes. C'est précisément ce que nous abordons dans le chapitre suivant.

Chapitre 8

FructOz : un canevas pour la construction de systèmes distribués fondés sur des architectures dynamiques

Nous décrivons dans ce chapitre comment nous proposons d'exprimer les actions sur les systèmes administrés. Nous présentons dans un premier temps les principes de conception supportant les descriptions de telles actions. Nous décrivons ensuite l'implantation de ces principes dans le canevas FructOz spécialisé dans la construction de systèmes distribués fondés sur des architectures dynamiques. Enfin, nous présentons plusieurs exemples d'application immédiate de ces principes démontrant l'intérêt et l'expressivité du canevas FructOz.

8.1 Objectifs

Contexte et hypothèses. Nous nous intéressons ici à l'administration des systèmes distribués dans le cadre de l'administration fondée sur l'architecture. Les systèmes distribués considérés ici reposent donc sur des architectures distribuées. L'administration des systèmes distribués consiste en des actions qui reconfigurent dynamiquement leur architecture. Nous désignons par *architectures dynamiques* les systèmes distribués ainsi mis en œuvre et qui intègrent des actions réalisant des opérations d'adaptations dynamiques.

Problématique. L'expression des actions sur les systèmes distribués administrés est un problème complexe. Nous identifions plusieurs sources de complexité : (1) Ces actions comportent potentiellement de nombreuses opérations et sont distribuées. (2) Ces actions peuvent être associées à diverses contraintes, notamment liées au parallélisme, à la distribution et aux synchronisations nécessaires pour leur exécution. (3) Enfin, ces

actions peuvent être soumises à diverses conditions ou être paramétrées, par exemple afin de prendre en compte l'état de l'architecture au moment de l'exécution de l'action.

Analyse. Dans le contexte de l'administration fondée sur l'architecture, l'architecture des systèmes distribués administrés est représentée à l'aide de composants. Les composants dotent les systèmes distribués de la faculté d'être dynamiquement reconfigurés (adaptation dynamique). Néanmoins, le modèle de composant ne spécifie pas de modalité ni d'organisation pour l'expression de ces actions de reconfigurations. Les architectures dynamiques intègrent ces actions réalisant des adaptations dynamiques. Aussi, les descriptions d'architectures dynamiques incorporent les descriptions des actions de reconfigurations. Nous nous intéressons donc naturellement à la question plus vaste et plus générale de la description des architectures dynamiques, lesquelles comportent des descriptions d'actions sur les systèmes administrés.

Proposition. Nous proposons une construction langage pour la description des systèmes à architectures distribuées dynamiques. Notre construction langage permet de décrire une architecture dynamique, son processus de déploiement distribué ainsi que ses reconfigurations dynamiques sous forme d'une structure appelée *paquetage* (ou *package*). Nous assimilons la description des architectures dynamiques à la description de leurs processus de déploiement. De ce fait, nous contrôlons l'ensemble du processus de déploiement, ce qui permet d'y associer du paramétrage, des contraintes de distribution et d'exécution (workflow), etc. En quelque sorte, il faut considérer que nos systèmes à architectures dynamiques évoluent au cours d'un processus continu de (re-)déploiement concurrent et distribué au cours duquel chaque élément de l'architecture peut être sujet à des modifications liées à la progression du processus de déploiement dans son ensemble.

8.2 Principes de conception

Nous présentons dans cette section les choix de conception relatifs à notre proposition pour la construction et la description des systèmes distribués fondés sur des architectures dynamiques.

Nous adoptons le modèle de composant Fractal [22, 21] comme base pour la représentation des architectures des systèmes distribués. Le modèle de composant Fractal, qui a été présenté à la section 6.1.2, est intéressant ici pour sa généralité et sa flexibilité. Il dote les systèmes distribués que nous considérons de capacités d'adaptations dynamiques. Nous complémentons le modèle de composant Fractal par une nouvelle construction langage spécialisée pour la description des systèmes distribués fondés sur les architectures dynamiques, le *package*. Le package est une structure permettant d'organiser la description et le déploiement des architectures dynamique. Il est associé à un modèle d'installation locale qui décrit précisément son mode de fonctionnement sur une machine simple.

8.2.1 Modèle d'installation locale

Nous décrivons maintenant le modèle d'installation locale qui s'intéresse au déploiement de programmes localement sur une machine. Nous nous appuyons sur ce mécanisme pour mettre en œuvre des déploiements distribués complexes en assemblant et en coordonnant plusieurs déploiements locaux. Le modèle d'installation locale s'appuie sur une unité de déploiement, le *module*, et en décrit précisément le fonctionnement. Le *package* est une construction langage que nous construisons au-dessus du module et qui permet de décrire l'architecture d'un système distribué. Le package permet d'intégrer à la description des programmes des indications relatives à leur déploiement dans le cadre du modèle d'installation locale. Nous détaillons précisément dans les paragraphes qui suivent le modèle d'installation locale, les modules et les packages.

Tas d'une machine. Une machine est un environnement d'exécution de programmes. La machine est principalement constituée d'un *tas* qui abrite des éléments inertes qui sont du *code* et des *données*. Le tas d'une machine forme ainsi un graphe de code et de données. L'activité de la machine est représentée par des éléments en évolution qui sont les *processus* (ou *threads*), qui exécutent du code et travaillent sur des données présents dans le tas de la machine, faisant ainsi évoluer le tas de la machine.

Module. Le *module* est une unité de structuration du code et des données qui joue un double rôle. D'une part, le module permet d'injecter du code, des données et des processus sur une machine (c'est-à-dire dans son tas). Le module est donc une unité de déploiement. D'autre part, le module est capable d'exprimer ses dépendances (ce que le module importe) ainsi qu'une interface publique (ce que le module exporte). Cela permet de structurer le tas de la machine en un graphe de modules avec des liaisons bien identifiées. Cette capacité de structuration est essentielle pour contrôler le sous-graphe de code et de données qu'embarque chaque module.

Un module a la forme d'un composant Fractal, dont les interfaces et liaisons représentent les dépendances et interactions avec d'autres modules, et dont le contenu est le graphe de code et de données embarqué par le module. Un module peut également contenir d'autres modules, ce qui signifie qu'il embarque également les graphes de code et de données de ces modules. Cette faculté permet d'organiser et de contrôler les transports des modules sur les machines sur lesquelles ils doivent être apportés. Enfin, un module peut également indiquer par des attributs sa version, des contraintes sur l'environnement d'accueil, sur la machine ou les modules dont il dépend, etc.

Installation. L'installation d'un module repose sur un *chargeur* dont la tâche est d'identifier les modules nécessaires et appropriés au fonctionnement du système, de les localiser puis de les importer sur la machine locale, et enfin de les injecter dans le tas de la machine.

Les modules peuvent être organisés dans des bases de modules, ou *entrepôt* (*repository*). L'entrepôt est un espace de nommage et de stockage des modules : il répertorie,

stocke et diffuse les modules (par exemple au moyen d'un serveur Web HTTP distribuant les modules sous forme de fichiers). Par exemple, les modules systèmes sont organisés dans des entrepôts systèmes directement présents sur toutes les machines. Les modules sont identifiés par des références qui peuvent prendre diverses formes (nom de fichier, URL, etc).

L'identification par le chargeur des modules nécessaires et appropriés au fonctionnement du système repose sur le graphe des dépendances exprimées par les modules (imports) ainsi que sur leurs attributs indiquant leurs versions, contraintes, etc. La localisation des modules consiste à déterminer dans quels entrepôts les modules sont disponibles. Un chargeur dispose généralement d'une liste d'entrepôts à considérer, dont par exemple les entrepôts systèmes des machines. L'importation du module sur la machine locale s'effectue en transportant le module sous forme sérialisée. Enfin, l'injection du module dans le tas de la machine consiste à réintroduire le graphe de code et de données embarqué par le module dans le tas de la machine.

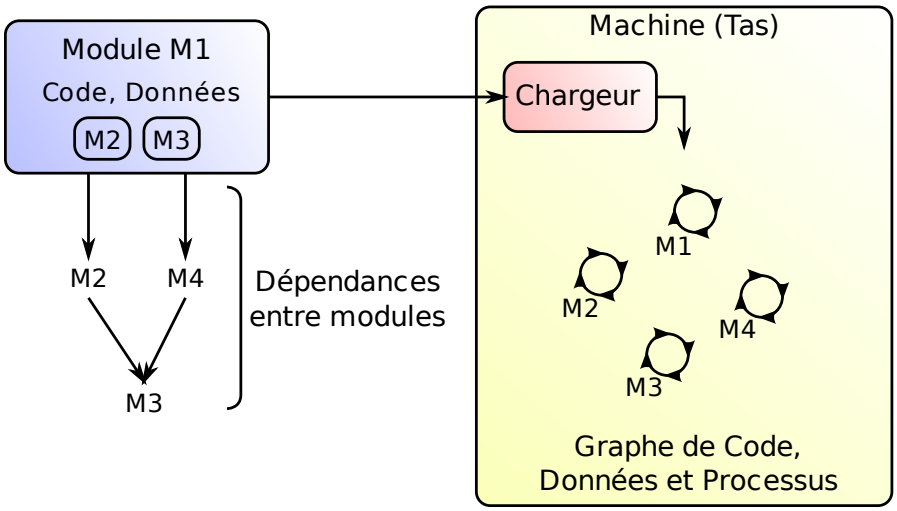


FIG. 8.1 – Installation de modules sur une machine.

La figure 8.1 présente un graphe de modules, dont le module M_1 exprimant des dépendances directes sur les modules M_2 et M_4, qui dépendent tous deux du module M_3. Les dépendances sont représentées par les interfaces modélisant la relation d'import de module. Le module M_1 embarque également les modules M_2 et M_3 (relation de composition). Lors de l'installation de M_1 sur une machine cible, le chargeur transporte le module M_1 pour le rendre disponible sur la machine locale, puis résout ses dépendances. Cela déclenche l'installation des modules M_2 et M_4, puis de M_3. Notons que M_2 et M_3 sont alors déjà présents sur la machine locale, car ils sont contenus dans le module M_1. Après installation, le code, les données et les processus des différents modules sont présents dans le tas de la machine locale.

Package. Un *package* est une construction langage permettant de décrire et de manipuler directement au sein d'un programme une forme particulière de module qui représente et embarque un composant. Le package est donc un module qui exporte un composant dont il embarque le code et les données. La construction langage associée au package est similaire à une déclaration de fonction ou de classe, et permet de décrire le code et les données contenus dans le package. En particulier, cette construction intègre la description d'une procédure spécifique réalisant le déploiement du composant représenté par le package. L'interface publique du module sous-jacent à un package rend accessible le composant produit par la procédure de déploiement du package.

En résumé, nous proposons une structure d'architectures dynamiques distribuées fondée sur le modèle de composant Fractal. Les architectures dynamiques sont décrites et manipulées au moyen de procédures de déploiement. Le package est l'unité de description et de déploiement d'un composant à la base des architectures dynamiques. Une procédure de déploiement décrit un processus qui effectue des opérations de construction et de manipulation d'architecture. L'installation d'un package initie le processus exécutant la procédure de déploiement correspondante et ayant pour résultat le déploiement du composant et donc de l'architecture dynamique représentée par le package. Nous détaillons dans la section suivante notre mise en œuvre du modèle de composant Fractal et du modèle d'installation locale, et particulièrement des packages dans le cadre de l'implantation du canevas FructOz spécialisé dans la construction des systèmes distribués fondés sur les architectures dynamiques.

8.3 Réalisation de FructOz

Cette section présente la mise en œuvre du canevas FructOz spécialisé dans la construction d'architectures dynamiques distribuées. Le canevas FructOz est construit sur la plate-forme Mozart/Oz présentée en section 6.1.3. FructOz intègre une implantation du modèle de composant Fractal ainsi qu'une implantation du modèle d'installation locale dont notamment des packages décrits précédemment.

Les abstractions représentant les composants sont présentes, introspectables et modifiables durant l'exécution des systèmes. Ce choix est guidé par le besoin de reconfiguration dynamique nécessaire pour construire des systèmes autonomes auto-optimisés. Plus précisément, une membrane de composant est un ensemble d'interfaces qui peut être dynamiquement modifié : autrement dit, nous pouvons dynamiquement ajouter ou retirer des interfaces au composant. Les liaisons architecturales (d'interface à interface) ainsi que les liaisons d'implantation (entre une interface et le substrat d'exécution natif) peuvent être créées et rompues dynamiquement à l'exécution. La composition est également manipulable à l'exécution, par ajout et retrait dynamique de sous-composant.

Nous choisissons d'implanter le modèle présenté précédemment sur la plate-forme d'accueil Mozart/Oz, et nous détaillons maintenant différents aspects de cette mise en œuvre, et notamment comment le langage Oz facilite l'implantation de nos abstractions.

8.3.1 Mise en œuvre du modèle de composant

Nous mettons en œuvre le modèle de composant Fractal présenté à la section 6.1.2 en implantant sous forme d'objets Oz les membranes, les interfaces et les liaisons. Nous assimilons et identifions un composant à sa membrane. Nous nous sommes attachés à autoriser un haut degré de dynamisme dans notre implantation du modèle Fractal. En particulier, dans notre implantation, nous pouvons dynamiquement ajouter ou supprimer des interfaces, des liaisons et des sous-composants à un composant. Enfin, nous n'implantons pas de contrôle ni de vérification active des opérations sur les composants. En particulier, la membrane du composant a un rôle exclusivement représentatif et ne présente aucune garantie d'isolation.

Liaisons. Une interface est mise en œuvre au moyen d'une référence vers un couple (port,stream) Oz. Le code d'implantation associé à l'interface traite les messages qui arrivent dans le stream. La référence permet de changer dynamiquement le code d'implantation associé à l'interface en redirigeant les émetteurs de messages vers un nouveau couple (port,stream). Une liaison repose sur un thread qui se comporte comme une pompe à messages, en consommant les messages arrivant sur le stream de l'interface source et en les adressant au port de l'interface destination. Ce choix est justifié par sa simplicité de mise en œuvre et par le faible coût des threads Oz.

Composition. La composition est mise en œuvre au moyen de deux contrôleurs interdépendants, l'un responsable de la gestion de l'ensemble des sous-composants, l'autre de la gestion des composants parent (aussi appelés super-composants).

Contrôleurs. Il n'y a pas de séparation structurelle entre la partie contrôle (les contrôleurs) et la partie fonctionnelle d'un composant. Les contrôleurs sont mis en œuvre au moyen d'interfaces serveurs conventionnelles. Nous fournissons certains des contrôleurs traditionnels du modèle de composant Fractal, dont un contrôleur d'attributs, un contrôleur de contenu ou encore un contrôleur de liaisons.

Cycle de vie. Le cycle de vie d'un composant est libre et notre implantation n'impose aucune contrainte. En particulier, une interface non liée peut être utilisée : les messages qui lui sont adressés seront stockés par l'interface (dans le stream qu'elle contient) et seront délivrés lors de l'établissement effectif d'une liaison sur l'interface. Cela implique toutefois de la part du programmeur qu'il s'assure qu'il n'y aura pas d'interblocage. En outre, cela autorise la conception de déploiements paresseux.

La table 8.1 rassemble les principales primitives relatives à la description et à la construction de composants dans notre implantation du modèle Fractal en Oz. Les primitives sont classées selon l'abstraction qu'elles manipulent, indiquée par la première lettre de leur nom : C pour composant, I pour interface et enfin B pour les liaisons

(*binding*). Les primitives {CNew}, {INew kind} et {BNew IFrom ITo} permettent de créer respectivement une membrane de composant initialement vide et ne comportant aucune interface, une interface de type cliente ou serveur (kind) et n'appartenant à aucun composant, et enfin une liaison (binding) entre une interface cliente (IFrom) et une interface serveur (ITo). Une interface est intégrée ou retirée d'une membrane au moyen des primitives {CAddInterface Comp Itf} et {CRemoveInterface Comp Itf}. Une liaison peut être rompue à l'aide de la primitive : {BBreak Binding}. Enfin, le lien entre les abstractions du modèle de composant et le substrat d'exécution est mis en œuvre d'une part au moyen de la primitive {IImplements Itf NativeCode} qui permet d'associer à une interface le code natif qui implante la fonction représentée par l'interface ; et d'autre part par une primitive symétrique {IResolve Itf} qui permet de récupérer une référence au code natif associé à une interface, dans l'objectif de l'exploiter fonctionnellement.

{CNew}	création d'une membrane de composant vide
{CAddInterface C I}	ajout de l'interface I à la membrane du composant C
{CRemoveInterface C I}	retrait de l'interface I de la membrane du composant C
{INew Kind}	création d'une interface libre (Kind : client ou serveur)
{IImplements I Proc}	liaison de l'interface I à la procédure d'implantation Proc
{IResolve I}	résolution de l'interface I en une référence (ou un proxy) à sa procédure d'implantation
{BNew IFrom ITo}	établissement d'une liaison de l'interface IFrom vers l'interface ITo
{BNewLazy IFrom ITo}	établissement d'une liaison paresseuse
{BBreak B}	rupture de la liaison B

TAB. 8.1 – Principales primitives de construction de composants Fractal.

Nous illustrons l'utilisation de ces primitives par les quelques fragments de code qui suivent.

Création d'un composant

```
%% Creation d'une membrane initialement vide
Comp = {CNew}

%% Ajout d'une interface serveur  la membrane du composant
Itf ={INew server [/* liste de tags */]}
{CAddInterface Comp Itf}
```

Liaison entre interfaces

```
%% Comment lier une interface client (service requis)  une interface serveur (service fourni)
IClient = {INew client [/* liste de tags */]}
IServer = {INew server [/* liste de tags */]}

%% Etablissement d'une liaison qui propage les messages adresses  l'interface IClient vers l'interface IServer
```

```
Binding = {BNew IClient IServer}

%% ... et plus tard, rupture de la liaison
{BBreak Binding}
```

Implantation fonctionnelle d'une interface

```
%% Comment definir l'implantation d'une interface serveur?
Itf = {INew server [/* liste de tags */]}

%% Implementation d'une interface par un objet Oz
{IImplements Itf
    {New class $
                meth message1(Param1 ...)
                    /* traitement des messages de type 'message1' */
                end
        end
    init}}
```

Usage fonctionnel d'une interface

```
%% Comment faire usage du service represente par une interface client?
Itf = {INew client [/* listes de tags */]}

%% Resolution de l'interface en un proxy
ItfProxy = {IResolve Itf}

%% Invocation du service represente par l'interface
{ItfProxy message1(Param1 ...)}
```

8.3.2 Mise en œuvre du modèle d'installation locale

Nous décrivons dans cette section comment nous implantons le modèle d'installation locale dans le cadre du canevas FructOz sur la plate-forme Mozart/Oz.

Module. Nous faisons correspondre les modules de notre modèle d'installation locale avec les modules Oz et leur construction langage : les foncteurs (**functor**). Les modules Oz sont en effet une excellente approximation des modules de notre modèle d'installation locale, car ils permettent d'exprimer leurs dépendances (section **import** du foncteur) et leur interface publique (section **export** du foncteur). Le code et les données embarqués par un module Oz sont décrits par la procédure de création du module (section **define** du foncteur).

Installation. Nous reposons sur les chargeurs de modules Oz : les *ModuleManagers*. Lors de l'installation d'un module Oz, le ModuleManager procède à la résolution et à l'installation des dépendances du module à installer. Le module à installer est ensuite

chargé dans le tas de la machine Oz puis initialisé en invoquant la procédure de création du module (section **define** du foncteur). En réalité, le ModuleManager Oz résoud et charge les dépendances au besoin et de manière paresseuse (à la manière du chargement des classes Java par le *ClassLoader*).

Package. Nous implantons le package comme une forme particulière de module Oz qui exporte la membrane du composant matérialisé par le package. Nous associons ainsi le package à une construction langage permettant la description d'un composant. Cette construction langage consiste en un modèle de foncteur Oz dont la section **export** désigne la membrane du composant embarqué et exporté par le package, et dont la section d'initialisation **define** correspond à la procédure de déploiement du composant :

```
functor ComponentPackage
export Membrane
define
   %% deployment procedure of the component
   ...
   Membrane = ...
end
```

L'installation d'un package dans le cadre du modèle d'installation locale repose l'installation du module sous-jacent au package, laquelle injecte le module et ses dépendances sur la machine puis invoque la procédure de déploiement du composant. En résultat, nous disposons d'une référence à la membrane du composant ainsi déployé et exporté par le package. L'instanciation d'un composant par installation d'un package repose sur la primitive Deploy :

```
%% local component deployment
ComponentInstance = {Deploy ComponentPackage}
```

Notons que le même package peut être installé de multiples fois, conduisant au déploiement d'autant d'instances du composant qu'il embarque.

Le fragment de code suivant montre la forme générale de l'utilisation de la construction langage de description d'un composant sous forme de package :

```
functor ComponentPackage
export Membrane
define
   %% create the component membrane, initially empty
   Comp = {CNew}

   %% create and register the component interfaces
   Itf = {INew server ...}
   {IImplements Itf ...}
   {CAddInterface Comp Itf}
   ...

   %% create other internal stuff (e.g. sub−components, threads, etc)
   SC1 = {Deploy ChildComponentPackage}
```

```
    {CAddSubComponent Comp SC1}
    ...

    %% now the component is deployed and ready to serve, make it visible (through its membrane)
    Membrane = Comp
end
```

8.3.3 Mise en œuvre des déploiements distribués

Le modèle d'installation locale décrit l'installation des modules et des packages localement sur une machine, c'est-à-dire le fonctionnement du déploiement localement sur une machine. Nous construisons les déploiements distribués en assemblant et en coordonnant plusieurs déploiements locaux qui sont distribués sur un ensemble de machines.

Architecture de l'environnement distribué. Nous considérons ici les déploiements distribués sur des grappes de machines. Nous modélisons une grappe de machines interconnectées par un réseau local sous forme d'une architecture à composants : un composant Cluster représente la grappe de machines, et contient un ensemble de machines représentées par des composants Host. Les composants Cluster sont des usines permettant d'intégrer de nouvelles machines à la grappe en créant de nouveaux composants Host. Les composants Host sont des usines à composants permettant de créer de nouveaux composants localement sur les machines qu'ils représentent. Les composants ainsi créés sont des sous-composants des composants Host. De cette manière, il est possible de découvrir la structure de l'environnement distribué en introspectant l'architecture modélisant la grappe de machines. Les composants Cluster peuvent ainsi servir de paramètre définissant la portée d'un déploiement distribué. La distribution exacte des composants sur l'environnement distribué est également observable en explorant le contenu des composants Host représentant les machines. La figure 8.2 illustre cette modélisation sur une grappe composée de deux machines hébergeant un système distribué formé d'un composant client lié à un composant serveur.

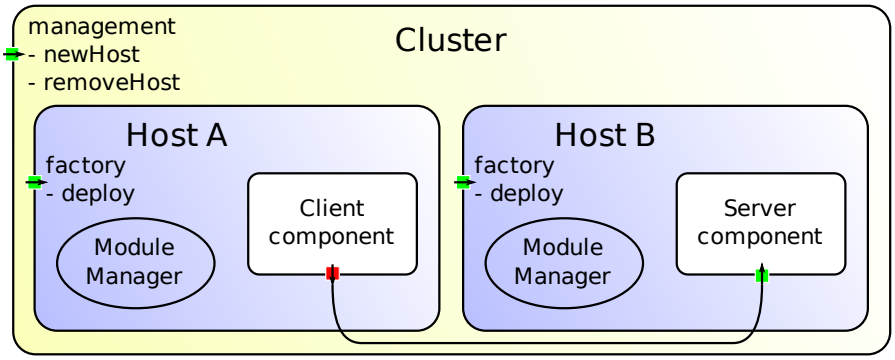


FIG. 8.2 – Modélisation et représentation de l'environnement distribué en grappe.

Contrôle de la distribution. L'exploitation de l'environnement distribué s'obtient en contrôlant le placement des composants instanciés sur les différentes machines de l'environnement distribué. Nous permettons de contrôler la machine sur laquelle déployer un composant au moyen d'une nouvelle primitive de déploiement RemoteDeploy qui étend la primitive Deploy. La primitive {RemoteDeploy Host Package} instancie le composant décrit par un package sur la machine représentée par le composant Host.

```
%% remote component deployment
Component = {RemoteDeploy Host Package}
```

Mise en œuvre de l'environnement distribué. Nous mettons en œuvre l'environnement distribué constitué d'une grappe de machines reliées par un réseau local grâce à la plate-forme distribuée Mozart/Oz présentée en section 6.1.3. La plate-forme Mozart/Oz supporte directement les environnements distribués sur des grappes de machines et fournit le contrôle de la distribution par le biais des proxy sur les gestionnaires de modules (*remote module manager*), qui permettent de commander l'installation de modules à distance sur les différentes machines de la grappe.

Le composant Cluster est une usine à composant Host. Pour créer de nouveaux composants Host représentant une machine à intégrer à la grappe, le composant Cluster démarre une machine virtuelle Oz sur la machine à intégrer et y instancie un ModuleManager duquel il conserve un proxy. Ce proxy permet d'instancier à distance un composant Host sur la nouvelle machine Oz distante. Le nouveau composant Host est physiquement localisé sur la machine qu'il représente. Le composant Host est une usine à composants locale à une machine. Il repose sur un ModuleManager localement présent sur la machine qu'il représente pour instancier de nouveaux composants sur cette machine.

Nous avons décrit dans cette section les détails d'implantation du canevas FructOz spécialisé dans la construction de systèmes distribués fondés sur des architectures dynamiques. Nous présentons dans la section qui suit plusieurs exemples d'application immédiate du canevas FructOz démontrant son intérêt et son expressivité.

8.4 Exemples d'utilisation de FructOz

Cette section regroupe différents exemples de code démontrant l'utilisation de notre construction linguistique pour la description des packages. Ces exemples sont représentatifs de l'expressivité du canevas FructOz pour la description d'architectures dynamiques. Ces extraits de code démontrent notamment les capacités de paramétrage, de contrôles de la distribution ainsi que de l'exécution (workflow) des architectures et des déploiements exprimés grâce au canevas FructOz. Ces capacités sont obtenues en combinant les descriptions de packages avec les structures naturelles du langage Oz.

8.4.1 Architectures paramétrées

Nous démontrons ici la capacité d'expression des architectures paramétrées, au moyen de l'exemple suivant : un composant composite contient deux ensembles de sous-composants de type C_1 et C_2, et de taille paramétrable N_1 et N_2. Les deux ensembles de sous-composants sont interconnectés selon un schéma d'interconnexion paramétré par une fonction BindingScheme qui décrit l'ensemble des paires de composants qui doivent être liés.

```
fun {ParameterizedComposite C1 N1 C2 N2 BindingScheme}
    functor $
    export Membrane
    define
        C = {CNew nil}
        %% Create the first set S1 with N1 instances of component C1.
        S1 = {SNew} % create a new empty set S1
        for I in 1..N1 do
            SubComp = {Deploy C1} % deploy a component C1 locally
        in
            {S1 add(SubComp)} % add the new component into the set S1
        end
        %% Create S2 similarly to S1
        ...
        %% Invoke the interconnection scheme function: generate a list of binding descriptors.
        LDescs = {BindingScheme S1 S2}
        %% Translate these descriptors into real bindings.
        LBindings = {List.map LDescs
            fun {$ IFrom#ITo}
                {BNew IFrom ITo}
            end}
        Membrane = C
    end
end
```

L'architecture paramétrée prend la forme d'un foncteur Oz paramétré, que nous exprimons au moyen d'une fermeture embarquant les différents paramètres de l'architecture.

Voici un exemple simple de schéma d'interconnexion réalisant l'interconnexion totale entre les deux ensembles de composants, c'est-à-dire calculant le produit cartésien des deux ensembles de sous-composants $S_1 \times S_2$.

```
fun {FullInterconnect S1 S2}
    L = {NewCell nil} % create a new empty descriptor list
in
    %% for all pairs (CFrom, CTo) in (S1 x S2)
    {S1 forAll(
        proc {$ CFrom}
            {S2 forAll(
                proc {$ CTo}
                    IFrom = {CGetInterface CFrom [/*tags list*/]} % locate the client interface
                    ITo = {CGetInterface CTo [/*tags list*/]} % locate the server interface
                    Desc = IFrom#ITo % make the descriptor
```

```
                in
                    L := Desc | @L % prepend the new descriptor to the list
                end)}
        end)}
    @L % return the list of descriptor
end
```

8.4.2 Synchronisations avancées

Nous démontrons ici la capacité à exprimer des contrôles de flots d'exécution complexes, dans l'optique de contrôler les déploiements distribués. La procédure suivante implante un schéma de contrôle réalisant une barrière de synchronisation sur l'exécution parallèle de N instances d'une procédure P.

```
%% Barrier synchronization pattern
proc {Barrier P N}
    Bar = {Tuple.make barrier N}
in
    for I in 1..N do
        thread
            {P} % invoke the procedure
            Bar.I = true % ready signal once the procedure is over
        end
    end
    {Record.forAll Bar Wait} % wait for all N ready signals
end
```

L'application de ce schéma de synchronisation au déploiement parallèle de N_2 composants C_2 prend alors la forme suivante :

```
%% Apply the barrier pattern to deploy N2 instances of C2 components
{Barrier
    proc {$}
        SubComp = {Deploy C2}
    in
        {S2 add(SubComp)}
    end
    N2}
```

Les N_2 composants sont alors déployés en parallèles, et l'exécution de la barrière se synchronise sur la terminaisons de l'ensemble des procédures de déploiements initiées.

8.4.3 Déploiements paresseux

Nous montrons ici comment mettre en œuvre des composants dont le déploiement est paresseux. Les déploiements paresseux sont intéressants, car ils permettent d'instancier uniquement les sous-systèmes effectivement utilisés dans un système composé. Cela permet, par exemple, d'économiser des ressources (mémoire, processeur, etc) en empêchant le déploiement des parties inutilisées du système. Nous considérons dans cet exemple les trois niveaux de déploiement suivants :

Niveau 0 : le composant n'est pas déployé, il est simplement représenté par une référence.

Niveau 1 : la membrane du composant est déployée, ce qui autorise l'introspection des interfaces du composant et de ses sous-composants immédiats ; l'implantation du composant n'est pas encore déployée.

Niveau 2 : le composant est entièrement déployé (membrane et implantation) ; cela n'impose rien sur les niveaux de déploiement de ses sous-composants.

Notons qu'il existe de multiples transitions entre les niveau 1 et 2, car l'implantation du composant peut éventuellement comporter d'autres niveaux de déploiement intermédiaires.

Ces déploiements paresseux reposent sur l'établissement de liaisons paresseuses par la primitive BNewLazy. Les liaisons paresseuses nécessitent l'existence de l'interface client mais tolèrent le déploiement paresseux de l'interface serveur participant à la liaison. Le déploiement de l'interface serveur sera alors déclenché lorsque la liaison sera sollicitée, soit fonctionnellement, soit pour des opérations de contrôle comme l'introspection.

```
%% Lazily deployed implementation
Itf = {INew server [tags ...]}
{IImplements Itf
    {ByNeed fun {$} % Implementation will be lazily instantiated
        {New class $ ... end}}}

%% Lazily deployed component
Client = {ByNeed fun {$} {Deploy LazyClient} end}

%% Lazily deployed binding between a Client component and a Server component
B = thread
        %% Wait until someone needs and thus triggers the deployment of the Client component
        %% Once this has happened, get the client interface
        IFrom = {CGetInterface {WaitQuietValue Client} [client]}
        %% Create a lazy reference to the server interface
        ITo = {ByNeed fun {$} {CGetInterface Server [service]} end}
    in
        %% Create a lazy binding between Client (now determined) and Server (might still be lazy)
        {BNewLazy IFrom ITo}
    end
```

8.4.4 Gestion d'erreurs

Nous démontrons maintenant comment intégrer la gestion d'erreurs aux déploiements. Nous considérons ici les erreurs prenant la forme d'exceptions. La gestion d'erreurs est intéressante, car elle constitue une base pour réaliser des déploiements atomiques (par exemple, par annulation pour par compensation). De manière plus générale, elle permet de concevoir des architectures qui s'adaptent à leur contexte d'exécution. À ce titre, nous instancions le schéma très général de contrôle d'erreur suivant :

```
%% Basic try/catch error handling pattern
proc {HandleError P E}
    try {P} % execute procedure P
    catch AnyException then
        {E} % if any error happens during P, then execute E
    end
end
```

Dans le scénario suivant, nous mettons en œuvre un mécanisme de compensation relatif à l'installation d'un composant de diagnostic (log). Par défaut, la procédure de déploiement tente de déployer un composant de diagnostic distant (RemoteLogger) ; en cas d'échec, nous nous rabattons sur un composant de diagnostic local. Notons que le schéma de contrôle d'erreur (HandleError) peut être utilisé en tant que paramètre au même titre que le schéma de synchronisation par barrière présenté précédemment à la section 8.4.2.

```
%% Example of deployment with error handling
{HandleError
    proc {$}
        % try to deploy a component RemoteLogger
        Log = {Deploy RemoteLogger}
    end
    proc {$}
        % if RemoteLogger cannot be instantiated, fall back on component FileLogger
        Log = {Deploy FileLogger}
    end}

%% Within the deployment procedure of RemoteLogger
if (/*remote resource is not available*/) then
    raise instantiationFailure(cause: unavailableResource) end
end
```

8.4.5 Déploiements distribués

Nous présentons ici des exemples démontrant les facultés d'expression de déploiements distribués complexes par différentes stratégies de déploiements distribués. Un déploiement distribué est un processus de déploiement d'un système fondé sur une architecture distribuée sur un ensemble de machines. Un tel processus implique divers déploiements localement sur chaque machine. Il est intéressant de paralléliser ou de coordonner ces déploiements locaux afin, par exemple, de respecter des cycles de vie arbitraires ou bien pour des raisons de performance.

Nous considérons le déploiement d'un composant composite abritant un ensemble de sous-composants identiques déployés sur une grappe de machines. La complexité du déploiement est reliée à la taille du composant composite, que nous mesurons par le nombre de ses sous-composants. Le composant composite est paramétré par le composant Cluster représentant la grappe de machines sur laquelle déployer les sous-composants, par le nombre N de sous-composants dans le composite et enfin par une procédure de déploiement DeployProc à utiliser pour déployer les N sous-composants sur la grappe de machines. Nous décrivons ce composite de la manière suivante :

```
fun {CompositePackage Cluster N DeployProc}
    functor $
    export Membrane
    define
        Comp = {CNew}
        {DeployProc SubcomponentPackage Cluster N Comp}
        Membrane = Comp
    end
end
```

La procédure de déploiement DeployProc est responsable du déploiement de N composants identiques sur la grappe de machines Cluster donnée en paramètre. Les sous-composants à déployer sont décrits dans un package SubcomponentPackage non spécifié ici.

Pour déterminer la machine sur laquelle déployer l'un des sous-composants, nous utilisons une stratégie de tourniquet (round-robin) sur l'ensemble des machines constituant la grappe. Cette stratégie prend la forme d'une fonction NextNode = {MakeRoundRobin Cluster} dont chaque invocation Host = {NextNode} renvoie la prochaine machine sur laquelle doit être effectué le déploiement.

Nous détaillons maintenant différentes procédures de déploiement correspondant au paramètre DeployProc.

Déploiement séquentiel

Une première stratégie consiste à déployer séquentiellement depuis un contrôleur (i.e. un processus de déploiement) centralisé les N composants sur les machines de la grappe. Le déploiement séquentiel prend la forme d'une simple boucle for.

```
proc {DeploySeq CompPackage Cluster N Parent}
    NextRoundRobin = {MakeRoundRobin Cluster}
    for I = 1..N do
        Host = {NextRoundRobin}
        NewComp = {RemoteDeploy Host CompPackage}
        {CAddSubComponent Parent NewComp}
    end
end
```

Déploiement parallèle asynchrone

La stratégie séquentielle précédente peut être trivialement rendue parallèle, mais non synchronisée en encadrant le contenu de la boucle for réalisant le déploiement d'un sous-composant par un bloc **thread** ... **end**.

```
proc {DeployParallelAsync CompPackage Cluster N Parent}
    NextRoundRobin = {MakeRoundRobin Cluster}
    for I = 1..N do
        thread
            Host = {NextRoundRobin}
            NewComp = {RemoteDeploy Host CompPackage}
```

```
        in
            {CAddSubComponent Parent NewComp}
        end
    end
end
```

Déploiement parallèle synchrone

En nous appuyant sur le schéma de synchronisation par barrière déjà présenté (voir en annexe, page 199), nous rendons le déploiement parallèle précédent synchrone.

```
proc {DeployParallel CompPackage Cluster N Parent}
    NextRoundRobin = {MakeRoundRobin Cluster}
    %% Deployment script
    proc {DeployProc}
        Host = {NextRoundRobin}
        NewComp = {RemoteDeploy Host CompPackage}
    in
        {CAddSubComponent Parent NewComp}
    end
in
    %% Execute and synchronize on the scripts' execution
    {Barrier {MakeCopyList DeployProc N}}
end
```

Déploiement décentralisé hiérarchique

Toutes les stratégies de déploiement présentées jusqu'à maintenant sont exécutées et contrôlées par un processus centralisé depuis une machine unique. Nous illustrons ici comment construire un processus distribué de déploiement selon une stratégie de propagation arborescente.

Le déploiement est initié sur une machine assimilée à la racine de l'arbre de propagation et est paramétré par la profondeur et le degré (l'arité) de l'arbre de propagation. Sur un nœud donné de l'arbre, le processus déploie localement une instance du sous-composant, puis initie un nouveau processus de déploiement arborescent sur chacun des nœuds correspondant aux branches du nœud courant, sur lesquels le processus est synchronisé. Enfin, lorsque le nœud correspond à une feuille de l'arbre de propagation, le processus consiste simplement à déployer un composant sur la machine locale.

```
NextRoundRobin = {MakeRoundRobin Cluster}
proc {DeployTree CompPackage Arity Depth Parent}
    functor DistributedDeployProc
    export Membrane
    define
        if (Depth > 0) then
            {DeployTree CompPackage Arity (Depth − 1) Parent}
        end
        Membrane = {Deploy CompPackage} % deploy component locally
    end
```

```
    proc {DeployProc}
        Host = {NextRoundRobin}
        NewComp = {RemoteDeploy Host DistributedDeployProc}
    in
        {CAddSubComponent Parent NewComp}
    end
in
    {Barrier {MakeCopyList DeployProc Arity}}
end
```

8.5 Interface de programmation du canevas FructOz

La table 8.2 rassemble l'ensemble des constructions et primitives constituant le canevas FructOz spécialisé dans la description de systèmes distribués fondés sur des architectures dynamiques.

Primitive	Description
functor Pkg **export** Membrane **define** ... **end**	description du package Pkg
{Deploy Pkg}	déploiement local du composant décrit par le package Pkg
{RemoteDeploy Host Pkg}	déploiement distant sur la machine Host du composant décrit par le package Pkg
{CNew}	création d'une membrane de composant vide
{CAddInterface C I}	ajout de l'interface I à la membrane du composant C
{CRemoveInterface C I}	retrait de l'interface I de la membrane du composant C
{CGetInterfaces C}	ensemble des interfaces du composant C
{INew Kind}	création d'une interface libre (Kind : client ou serveur)
{IImplements I Proc}	liaison de l'interface I à la procédure d'implantation Proc
{IResolve I}	résolution de l'interface I en une référence (ou un proxy) à sa procédure d'implantation
{IGetComponent I}	composant propriétaire de l'interface I
{IGetBindingsFrom I}	ensemble des liaisons dont I est l'interface client
{IGetBindingsTo I}	ensemble des liaisons dont I est l'interface serveur
{BNew IFrom ITo}	établissement d'une liaison de l'interface IFrom vers l'interface ITo
{BNewLazy IFrom ITo}	établissement d'une liaison paresseuse
{BBreak B}	rupture de la liaison B
{BGetClientInterface B}	interface client dans la liaison B
{BGetServerInterface B}	interface serveur dans la liaison B
{CAddSubComponent C SC}	ajout du sous-composant SC au composant C
{CRemoveSubComponent C SC}	retrait du sous-composant SC du composant C
{CGetSubComponents C}	ensemble des sous-composants du composant C
{CGetSuperComponents C}	ensemble des composants parents du composant C
{CSetAttribute C Attr Val}	définition de l'attribut Attr du composant C à la valeur Val
{CGetAttribute C Attr}	valeur de l'attribut Attr du composant C
{CHasAttributes C Attr}	test de l'existence de l'attribut Attr du composant C
{CListAttributes C}	ensemble des attributs du composant C

TAB. 8.2 – Principales primitives du canevas FructOz spécialisé dans la construction d'architectures dynamiques distribuées.

8.6 Synthèse

Nous avons présenté au cours de ce chapitre la conception, la réalisation et quelques exemples d'utilisation du canevas FructOz spécialisé dans la description et la construction des systèmes distribués fondés sur des architectures dynamiques. FructOz représente l'architecture des systèmes distribués sous forme de composants Fractal, lesquels

sont implantés et intégrés à la plate-forme Mozart/Oz. FructOz intègre une nouvelle construction linguistique, le *package*, qui constitue l'unité de description et de déploiement des composants. Les descriptions de composants FructOz bénéficient ainsi directement de l'expressivité et des facultés offertes par le langage Oz. FructOz implante de cette manière un langage de description d'architecture (ADL) d'ordre supérieur. Cela permet de construire des architectures dynamiques et des déploiements paramétrés qui autorisent un contrôle étendu de la distribution, du parallélisme et de l'exécution (workflow) des systèmes distribués construits.

Les packages sont des descripteurs de composants et des unités de déploiement qui permettent de décrire simultanément l'architecture d'un système distribué, son processus de déploiement distribué ainsi que ses manipulations (reconfigurations dynamiques) sous forme de procédures de déploiements. Le gain en expressivité provient de l'intégration forte entre la description des composants et la description de code fonctionnel au sein de la plate-forme Mozart/Oz, qui permet de mélanger des descriptions d'éléments d'architectures (interfaces, sous-composants, liaisons) avec des éléments de déploiement, concernant notamment la distribution et la coordination (workflow) et également des éléments de reconfiguration dynamique.

Le canevas FructOz permet la description et la construction de systèmes distribués à architectures dynamiques. Les descriptions et les procédures de déploiement de ces systèmes offrent dans FructOz un très haut niveau de paramétrage, comme par exemple le paramétrage du nombre et du type des composants, de leur schéma d'interconnexion, de leur mode de déploiement, de schémas de synchronisation, etc. FructOz offre également la possibilité de spécifier et de contrôler précisément le parallélisme, la distribution et les synchronisations intervenant lors du déploiement et lors des actions sur les systèmes décrits. FructOz permet en outre d'intégrer aux déploiements des mécanismes de gestion d'erreur, comme par exemple la compensation. Tous ces éléments et mécanismes font partie d'une même description et sont donc décrits dans un contexte uniforme. En particulier, FructOz repose sur la maîtrise d'un unique langage. Cette caractéristique élimine la nécessité de devoir concilier plusieurs modes d'expressions et de faire le lien entre de multiples descripteurs. Une évaluation du canevas FructOz est donnée au chapitre 11.

Ces travaux ouvrent la voie à de nombreuses extensions et améliorations. Pour finaliser la mise en œuvre du canevas FructOz, il faudrait achever l'intégration des abstractions des composants et des packages dans le langage d'accueil, notamment avec un support syntaxique spécialisé. Le canevas FructOz semble par ailleurs constituer une base solide pour envisager l'étude des reconfigurations transactionnelles dans le cadre de la construction des systèmes distribués fondés sur les architectures dynamiques. Les mécanismes transactionnels déjà existants dans le langage Oz reposent sur les espaces de calculs qui permettent de réaliser des calculs spéculatifs, et qui pourraient notamment être étendus dans cet objectif. Il semble enfin intéressant d'étudier la gestion des versions des packages et de leurs contraintes sur l'environnement.

La construction de systèmes distribués reposant sur des architectures dynamiques introduit une nouvelle source de complexité. Cette complexité provient justement du dynamisme de l'architecture des systèmes qui rend éphémères, et donc difficilement exploitables, les techniques classiques d'observation de l'architecture des systèmes distribués administrés. Le chapitre suivant s'intéresse justement à l'observation des systèmes fondés sur des architectures dynamiques.

Chapitre 9

LactOz : une bibliothèque pour l'observation et l'instrumentation des systèmes administrés

Nous présentons dans ce chapitre notre proposition pour l'observation des systèmes distribués administrés fondés sur des architectures dynamiques. Nous rappelons dans un premier temps la problématique relative à l'observation des systèmes distribués à architectures dynamiques. Nous détaillons ensuite les principes de conception de notre proposition pour l'observation de ces systèmes. Nous présentons l'implantation de la bibliothèque LactOz spécialisée dans l'observation et la navigation dans les architectures dynamiques. Enfin, nous donnons quelques exemples d'application immédiate de la bibliothèque LactOz, ainsi qu'une liste de référence des principales primitives de la bibliothèque.

9.1 Objectifs

Problématique. Dans le cadre de l'administration fondée sur l'architecture, les systèmes distribués administrés sont intimement associés à leur architecture qui constitue l'interface des opérations d'administration. L'architecture est une structure de données distribuée et dynamique, qui évolue au gré des reconfigurations dynamiques des systèmes administrés. L'état global de l'architecture du système administré est inaccessible pour deux raisons : (1) la nature distribuée de l'architecture éclate l'état sur l'ensemble des machines qui composent le système ; et (2) le dynamisme de l'architecture rend éphémère toute observation de l'état du système distribué en raison des évolutions du système.

Exemple. Nous souhaitons pouvoir réaliser des observations du système distribué administré qui sont d'une part indépendantes de la distribution de l'état du système, et qui d'autre part conservent leur cohérence malgré le dynamisme et les évolutions du

système. Par exemple, nous voudrions pouvoir exprimer dans le cadre d'un système dupliqué en grappe des propriétés de la forme : "chaque machine hébergeant une instance de service dupliqué doit accueillir une sonde de surveillance qui est raccordée à un coordinateur central". Cette propriété est fondée sur l'observation de l'*ensemble dynamique* des machines qui hébergent une instance d'un service dupliqué. Cet ensemble de machines est dynamique précisément parce les machines qui le composent ne sont pas connues dans l'absolu. En effet, cet ensemble de machines est défini par intension et évolue lorsque des opérations d'approvisionnement dynamique interviennent sur le système dupliqué en grappe.

Proposition. Nous proposons une technique pour consolider l'état global du système distribué administré permettant de s'abstraire de sa distribution et de capturer son dynamisme. Notre proposition repose sur un modèle de calcul dynamique distribué. D'une part, ce modèle unifie les tas d'un ensemble de machines pour constituer un tas distribué global et permet ainsi de raisonner indépendamment de la distribution des données. D'autre part, ce modèle introduit une représentation des valeurs dynamiques et autorise leur manipulation, ce qui permet ainsi d'exprimer des calculs sur des valeurs dynamiques dont les valeurs effectives évoluent au cours du temps. Nous appliquons ce modèle de calcul dynamique distribué aux composants mis en œuvre dans le cadre du canevas FructOz, et nous construisons par la suite la bibliothèque de primitives de navigation dynamique LactOz spécialisée dans l'observation des architectures dynamiques. La bibliothèque LactOz élève le niveau d'expression et de réflexion lors de la conception d'architectures dynamiques, en permettant précisément de désigner des sous-ensembles dynamiques de l'architecture, de raisonner sur ces ensembles dynamiques et d'exprimer des calculs sur ces ensembles dynamiques. En reprenant l'exemple d'un système dupliqué en grappe cité précédemment, il devient alors possible grâce au modèle de calcul dynamique de construire une valeur représentant l'ensemble dynamique des machines qui hébergent une instance d'un service dupliqué.

9.2 Principes de conception

Nous présentons dans cette section les choix de conception qui ont guidé la construction de la bibliothèque LactOz spécialisée dans l'observation et la navigation dans les architectures dynamiques. La bibliothèque LactOz repose sur (1) un modèle de calcul dynamique distribué et (2) l'application du modèle de calcul distribué et dynamique aux architectures des systèmes distribués dans le cadre du canevas FructOz.

9.2.1 Modèle de calcul dynamique distribué

Nous nous plaçons dans le cadre d'un environnement de calcul distribué donnant l'illusion d'un tas global unifié dans lequel les valeurs peuvent être consultées et manipulées indépendamment de leur distribution physique. Nous présentons maintenant

le modèle de calcul dynamique qui permet de définir et de désigner des valeurs dynamiques, ainsi que de raisonner sur ces valeurs dynamiques.

Le modèle de calcul dynamique repose sur des variables dynamiques définies par intension en fonction d'autres variables dynamiques, et dont les valeurs effectives sont automatiquement mises à jour. Nous distinguons différentes natures de variables. Les variables *statiques* sont des variables non mutables, définies une fois pour toute la durée de leur existence (à l'instar des variables Oz ou Caml, ou encore des variables const C et C++). Les variables *dynamiques* sont mutables et peuvent prendre des valeurs différentes au cours de leur existence. On distingue deux types de variables dynamiques. D'une part, les variables dynamiques *explicites* qui peuvent être directement modifiées par le programmeur ; ce sont l'équivalent des variables C traditionnelles, des cellules Oz, ou encore des références Caml. Et d'autre part, les variables dynamiques *implicites* qui ne peuvent être directement modifiées par le programmeur. Ces variables sont définies de manière intensionnelle en tant que fonctions d'autres variables. Ainsi, la valeur effective d'une variable dynamique implicite dépend des valeurs effectives des variables dynamiques dont elle est fonction, et doit donc être systématiquement réévaluée en cas de mise à jour de l'une des variables dont elle dépend.

Le fragment de code suivant illustre la notion de variable dynamique. Dans cet exemple, alors que la variable Y traditionnelle conserve la dernière valeur qui lui est assignée, ici $X + 5 = 6$, quand X est ensuite modifié et prend la valeur 2, la valeur effective de son équivalent dynamique s'adapte à la nouvelle valeur de X, et conduit à $Y = 7$.

```
% Variables traditionnelles (type C)
X = 1       % X=1
Y = X+5     % X=1, Y=6
X = 2       % X=2, Y=6
```

```
% Variables dynamiques
X = 1       % X=1 (X variable explicite)
Y = X+5     % X=1, Y=6 (Y variable implicite, fonction de X)
X = 2       % X=2, Y=7 (Y mis a jour, respecte Y=X+5)
```

Lorsqu'une variable dynamique est mise à jour, toutes les variables qui en dépendent doivent également être mises à jour. L'origine d'une mise à jour est nécessairement liée, directement ou indirectement, à une modification d'une variable dynamique explicite. La propagation du changement de l'état d'une variable dynamique peut alors intervenir soit de manière synchrone, soit de manière asynchrone. La figure 9.1 résume les différents types de variables impliquées dans notre modèle de calcul dynamique.

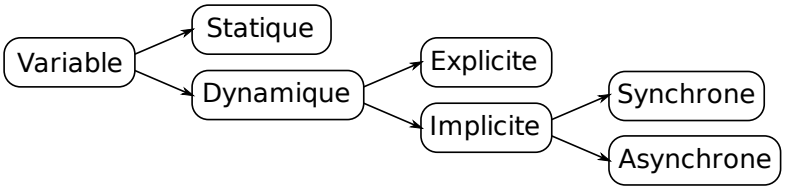


FIG. 9.1 – Hiérarchie des types de variables dynamiques.

Système à évènements

Les variables dynamiques sont supportées par un système à évènements distribué, lequel s'appuie sur le patron de conception de l'observateur [51]. Les mises à jour de variables dynamiques peuvent être implantées de deux manières différentes. Nous distinguons d'une part les variables dynamiques *absolues*, dont les mises à jour consistent à propager simplement aux observateurs la nouvelle valeur effective de la variable, comme illustré par l'algorithme 2. Les types de données primitifs comme les booléens ou les variables numériques, etc, sont adaptés à ce mode de propagation absolue. Et d'autre part, les variables dynamiques *relatives*, dont les mises à jour consistent à propager les modifications de la variable relativement à sa précédente valeur effective. Les variables composées comme les collections, les ensembles, etc, font partie de cette catégorie et gagnent généralement à propager leurs mises à jour sous la forme d'ajouts et de retraits d'éléments relativement à leur état précédent, selon le schéma illustré par l'algorithme 3.

Algorithme 2 Observateur générique absolu d'une variable dynamique.

```
on Update(new_value) do
    /* Notification de la nouvelle valeur effective (absolue) */
end
```

Algorithme 3 Observateur relatif d'une collection dynamique.

```
on AddElement(element) do
    /* Notification de l'ajout d'un élément dans l'ensemble (mise à jour relative) */
end
```

```
on RemoveElement(element) do
    /* Notification du retrait d'un élément dans l'ensemble (mise à jour relative) */
end
```

9.2.2 Application du calcul dynamique à l'observation des architectures dynamiques

Nous intégrons ce modèle de calcul dynamique et le modèle de composant Fractal issu du canevas FructOz, faisant ainsi des éléments de l'architecture des valeurs dynamiques manipulables dans le cadre du modèle de calcul dynamique. Nous fournissons par la suite un ensemble extensible de primitives de navigation dans les architectures dynamiques facilitant la construction d'expressions pour l'observation des architectures dynamiques.

L'intégration du modèle de composant Fractal au modèle de calcul dynamique repose fortement sur les structures de collections dynamiques. En effet, les relations entre les différents éléments du modèle de composant sont essentiellement de nature ensembliste. Par exemple, la membrane d'un composant maintient un ensemble d'interfaces, chaque interface est associée à deux ensembles de liaisons, un composant contient

un ensemble de sous-composants, etc. Nous adaptons en conséquence notre implantation du modèle de composant Fractal dans le cadre du canevas FructOz afin d'intégrer les éléments d'architecture au modèle de calcul dynamique. Cette adaptation consiste principalement à remplacer les structures de données traditionnelles par des structures équivalentes dynamiques. Les structures des entités qui représentent les abstractions du modèle de composant sont ainsi modifiées pour remplacer les ensembles traditionnels par des ensembles dynamiques.

Les trois entités membranes, interfaces et liaisons (respectivement notées $\mathcal{C}$, $\mathcal{I}$ et $\mathcal{B}$) font désormais partie intégrante du modèle de calcul. De façon analogue, les primitives de navigations qui permettent de parcourir et interroger l'architecture sont étendues et intégrées au modèle de calcul. Par exemple, une primitive CGetInterfaces Component : $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{I}\}$, permet de récupérer l'ensemble dynamique des interfaces que possède un composant.

Par composition des primitives de navigation et d'interrogation, des opérations ensemblistes et à l'aide de prédicats, LactOz permet ainsi de construire des expressions arbitraires de navigation et d'interrogation dynamique dans les architectures.

9.3 Réalisation de LactOz

LactOz implante le modèle de calcul dynamique au moyen d'un système évènementiel distribué, et s'intègre à notre modèle de composant sous la forme d'une extension du canevas FructOz qui, d'une part, le dote des structures minimales permettant de rendre dynamiques les abstractions du modèle de composant et, d'autre part, fournit une bibliothèque de primitives dynamiques de navigation et d'interrogation des architectures à base de composants.

9.3.1 Mise en œuvre du modèle de calcul dynamique distribué

Nous choisissons la plate-forme distribuée Mozart/Oz comme environnement d'exécution pour mettre en œuvre les calculs distribués. La plate-forme Mozart/Oz offre précisément l'illusion d'un tas unique indépendamment de la distribution des valeurs sur les différentes machines formant l'environnement distribué. Nous implantons un système à évènements dans le cadre défini par la plate-forme Mozart/Oz, et permettant de mettre en œuvre les calculs dynamiques. Les variables dynamiques LactOz sont représentées par des objets Oz qui propagent leurs mises à jour en s'échangeant des messages selon le schéma de l'observateur présenté précédemment.

Concrètement, une variable dynamique LactOz est assimilée à une entité observable qui propage ses mises à jour. L'abstraction d'une "entité observable" est implantée au moyen d'un gestionnaire d'observateurs (*ListenerManager*, ou *LM*) qui est chargé de diffuser les messages de mise à jour à un ensemble d'observateurs enregistrés auprès du ListenerManager. Une variable dynamique implicite est spécifiquement conçue pour dépendre d'autres variables. Pour chacune des variables dont elle dépend, la variable

dynamique met en œuvre un observateur au moyen d'une procédure de rappel (*Listener*) qu'elle enregistre auprès du gestionnaire d'observateurs de la variable dont elle souhaite recevoir les mises à jour. Chaque observateur indique s'il souhaite être averti de manière synchrone ou asynchrone des mises à jour de la variable qu'il observe. Les variables dynamiques explicites sont directement manipulées par le programmeur et n'ont aucune dépendance vis-à-vis d'autres variables. Les variables dynamiques explicites ne comportent donc pas de procédure de rappel.

Notons, comme illustré sur la figure 9.2, qu'une variable dynamique implicite est construite par l'assemblage d'une variable explicite dont elle contrôle précisément les mises à jour, empêchant spécifiquement toute mise à jour explicite de la part du programmeur, et d'un ensemble de procédures de rappel implantant des observateurs. Une variable statique peut être vue comme un cas particulier de variable dynamique qui n'évolue plus. Les variables statiques permettent d'optimiser l'évaluation des expressions dynamiques en économisant notamment des observateurs inutiles.

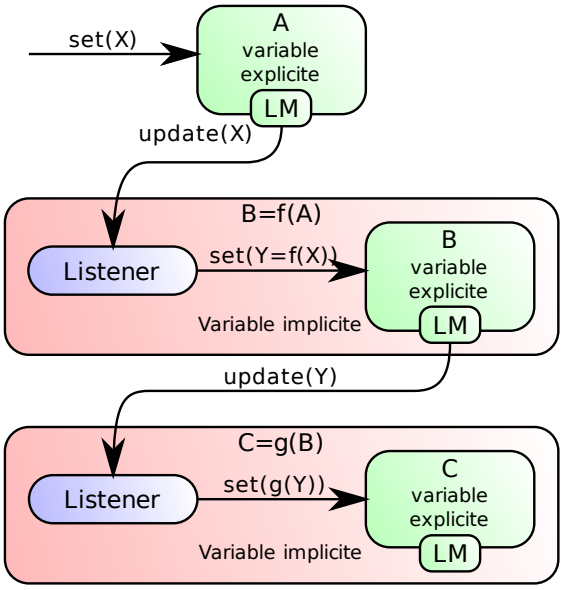


FIG. 9.2 – Illustration de l'implantation des variables dynamiques. Ici $C = g(B), B = f(A)$.

La figure 9.2 présente un exemple de variables dynamiques absolues : $C = g(B)$ et $B = f(A)$, où A est une variable explicite, et B et C sont deux variables implicites. Chaque variable implicite repose sur une variable explicite exclusivement contrôlée par un observateur. Une variable explicite n'expose donc qu'une interface permettant de consulter sa valeur effective et n'autorise pas sa modification explicite.

La figure 9.3 détaille rapidement l'implantation du schéma générique de conception de l'observateur/observé sur lequel nous nous appuyons pour mettre en œuvre le système évènementiel distribué. Les observateurs enregistrent leurs procédures de rappel au moyen de la méthode add(Listener Sync) qui indique l'objet Listener à enregistrer

```
% Gestionnaire d'observateurs
class ListenerManager
   meth add(Listener Sync)
      ... % Enregistrement d'un nouveau listener (synchrone ou asynchrone, selon Sync)
   end
   meth remove(Listener)
      ... % Suppression d'un listener
   end
   meth notify(Message)
      ... % Diffusion d'un message de mise  jour  tous les listeners enregistrs
   end
end
```

```
% Observateur de variable dynamique absolue
class AbsListener from Listener
   meth update(NewValue)
      ...
   end
end
```

```
% Observateur d'un ensemble dynamique (relatif)
class SetListener from Listener
   meth add(Elements)
      ...
   end
   meth remove(Elements)
      ...
   end
end
```

FIG. 9.3 – Interfaces génériques du patron de l'observateur.

ainsi que le mode de mise à jour souhaité (synchrone ou asynchrone). Les diffusions de messages de mises à jour sont commandées par la méthode notify(Message) qui invoque les procédures de rappel de chacun des observateurs enregistrés, soit de manière synchrone par invocation directe, soit de manière asynchrone au moyen d'un nouveau thread.

9.4 Exemples d'utilisation de LactOz

Nous présentons dans cette section quelques exemples d'application immédiate de notre mise en œuvre du modèle de calcul dynamique distribué dans le cadre de la construction de la bibliothèque LactOz et de son exploitation pour la construction de systèmes distribués fondés sur des architectures dynamiques. Nous détaillons d'abord la construction d'un prédicat de navigation et d'observation des architectures dynamiques. Nous présentons ensuite l'utilisation de la bibliothèque LactOz dans le cadre de la conception d'une architecture dynamique.

9.4.1 Construction de prédicats d'observation

Nous montrons ici comment exploiter et étendre la bibliothèque LactOz, en construisant des expressions spécialisées pour observer l'architecture du système administré. Ces expressions d'observation de l'architecture du système s'appuient sur des prédi-

cats dont la construction repose sur l'exploitation d'expressions Oz traditionnelles qui se composent de primitives d'introspection FructOz et de calcul dynamique LactOz. Nous faisons ici une démonstration de l'utilisation de ces primitives afin de décrire un nouveau prédicat : CGetExternalComponentsBoundFrom : $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{C}\}$, qui extrait l'ensemble des composants directement connectés à un composant donné et qui lui fournissent un service. Le processus d'exploration de l'architecture pour la réalisation de ce prédicat est illustré sur la figure 9.4 :

1. Le processus démarre initialement avec une référence sur le composant C ;
2. nous déduisons l'ensemble des interfaces clients ;
3. puis nous calculons l'ensemble des liaisons issues de ces interfaces ;
4. nous calculons l'ensemble des interfaces serveurs des composants liés à C ;
5. enfin, nous déduisons l'ensemble des références aux composants liés à C.

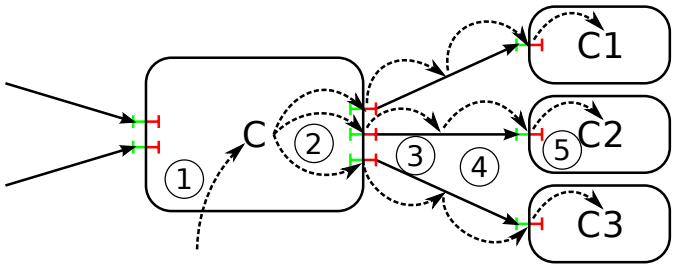


FIG. 9.4 – Processus d'exploration de l'architecture pour la construction du prédicat CGetExternalComponentsBoundFrom.

En nous appuyant sur les primitives de calcul dynamique et sur les primitives d'introspection primordiales, la mise en œuvre de ce nouveau prédicat prend la forme suivante :

```
%% 2 auxilliary introspection functions
fun {CGetClientInterfaces C} % get the dynamic set of client interfaces of a component
    %% The kind of an interface cannot change, so we optimize
    %% using a dynamic set filtering with a static filter function
    {SStaticFilter AllItfs (fun {$ I} I.kind == client end)}
end

fun {BGetServerComponent B} % get the component owning the interface the binding points to
    {IGetComponent {BGetServerInterface B}}
end

%% CGetExternalComponentsBoundFrom
%% Parameter: component C
%% Result: dynamic set of components
fun {CGetExternalComponentsBoundFrom C}
    ClientItfs = {CGetClientInterfaces C}
    %% SMap maps the Set of (client) Interfaces into a Set of Set of Bindings,
    %% that we flatten thanks to SUnion into a Set of Bindings
```

```
    ClientBindings = {SUnion {SMap ClientItfs (fun {$ I} {IGetBindingsFrom I} end)}}
    %% and finally, map the Set of Bindings into a Set of Components
    {SMap ClientBindings (fun {$ B} {BGetServerComponent B} end)}
end
```

Afin d'alléger les notations, nous reposons sur deux prédicats auxiliaires : (1) un prédicat CGetClientInterfaces qui calcule l'ensemble des interfaces clients d'un composant donné, et (2) un prédicat BGetServerComponent qui calcule le composant propriétaire de l'interface serveur participant à une liaison.

L'exploitation de ce nouveau prédicat s'opère sur le même modèle que celui utilisé pour sa construction. Supposons par exemple que nous souhaitons extraire l'ensemble des composants fournissant un service au composant C et qui ont un sous-composant associé à un marqueur "interesting" :

```
%% filter the dynamic set of components generated with the previous function
SInterestingComponents =
    {SFilter {CGetExternalComponentsBoundFrom C}
        fun {$ C}
            %% get the sub−components of C
            Children = {CGetSubComponents C}
        in
            %% only keep components for which the sub−component set contains at least
            %% one child component tagged with 'interesting', i.e. for which the sub−component
            %% set filtered with respect to the 'interesting' tag is not empty
            {BNot {SIsEmpty {SFilter Children {TaggedWith 'interesting'}}}}
        end}
```

Nous reposons sur l'opérateur de filtrage SFilter associé au prédicat anonyme dynamique, de prototype : $\mathcal{C} \rightarrow \mathbb{B}$, qui à chaque composant associe un booléen dynamique indiquant s'il contient un sous-composant associé au marqueur "interesting".

Notons ici que, dans un souci de clarté, nous avons omis quelques détails permettant de spécifier pour chacune de ces expressions si leur évaluation doit s'effectuer de façon synchrone ou asynchrone. En pratique, cela consiste à préfixer dans notre bibliothèque de fonctions les noms de primitives par les préfixes Sync ou Async.

9.4.2 Construction d'architectures dynamiques

Composant redimensionnable

Nous montrons ici comment mettre en œuvre une architecture dynamique qui consiste en un composant redimensionnable dont la taille est dynamiquement ajustable. Il s'agit ici de construire une architecture paramétrée par une variable dynamique représentant la taille du composant redimensionnable. Ce composant redimensionnable prend la forme paramétrée suivante : {ResizablePkg SubCompPkg Size Deploy Undeploy} où SubCompPkg désigne le type des sous-composants, et Size est une variable dynamique de type entier naturel. Enfin les paramètres Deploy et Undeploy désignent deux fonctions : la première est chargée de réaliser le déploiement d'une nouvelle instance de sous-composant, la

seconde est chargée de désinstaller une instance de sous-composant. Ces deux paramètres permettent notamment de spécialiser la manière dont les sous-composants sont instanciés, par exemple en utilisant différents allocateurs de machines, etc.

```
fun {ResizablePkg SubCompPkg Size Deploy Undeploy}
   functor $
   export Membrane
   define
      C = {CNew nil}
      %% Instantiate sub−components
      ...
      Membrane = C
   end
end
```

L'instanciation des sous-composants est assujettie à la variable dynamique Size, ce que nous mettons en œuvre par un observateur. L'observateur s'abonne aux mises à jour de la variable dynamique Size. Lors d'une mise à jour, ou bien la taille a été augmentée, et il faut dans ce cas déployer et intégrer de nouveaux sous-composants ; ou bien la taille a été diminuée, et il faut extraire et supprimer des sous-composants.

```
%% Listener class for a dynamic Size
class SizeListener from VariableListener
   attr currentSize

   %% Constructor
   meth init(Dest Source Sync)
      VariableListener,init(Dest Source Sync)
      currentSize := 0
   end

   meth update(NewSize)
      %% Ignore updates to 'undefined' or erroneous values
      if (NewSize \= undefined) andthen (NewValue>=0) then
         lock
            Diff = NewSize − @currentSize
         in
            if (Diff >= 0) then
               %% Size has been increased: deploy new sub−components
               for I in (@currentSize+1)..NewSize do
                  SubComp = {Deploy SubCompPkg}
                  Id = {VirtualString.toAtom 'subcomp'#I}
               in
                  {Tag SubCompC Id} % identify the sub−component with a tag 'subcomp<I>'
                  {CAddSubComponent C SubComp} % add the sub−component
               end
            else
               %% Size has been decreased: remove the corresponding sub−components
               for I in (NewSize+1)..@currentSize do
                  Id = {VirtualString.toAtom 'subcomp'#I} % select the sub−component with tag 'subcomp<I>'
                  SubComp = {SGetSingleton {SFilter {CGetSubComponents C} {TaggedWith Id}}}
               in
                  {Undeploy SubComp}
```

```
                        {CRemoveSubComponent C SubComp} % remove the sub−component
                    end
                end

                currentSize := NewSize % update the current size
            end % lock
        end % if
    end
end

%% Initialize the listener on the dynamic integer Size
{Size listen({New SizeListener init(nil Size sync)})}
```

Nous avons alors construit un composant redimensionnable paramétré par une taille dynamique qui peut être modifiée lors de l'exécution de la manière suivante. Le composant redimensionnable que nous exhibons déploie ses sous-composants localement grâce à la procédure de déploiement local Deploy. La suppression d'une instance de sous-composant ne requiert aucun traitement, d'où l'usage d'une procédure vide NullUndeploy.

```
% Size is a dynamic integer variable initialized to 1
Size = {VNew 1}

% Empty undeployment procedure
proc {NullUndeploy Comp} skip end

% Deploy the resizable component
Resizable = {Deploy {ResizablePkg SubCompPkg Size Deploy NullUndeploy}}
...
% Update the size of the component
{Size set(2)}
```

Schéma d'interconnexion dynamique

Nous avons précédemment montré comment mettre en œuvre des architectures paramétrées, en section 8.4.1, et intégrant notamment un paramètre décrivant le schéma d'interconnexion à mettre en place entre deux ensembles de composants. Sur le même principe, nous montrons ici comment réaliser une architecture exhibant un schéma d'interconnexion dynamique, capable de s'adapter aux dynamismes des ensembles de composants à interconnecter. Un schéma d'interconnexion est abstrait sous la forme d'une fonction de deux ensembles de composants à interconnecter, et générant un ensemble de descripteurs décrivant les liaisons à établir : $\mathcal{S}\{\mathcal{C}\} \times \mathcal{S}\{\mathcal{C}\} \rightarrow \mathcal{S}\{\text{descriptor}\}$.

L'interconnexion dynamique est donc paramétrée par deux ensembles dynamiques S_1 et S_2 de composants. Nous construisons ici un ensemble dynamique implicite de descripteurs de liaisons, calculé en fonction des deux ensembles dynamiques S_1 et S_2. Dans cet exemple, nous implantons l'interconnexion totale entre les deux ensembles de composants, ce qui revient à générer l'ensemble dynamique décrivant toutes les paires de composants $S_1 \times S_2$. En pratique, nous générons à partir des ensembles S_1 et S_2 un

ensemble de descripteurs de liaison. Un descripteur de liaison est un record contenant essentiellement une interface cliente et une interface destination :

```
Descriptor = desc(
        id: {NewName} % unique id
        iFrom: IFrom % client interface
        iTo: ITo % server interface
        binding: _ % real binding (unbound yet)
)
```

L'ensemble dynamique des descripteurs de liaisons est mise en œuvre au moyen de deux observateurs qui permettent de recevoir les mises à jour des deux ensembles de composants S_1 et S_2. Chaque mise à jour de l'un des ensembles déclenche l'ajout ou le retrait de descripteurs de liaisons correspondant aux composants ajoutés ou retirés de l'ensemble concerné.

```
local
    %% Implicit dynamic set class realizing the product set S1*S2 into a set of binding descriptor
    class BindingDescriptorSet from ImplicitSet
        attr s1 s2 % current views on S1 and S2
        feat l1 l2 % listeners on S1 and S2

        %% Constructor parametered with the two sets S1 and S2
        meth init(S1 S2)
            ImplicitSet,init
            %% S1 and S2 are initially assumed empty
            s1 := FSet.empty
            s2 := FSet.empty
            %% Set up listeners on S1 and on S2, which will foward updates to the
            %% methods addS1, removeS1, addS2 and removeS2 of this class
            self.l1 = {New SetListenerForwarder init(self S1 sync addS1 removeS1)}
            {S1 listen(self.l1)}
            self.l2 = {New SetListenerForwarder init(self S2 sync addS2 removeS2)}
            {S2 listen(self.l2)}
        end

        %% New elements added to S1
        meth addS1(SElements)
            lock
                s1 := {FSet.addAll @s1 SElements} % update our view of S1
                {FSet.forAll SElements
                    proc {$ CFrom} % for every new component CFrom added to S1
                        {FSet.forAll @s2
                            proc {$ CTo} % create a binding descriptor (CFrom, CTo) for any element CTo in S2
                                IFrom = {CGetInterface CFrom [client]}
                                ITo = {CGetInterface CTo [server]}
                            in
                                Set,add(desc(id:{NewName} iFrom:IFrom iTo:ITo binding:_))
                            end}
                    end}
            end
        end

        %% Elements removed from S1
```

```
        meth removeS1(SElements)
            lock
                s1 := {FSet.removeAll @s1 SElements} % update our view of S1
                {FSet.forAll SElements
                    proc {$ CFrom} % for each component CFrom removed from S1
                        Set,filterInPlace( % remove any descriptor refering to CFrom
                            fun {$ Desc}
                                ({IGetComponent Desc.iFrom} == CFrom)
                            end)
                    end}
            end
        end

        %% New elements added to S2
        %% Similar to addS1, except that we loop on @s1 to generate descriptors
        meth addS2(SElements) ... end

        %% New elements removed from S2
        %% Similar to removeS1, except that we filter on Desc.iTo
        meth removeS2(SElements) ... end
    end
in
    fun {DynamicFullInterconnect S1 S2}
        {New BindingDescriptorSet init(S1 S2)}
    end
end
```

Nous exploitons alors le schéma d'interconnexion dynamique dans une description d'architecture par un opérateur d'application du schéma d'interconnexion ApplyBindingScheme. Cet opérateur est mis en œuvre par un observateur abonné aux mises à jour opérant sur un ensemble dynamique de descripteurs de liaison. Lorsqu'un nouveau descripteur est généré et ajouté à l'ensemble dynamique des descripteurs, l'observateur réalise la liaison décrite par le descripteur. Symétriquement, lorsqu'un descripteur est retiré de l'ensemble, l'observateur identifie la liaison associée et la rompt.

```
proc {ApplyBindingScheme S1 S2 BindingScheme}
    %% Apply the binding scheme and generate a dynamic set of descriptors
    SDescriptor = {BindingScheme S1 S2}

    %% Listener class for a dynamic set of descriptors
    class SDescriptorListener from SetListener
        %% Adding some binding descriptors
        meth add(SElements)
            {FSet.forAll SElements % for every new descriptor
                proc {$ Desc} % create a binding and save its reference in the descriptor
                    Desc.binding = {BNew Desc.iFrom Desc.iTo}
                end}
        end

        %% Removing some binding descriptors
        meth remove(SElements)
            {FSet.forAll SElements % for every removed descriptor
                proc {$ Desc} % break and forget the corresponding binding
```

```
                {BBreak Desc.binding}
            end}
        end
    end
in
    %% Apply the descriptor listener on the generated dynamic set of descriptors
    {SDescriptor listen({New SDescriptorListener init(SDescriptor sync)})}
end
```

L'assemblage final au sein d'une description d'architecture dynamique est finalement réalisé comme suit :

```
fun {ParameteredComposite SubCompPkg1 SubCompPkg2 Size1 Size2 BindingScheme}
    functor $
    export Membrane
    define
        C = {CNew nil}

        %% Deploy two resizable components
        RC1 = {Deploy {ResizablePackage SubCompPkg1 Size1 Deploy NullUndeploy}}
        RC2 = {Deploy {ResizablePackage SubCompPkg2 Size2 Deploy NullUndeploy}}

        %% Extract two sets of processing units
        S1 = {SFilter {CGetSubComponents RC1} {TaggedWith 'Processing_Unit'}}
        S2 = {SFilter {CGetSubComponents RC2} {TaggedWith 'Processing_Unit'}}

        %% Apply the binding scheme between the two processing unit sets
        {ApplyBindingScheme S1 S2 BindingScheme}

        Membrane = C
    end
end
```

9.5 Primitives de la bibliothèque LactOz

Nous transposons dans l'environnement de calcul dynamique l'ensemble traditionnel des primitives de calcul sur les valeurs primitives (booléens, nombres entiers et flottants, etc).

9.5.1 Notations

La table 9.1 présente les notations que nous utilisons dans la suite pour exprimer les informations de typage des primitives de calcul et de navigation dynamique. Ces notations s'inspirent des notations utilisées pour typer les valeurs dans les langages de type ML. Néanmoins, les informations de typage sont données à titre purement informatif afin de préciser la nature des paramètres des primitives. En particulier, le typage n'est vérifié ni à la compilation ni à l'exécution dans notre implantation.

$'a$	type non spécifié statique (p. ex. un entier statique traditionnel)
$'A$	type non spécifié dynamique (p. ex. un entier dynamique)
$X \rightarrow Y$	Fonction d'un élément de type X vers un élément de type Y
$\mathbb{B}$	Booléen
$\mathbb{N}$	Valeur numérique (entière $\mathbb{Z}$ ou flottante $\mathbb{R}$)
$\mathcal{S}$	Ensemble dynamique
$\mathcal{S}\{X\}$	Ensemble dynamique d'éléments de type X
$\mathcal{C}$	Composant (membrane)
$\mathcal{I}$	Interface
$\mathcal{B}$	Liaison

TAB. 9.1 – Notations utilisées pour le typage des primitives.

9.5.2 Table de référence

La table de référence présente l'ensemble des primitives les plus générales et les plus expressives. De multiples versions des primitives citées peuvent exister afin d'optimiser les expressions dynamiques construites en fonction du contexte. Par exemple, l'opérateur "et" booléen BAnd citée opère de manière générale sur un ensemble dynamique de booléens dynamiques. En réalité, la bibliothèque LactOz inclut plusieurs versions de cet opérateur, dans le cas d'un ensemble statique de booléens dynamiques, dans le cas d'un ensemble dynamique de booléens statiques, etc.

La table de référence inclut les primitives primordiales de navigation qui ont déjà été présentées dans le contexte du canevas FructOz. En composant ces primitives primordiales, nous construisons des primitives étendues de navigation dans les architectures dynamiques plus expressives et extensibles à souhait. La table de référence n'inclut qu'un nombre restreint de ces primitives étendues.

La première lettre du nom d'une primitive indique la nature des principaux paramètres de la primitive (p. ex. S pour un ensemble (*Set*), ou C pour un composant, etc). Certaines des primitives se déclinent selon un critère de spécialisation. Les noms de ces primitives sont parfois factorisés comme dans CGet(Sub,Super)Components, qui désigne les deux primitives CGetSubComponents et CGetSuperComponents, respectivement spécialisées pour les sous-composants et pour les super-composants (ou composants parents).

TAB. 9.2: Table de référence des primitives de calcul dynamique et de navigation dans les architectures dynamiques.

Primitive	Description
Calcul sur les valeurs dynamiques booléennes et numériques	
Unary	Construction d'opérateur unaire dynamique à partir d'un opérateur unaire traditionnel : $('a \rightarrow' b) \rightarrow ('A \rightarrow' B)$
Binary	Construction d'opérateur binaire dynamique à partir d'un opérateur binaire traditionnel : $('a \times 'b \rightarrow' c) \rightarrow ('A \times 'B \rightarrow' C)$
Nary	Construction d'opérateur n-aire dynamique à partir d'un opérateur n-aire traditionnel : $('a \text{ list} \rightarrow' b) \rightarrow ('A \text{ list} \rightarrow' B)$
Reduce	Construction d'opérateur de réduction d'un ensemble de valeurs (e.g. somme ou produit d'un ensemble de nombres)
BNot	Négation logique : $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
BAnd et BOr	Opérateur «et» et «ou» logique : $\mathcal{S}\{\mathbb{B}\} \rightarrow \mathbb{B}$
NNegate, NInvert, NCos, etc	Opposé, inverse, cosinus, etc (construits avec l'opérateur Unary) : $\mathbb{N} \rightarrow \mathbb{N}$
NSubtract, NDiv, NPower, etc	Soustraction, division, etc (construits avec l'opérateur Binary) : $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
NSum, NMul, etc	Somme, produit, etc (construits avec l'opérateur Reduce) : $\mathcal{S}\{\mathbb{N}\} \rightarrow \mathbb{N}$
Calcul sur les ensembles dynamiques	
SNew	Nouvel ensemble dynamique vide : $unit \rightarrow \mathcal{S}$
SUnion, SIntersect	Union et intersection d'un ensemble dynamique d'ensembles : $\mathcal{S}\{\mathcal{S}\} \rightarrow \mathcal{S}$
SSubtract	Différence de deux ensembles : $\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$

Suite en page suivante...

TAB. 9.2 – *Suite*

Primitive	Description
SMap	Ensemble dynamique obtenu par application d'une fonction sur un ensemble dynamique ; cette opération fait l'hypothèse que la fonction est déterministe, c'est-à-dire que l'application de la fonction à un même élément renverra toujours le même résultat : $\mathcal{S}\{'a\} \times ('a \rightarrow' b) \rightarrow \mathcal{S}\{'b\}$
SFilter	Sous-ensemble dynamique par filtrage (prédicat) d'un ensemble dynamique : $\mathcal{S}\{'A\} \times ('A \rightarrow \mathbb{B}) \rightarrow \mathcal{S}\{'A\}$
SSize	Taille d'un ensemble dynamique : $\mathcal{S} \rightarrow \mathbb{Z}$
SIsEmpty	Prédicat testant si un ensemble dynamique est vide ou non : $\mathcal{S} \rightarrow \mathbb{B}$
SHasSome, SHasAll	Prédicats testant si certains (resp. tous les) éléments d'un ensemble dynamique satisfont un prédicat dynamique donné : $\mathcal{S} \times ('A \rightarrow \mathbb{B}) \rightarrow \mathbb{B}$
Primitives primordiales de navigation dynamique	
CGetInterfaces	Ensemble des interfaces d'un composant : $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{I}\}$
CGet(Sub,Super)Components	Ensemble des sous-composants (resp. super-composants ou composants parents) du composant donné : $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{C}\}$
CListAttributes	Ensemble des attributs du composant donné : $\mathcal{C} \rightarrow \mathcal{S}\{?\}$
CGetAttribute	Valeur associée à un attribut du composant donné : $\mathcal{C} \times ? \rightarrow ?$
IIsBound(From,To)	Test de l'existence d'une liaison depuis (resp. à destination de) l'interface donnée $\mathcal{I} \rightarrow \mathbb{B}$
IGetComponent	Composant possédant l'interface spécifiée : $\mathcal{I} \rightarrow \mathcal{C}$
IGetBindings(From,To)	Ensemble des liaisons depuis (resp. à destination de) l'interface donnée : $\mathcal{I} \rightarrow \mathcal{S}\{\mathcal{B}\}$
BGet(Client,Server)Interface	Interface client (resp. serveur) participant à la liaison donnée : $\mathcal{B} \rightarrow \mathcal{I}$

Suite en page suivante...

TAB. 9.2 – *Suite*

Primitive	Description
Primitives étendues de navigation dynamique	
CGet(Client,Server)Interfaces	Ensemble des interfaces clients (resp. serveurs) du composant donné : $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{I}\}$
CGet(Int,Ext)ernalBindings	Ensemble des liaisons internes (resp. externes) depuis et à destination du composant donné : $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{B}\}$
CGet(Int,Ext)ernalBindings(From,To)	Ensemble des liaisons internes (resp. externes) depuis un (resp. à destination du) composant donné : $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{B}\}$
CGet(Int,Ext)ernal.. ..ComponentsBound(With,From,To)	Ensemble des composants internes (resp. externes) au composant donné, et directement liés depuis et/ou à destination du composant donné : $\mathcal{C} \rightarrow \mathcal{S}\{\mathcal{C}\}$
IGet(Int,Ext)ernalBindings	Ensemble des liaisons internes (resp. externes) associées à l'interface donnée : $\mathcal{I} \rightarrow \mathcal{S}\{\mathcal{B}\}$
IIsBound(Int,Ext)ernally	Test de l'existence d'une liaison interne (resp. externe) associée à l'interface donnée : $\mathcal{I} \rightarrow \mathbb{B}$
BGet(Client,Server)Component	Composant client (resp. serveur) dans la liaison donnée : $\mathcal{B} \rightarrow \mathcal{C}$

9.6 Synthèse

Nous avons présenté dans ce chapitre la bibliothèque LactOz spécialisée dans l'observation des architectures dynamiques et qui constitue un complément au canevas FructOz pour la construction de systèmes distribués fondés sur des architectures dynamiques. La bibliothèque LactOz repose sur un modèle de calcul dynamique qui permet de construire des raisonnements sur des éléments dynamiques tels que des architectures dynamiques. Grâce à LactOz, ces raisonnements conservent leur cohérence et restent valides malgré le dynamisme des valeurs effectives des éléments manipulés. Il devient alors possible de parler d'un ensemble de composants, de le désigner, d'en calculer des sous-ensembles, etc, indépendamment du contenu réel effectif a priori inconnu de cet ensemble. En effet, le contenu de cet ensemble ne sera déterminé qu'au moment de l'exécution et pourra continuer à évoluer durant l'exécution. LactOz garantit que les raisonnements fondés sur cet ensemble seront mis et maintenus en cohérence au fur et à mesure de ses évolutions.

Nous avons montré, au moyen de quelques exemples génériques, comment combiner la bibliothèque LactOz avec le canevas FructOz pour réaliser des architectures dynamiques paramétrées. Nous avons notamment détaillé l'architecture d'un composant redimensionnable exhibant un paramètre dynamique correspondant au nombre de ses sous-composants, et nous avons également détaillé la réalisation d'un schéma d'interconnexion dynamique entre deux ensembles de composants. Ces quelques briques architecturales peuvent être abstraites et étendues pour former une bibliothèque de modèles d'architectures réutilisables dans le cadre de la construction d'architectures dynamiques plus complexes. Une évaluation de la bibliothèque LactOz est donnée au chapitre 11.

L'utilisation de la bibliothèque LactOz conduit à la construction d'un système évènementiel distribué implantant les valeurs dynamiques explicites et implicites utilisées dans les expressions LactOz. Ce système évènementiel distribué offre des perspectives d'optimisation intéressantes. Il est notamment envisageable d'étudier la factorisation et la réutilisation des calculs dynamiques intermédiaires. D'un point de vue génie logiciel, l'utilisation intensive des primitives LactOz dans le contexte de la plate-forme Mozart/Oz révèle une faiblesse du langage Oz en l'absence de typage statique vérifiable à la compilation. Or la grande majorité des primitives LactOz ne tire aucun bénéfice de l'avantage du typage dynamique et pourrait profiter d'une vérification, même limitée. Il serait donc intéressant d'étudier l'intégration d'un typage partiel (*soft types*) dans le langage Oz afin de concilier la liberté liée au typage dynamique et la sécurité obtenue par des vérifications statiques lorsqu'elles sont possibles. L'implantation des calculs dynamiques a également révélé une limitation des ramasse-miettes, liée à la nature active des valeurs dynamiques mises en œuvre dans LactOz. Le graphe des valeurs impliquées dans l'instanciation d'une expression dynamique LactOz met en scène un jeu complexe de références fortes et faibles entre les différentes valeurs dynamiques intermédiaires. Toutefois, la nature active des threads qui implantent les procédures de

rappel que nous mettons en œuvre pourrait parfois installer des situations de famine qui sollicitent régulièrement certaines références faibles et empêchent ainsi le ramasse-miette de collecter certaines valeurs dynamiques inutilisées. Nous n'avons pas trouvé à ce jour de solution satisfaisante à ce problème, hors une gestion explicite de la création et de la destruction des expressions dynamiques.

La partie suivante rassemble les expérimentations et évaluations que nous avons conduites dans le cadre de la réalisation de nos travaux. Ces évaluations visent à démontrer l'intérêt et/ou les limitations de nos propositions pour la construction de systèmes distribués auto-optimisés.

Troisième partie

Expérimentations et évaluations

Chapitre 10

Évaluation des politiques d'auto-optimisation

Nous présentons ici un ensemble de résultats démontrant la pertinence et l'efficacité de l'auto-optimisation, par le biais de politiques d'optimisation améliorant au mieux les performances des systèmes administrés par des techniques d'approvisionnement dynamique, et ensuite grâce à des politiques d'optimisation garantissant le niveau de performance du système administré.

10.1 Heuristiques pour optimisation au mieux

Nous validons notre politique d'optimisation par approvisionnement dynamique sur nos deux contextes d'application : les services à messages et les services Internet.

10.1.1 Service à messages

L'évaluation suivante s'intéresse aux performances et à l'auto-optimisation de systèmes mettant en œuvre des services à messages. Dans un premier temps, nous effectuons une série de mesures afin d'évaluer les performances de l'intergiciel supportant les services à messages et afin d'identifier les facteurs déterminants pour les performances des systèmes administrés. Dans un second temps, nous évaluons la pertinence de la mise en œuvre de l'auto-optimisation sur ces systèmes.

Environnement matériel. Les expérimentations relatives aux services à messages présentées dans cette section ont été conduites sur une grappe de machines Mac Mini compatibles x86, équipées de processeurs Intel Core Duo fonctionnant à 1.66GHz, disposant chacune de 2Go de mémoire vive et interconnectées par un réseau Ethernet 1Gb/s.

Environnement logiciel. La grappe de machines fonctionne sous un environnement Mac OS X en version 10.4.7, offrant l'environnement Java 2 Sun en version 1.4.2_13 et hébergeant l'intergiciel JORAM [33] en version 4.3.21.

Application. Les expérimentations réalisées ici ont été conduites avec des bancs d'essai synthétiques mis en œuvre par nos propres moyens. Ces bancs d'essai ont été déployés puis exécutés sur le système à messages en grappe présenté à la section 7.1.3 (page 85) et intégrant des capacités d'approvisionnement dynamique. Nous avons utilisé les sondes fournies par l'intergiciel JORAM et accessibles grâce à l'interface JMX qu'il intègre. Nous avons notamment surveillé les attributs JMX *NbMsgsDeliverSinceCreation* et *MessageCounter* des files d'attente, qui indiquent respectivement le nombre de messages lus par des consommateurs dans la file d'attente depuis sa mise en service, et le nombre de messages actuellement présents et en attente dans la file d'attente. Ces sondes JMX ont été sollicitées périodiquement chaque seconde durant nos expérimentations. Au cours de ces expérimentations, nous avons utilisé des messages d'une taille de 1Ko. L'interconnexion entre les machines n'est pas limitante. Afin d'obtenir des résultats cohérents, chaque expérimentation a été répétée trois fois, et les résultats exploités correspondent à la moyenne obtenue sur les trois exécutions.

Approvisionnement dynamique d'une file à messages en grappe

Observations et mesures préliminaires. Cette évaluation vise à démontrer l'impact du nombre de messages en attente dans une file JORAM sur les performances du service à messages. Au cours d'une première étape, un client producteur de messages adresse 1500 messages à une file d'attente unique ; dans une seconde étape, un client consommateur de messages extrait et consomme les messages stockés dans la file. La figure 10.1 présente le comportement de la file d'attente lors de cette expérimentation. Nous observons que le nombre de messages accumulés par une file d'attente a un impact fort sur les performances. Les débits accessibles en production et en consommation de messages diminuent à mesure que les messages s'accumulent dans la file d'attente.

La figure 10.2 présente les performances d'une file d'attente simple en fonction du nombre de ses clients (producteurs et consommateurs). Ces résultats démontrent l'existence d'un nombre optimal de clients en dehors duquel les performances de la file d'attente sont amoindries. La configuration optimale mesurée ici est de 12 clients avec un débit maximal de production et de consommation de message avoisinant les 1.8 messages/ms. À titre d'information, les clients de la file d'attente sont ici en proportion d'un producteur de messages pour deux consommateurs ce qui permet d'assurer la stabilité de la file d'attente.

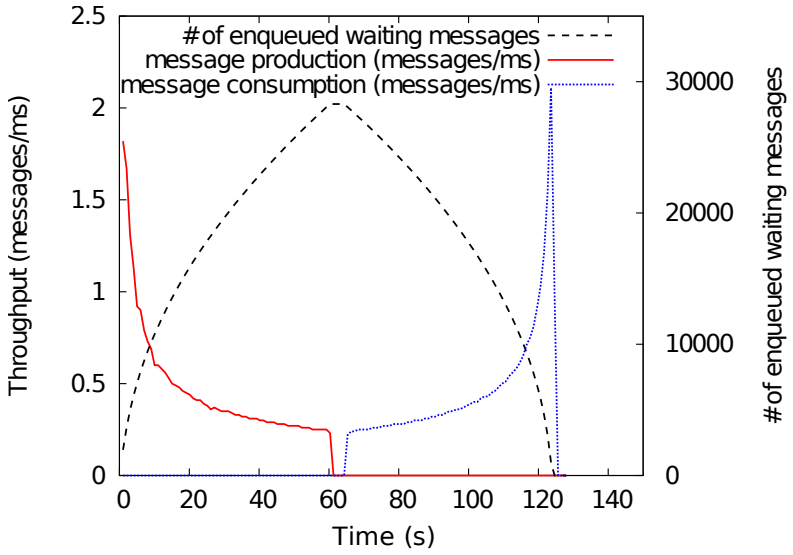


FIG. 10.1 – Impact de l'accumulation de messages dans une file d'attente sur le débit de messages.

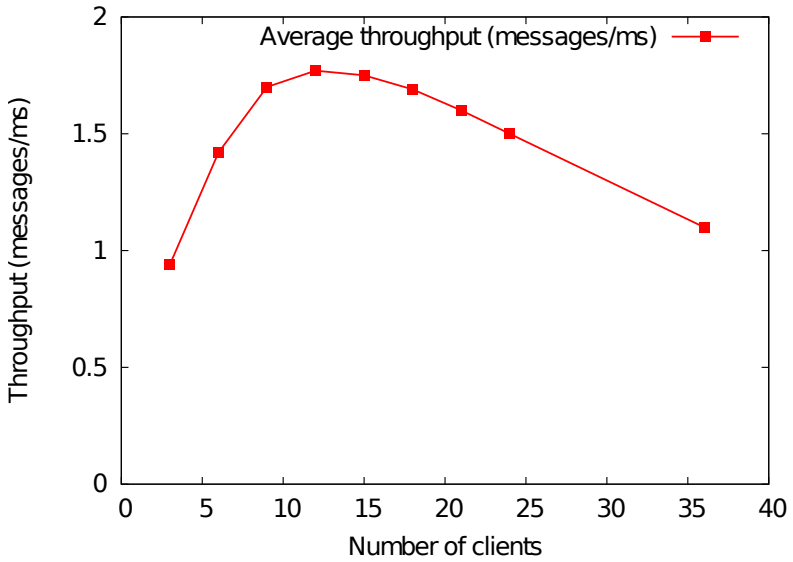


FIG. 10.2 – Configuration optimale d'une file d'attente simple.

Ces observations des comportements des files d'attente simples justifient l'intérêt des files d'attente en grappe et de leur auto-optimisation par approvisionnement dynamique dans l'objectif d'assurer des performances toujours optimales, et notamment en maintenant la stabilité des files d'attente pour éviter ainsi l'accumulation de messages dégradant les performances des services à messages.

Approvisionnement dynamique d'une file d'attente en grappe. Nous présentons ici une évaluation destinée à démontrer la pertinence de l'auto-optimisation dans le contexte des services à messages par approvisionnement dynamique des files d'attente en grappe, en ajustant dynamiquement le nombre de files d'attente simples participant à la file d'attente en grappe en fonction de la charge de travail imposée au service.

La charge de travail générée sur le service à messages s'organise autour de 5 clients producteurs de messages générant chacun 10000 messages, et de 10 clients consommateurs de messages consommant chacun 5000 messages. Afin de générer une charge de travail dynamique croissante, les différents clients sont progressivement activés l'un après l'autre, en ajoutant un nouveau client toutes les 10s. La stabilité de la file d'attente en grappe est assurée en maintenant une proportion adaptée entre producteurs et consommateurs de messages. La file d'attente en grappe est initialement composée d'une unique file d'attente.

Les figures 10.3 et 10.4 présentent respectivement le comportement d'une file d'attente non auto-optimisée et statiquement approvisionnée, et le comportement d'une file d'attente en grappe auto-optimisée et dynamiquement approvisionnée comme détaillé précédemment. La file d'attente en grappe non auto-optimisée est statiquement approvisionnée et repose sur un unique serveur durant l'intégralité de l'expérimentation et indépendamment du nombre de clients du service à messages. Les mesures montrent la stabilisation du comportement du service à messages vers 50s après le début de l'expérimentation à environ 1.9 message/ms, et ce, jusqu'à la fin de l'expérimentation. Lorsque la file d'attente est auto-optimisée et dynamiquement approvisionnée, le comportement de la file d'attente est similaire à celui de la file d'attente non auto-optimisée aussi longtemps que la charge de travail n'excède pas la capacité d'une file d'attente simple. Ensuite, vers 120s après le début de l'expérimentation, la charge de travail dépasse la capacité d'une file d'attente simple, ce qui déclenche une opération d'approvisionnement dynamique pour ajouter un serveur à la file d'attente en grappe, vers lequel les nouveaux clients sont dirigés. De manière attendue, les performances de la file d'attente en grappe sont quasiment doublées, passant de 1.9 messages/ms à 3.7 messages/ms. Au final, alors que le système non optimisé achève le traitement de tous les messages en 300 secondes environ, le système optimisé traite l'ensemble des messages en un peu plus de 200 secondes.

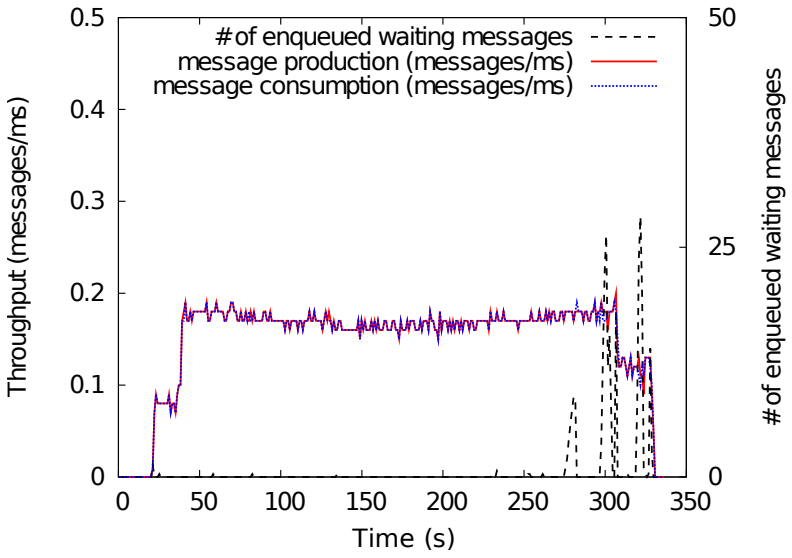


FIG. 10.3 – Comportement de la file d'attente sans approvisionnement dynamique (restreinte à une seule file d'attente).

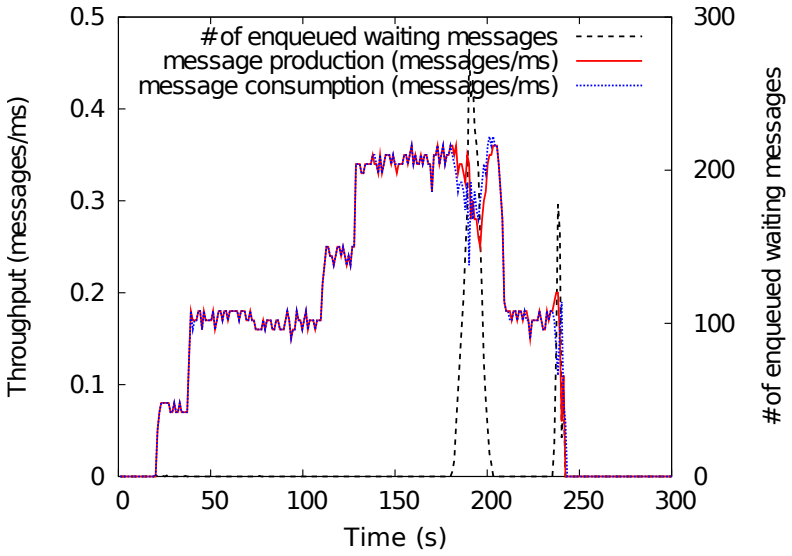


FIG. 10.4 – Comportement de la file d'attente avec approvisionnement dynamique (une à deux files d'attente).

L'évaluation précédente montre la faisabilité et l'intérêt de l'auto-optimisation pour les services à messages. Les sections suivantes transposent cette évaluation sur des systèmes implantant des services Internet, démontrant ainsi la généralité des concepts sous-jacents à l'auto-optimisation que nous mettons en œuvre.

10.1.2 Service Internet en grappe

Environnement matériel. Nos évaluations concernant l'auto-optimisation des Services Internet en grappe ont été conduites sur une grappe de machines compatibles x86 équipées de bi-processeurs Xeon fonctionnant à 1.8GHz, comportant 1Go de mémoire vive et interconnectées par un réseau Ethernet 100Mb/s.

Environnement logiciel. La grappe de machines fonctionne sous un environnement Linux 2.4 avec les intergiciels J2EE suivants : le serveur Web Apache HTTPD [47] en version 1.3.9, le serveur d'applications Apache Tomcat [49] en version 3.3.2, le serveur de bases de données MySQL [1] en version 4.0.17, le répartiteur de charge Web PLB [42] en version 0.3, le répartiteur de charge intégré au serveur d'applications Apache Tomcat, et enfin le répartiteur de charge de bases de données C-JDBC [24] en version 2.0.2.

Application. Nous avons réalisé nos évaluations au moyen du banc d'essai Rubis [8] en version 1.4.2, implantant un service Internet d'enchères en ligne sous forme d'un système multi-étagé J2EE en grappe et modélisé d'après le site d'enchères eBay [45]. Le service Internet Rubis spécifie et met en œuvres plusieurs types d'interactions, comme par exemple l'enregistrement de nouveaux utilisateurs du service, l'exploration des différents objets en vente sur le service, ou encore l'achat et la mise en vente d'objets. Par ailleurs, le banc d'essai Rubis intègre un outil de mesure de performances du service Internet, capable d'émuler le comportement d'un ensemble d'utilisateurs du service afin de soumettre le service Internet à une charge de travail paramétrable. Le banc d'essai Rubis fournit en outre deux types de charge de travail préconfigurés : le premier émule une charge de travail induisant exclusivement des opérations en lecture seulement sur le service, tandis que le second émule une charge de travail induisant 15% d'opérations en écriture sur le service. Lors d'une simulation, l'outil de mesures de performances permet de collecter différentes informations relatives au comportement du service Internet.

Nous avons déployé le service Internet Rubis sur le système J2EE dynamique multi-étagé en grappe présenté à la section 7.1.3 (page 87), composé d'une grappe de serveurs frontaux réalisant l'étage Web et Application du système au moyen de serveurs Tomcat, et d'une grappe de serveurs réalisant l'étage de Bases de données du système par des serveurs MySQL. Ce système intègre des capacités d'auto-optimisation par approvisionnement dynamique, lui permettant d'adapter dynamiquement le nombre de ressources mobilisées pour chacun des deux étages du système, et fournit un ensemble de sondes appropriées à la mise en œuvre de l'auto-optimisation.

Nous présentons maintenant les résultats principaux obtenus à l'issue de différents scénarios d'expérimentation réalisés sur l'environnement décrit précédemment. L'objectif de cette évaluation est de démontrer la validité et l'efficacité de notre approche pour la mise en œuvre de l'auto-optimisation fondée sur l'approvisionnement dynamique. Dans cette optique, chaque évaluation inclut une comparaison du système auto-optimisé avec le même système statique et non auto-optimisé. Cette dernière configu-

ration statique du système correspond au système J2EE multi-étagé présenté sur la figure 7.8 (page 87) composé d'une unique instance du serveur Apache Tomcat ainsi que d'une unique instance du serveur MySQL ; cette configuration correspond également à l'état initial du système auto-optimisé.

Gestion de variations graduelles de charge

Nous présentons ici une expérimentation démontrant la capacité de notre approche à traiter des variations graduelles de la charge. Dans cet objectif, nous soumettons le service Internet du banc d'essai Rubis hébergé par l'environnement dynamique décrit précédemment à une charge de travail décrivant des variations graduelles, afin de mettre en évidence la faculté du système ainsi auto-optimisé à s'adapter en suivant ces variations graduelles de charge de manière appropriée.

L'allure de la charge de travail appliquée au système est illustrée sur la figure 10.5. L'intensité de la charge de travail est initialement faible et correspond à une charge générée par 80 clients du service Internet. La charge de travail augmente alors progressivement par ajouts successifs de 20 clients toutes les minutes, jusqu'à atteindre un maximum de 500 clients. Ensuite, la charge de travail décroit symétriquement pour retrouver son niveau initial de 80 clients. Ce schéma de variation de charge dynamique correspond à un scénario conforme au fonctionnement normal d'un service en activité durant la journée, pour lequel le nombre d'utilisateurs augmente progressivement pendant la matinée, pour atteindre son maximum en milieu de journée, et diminue ensuite en soirée.

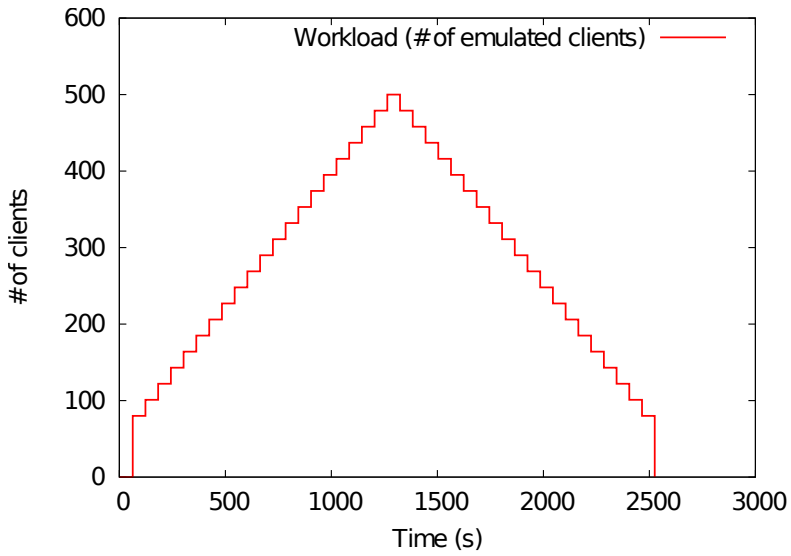


FIG. 10.5 – Variation graduelle de la charge appliquée au service Internet dynamique administré.

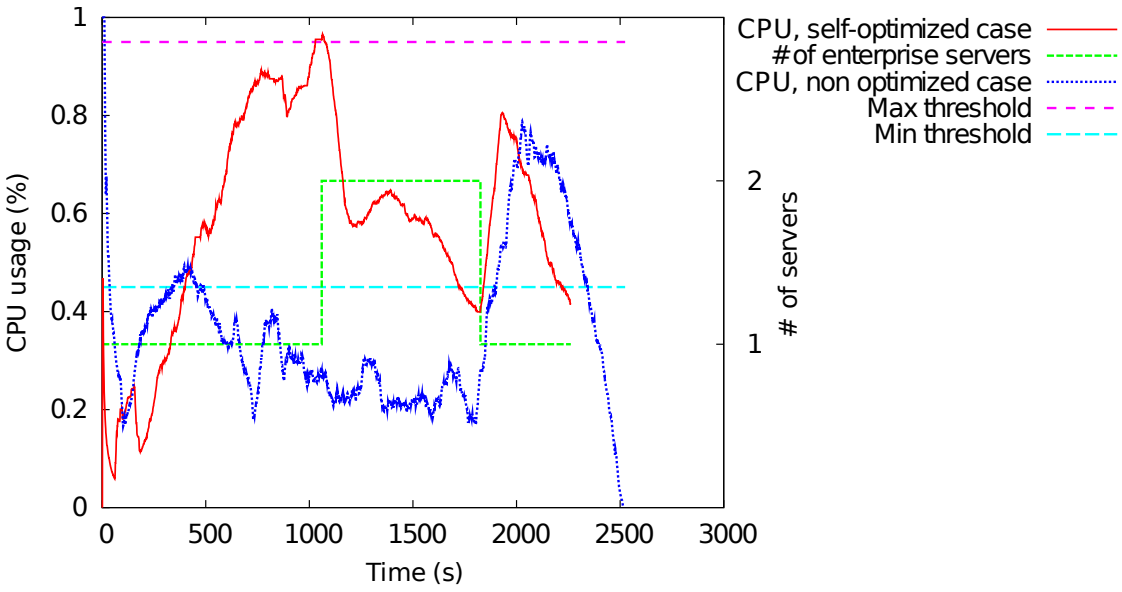


FIG. 10.6 – Réponse de l'étage Web et Application à la variation graduelle de charge générée, avec et sans auto-optimisation.

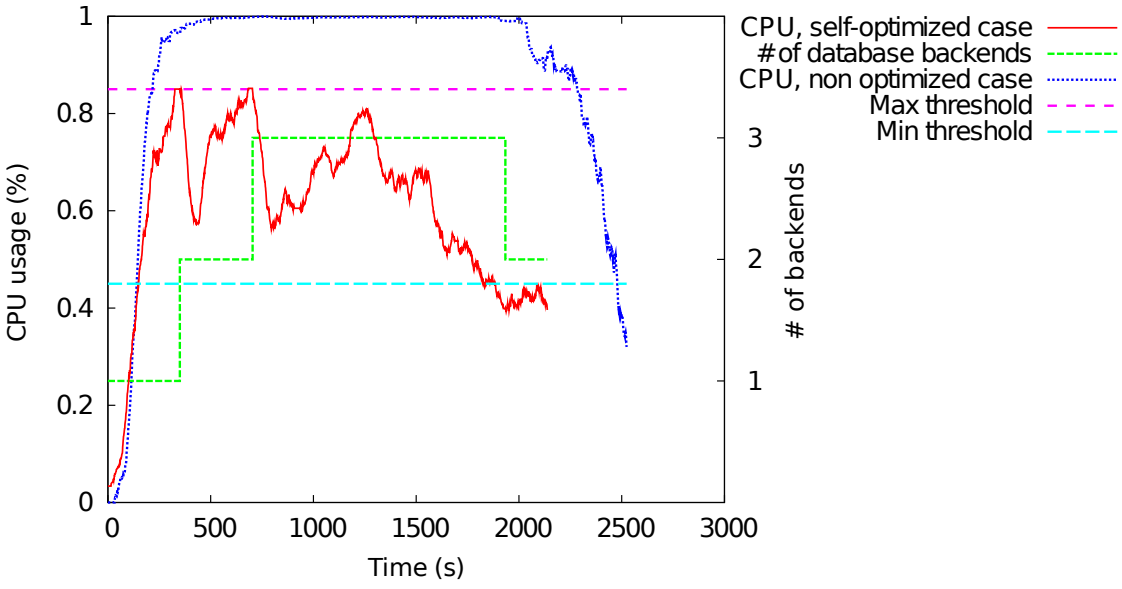


FIG. 10.7 – Réponse de l'étage Bases de données à la variation graduelle de charge générée, avec et sans auto-optimisation.

Les figures 10.6 et 10.7 présentent les niveaux de consommation agrégée des ressources des serveurs des bases de données et des serveurs d'applications qui hébergent le service Internet Rubis au cours de l'expérimentation. Ces figures représentent le comportement du système non auto-optimisé et statiquement approvisionné ainsi que celui du système auto-optimisé et dynamiquement approvisionné. Ces figures affichent en outre les seuils minimum et maximum qui ont été utilisés pour piloter la boucle d'auto-optimisation. Enfin, ces figures font également figurer les niveaux d'approvisionnement du système administré, c'est-à-dire le nombre de machines allouées aux serveurs de bases de données ainsi qu'aux serveurs d'applications. Durant cette expérimentation, le processeur est l'unique ressource limitante du système. C'est pourquoi seul le niveau de consommation des processeurs est représenté sur ces figures.

Lorsque le système hébergeant le service Internet Rubis n'est pas auto-optimisé et est statiquement approvisionné, nous observons que le service entre en surcharge assez rapidement, ce qui se traduit sur la figure par une saturation nette des ressources (ici, le processeur). À l'inverse, le système équivalent auto-optimisé et dynamiquement approvisionné détecte la surcharge lorsque le niveau de consommation des ressources dépasse le seuil maximum paramétré et déclenche alors des augmentations unitaires de l'approvisionnement du système en ressources. Cette augmentation de ressources induit une diminution visible du niveau de consommation des ressources agrégé, ce qui permet ainsi de maintenir le service en état de fonctionnement, alors que le service non auto-optimisé s'écroule.

La figure 10.8 résume l'état d'approvisionnement de chacun des deux étages composant le système dynamique au cours de l'expérimentation. Le mécanisme d'approvisionnement dynamique est déclenché à trois reprises durant l'augmentation de la charge reproduite au cours de l'expérimentation : les deux premières activations de l'approvisionnement dynamique concernent l'étage de bases de données, conduisant alors à un déplacement du goulot d'étranglement du système sur les serveurs d'applications, qui profitent alors de la troisième activation de l'approvisionnement dynamique. Symétriquement, alors que la charge de travail diminue progressivement, le niveau de consommation des ressources agrégé chute et tombe finalement sous le seuil minimum configuré dans la boucle d'auto-optimisation. Cette violation du seuil est détectée et conduit alors à une diminution de l'approvisionnement du système conformément à nos attentes.

La figure 10.9 résume sur un histogramme la distribution des temps de réponse mis par le service Internet pour traiter les requêtes utilisateur sur l'ensemble de l'expérimentation avec et sans auto-optimisation. Sur cette figure, nous pouvons lire que la proportion des requêtes traitées en moins d'une seconde atteint 86.25% lorsque le système est auto-optimisé, contre seulement 36.17% en l'absence d'auto-optimisation. La proportion des requêtes traitées en plus d'une seconde et moins de 20 secondes est seulement de 13.47% avec auto-optimisation contre 48.28% sans. Enfin, aucune requête n'est traitée en plus de 40 secondes lorsque le système est auto-optimisé, contre 5.5% des requêtes lorsque le système est statiquement configuré.

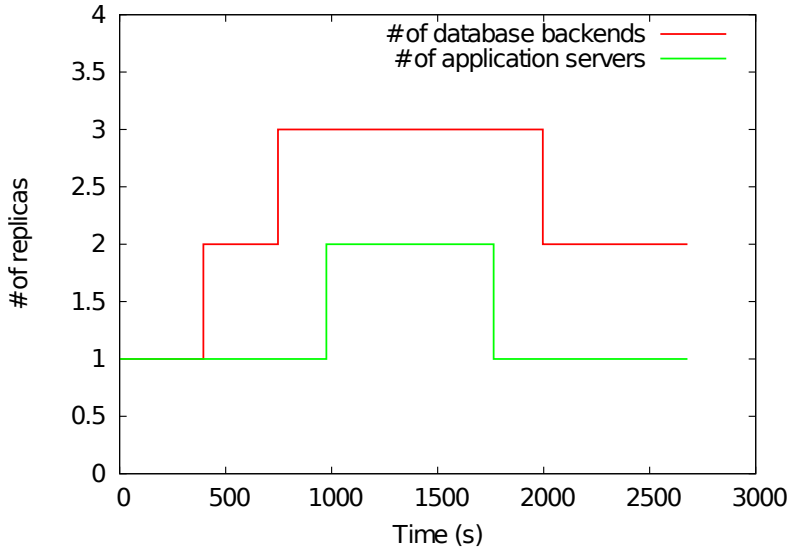


FIG. 10.8 – Approvisionnement dynamique en ressources de chacun des deux étages du système administré.

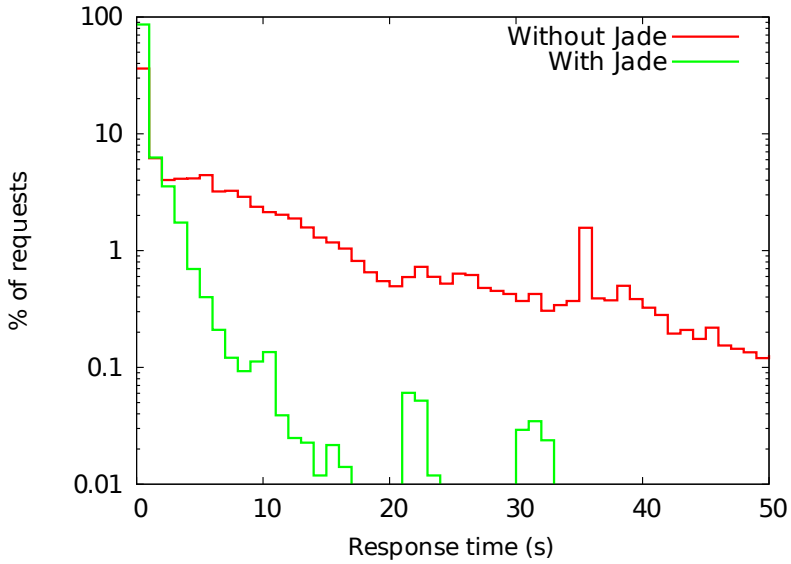


FIG. 10.9 – Histogramme des temps de réponse aux requêtes utilisateur sur l'ensemble de l'expérimentation, avec et sans auto-optimisation.

Les différents résultats présentés ici démontrent le gain significatif apporté par notre politique d'auto-optimisation fondée sur les opérations unitaires d'approvisionnement dynamique dans le cas de variations graduelles de la charge. Le système pouvant toutefois être sujet à d'autres formes de variations de charge, nous évaluons maintenant notre politique destinée au traitement des variations brutales de la charge.

Gestion de variations brutales de charge

Nous détaillons dans cette section une expérimentation visant à démontrer l'efficacité de notre approche pour l'approvisionnement dynamique du système en cas de variation brutale de la charge appliquée au système. À ce titre, nous exposons le service Internet Rubis à un pic de charge qui dépasse considérablement la capacité de traitement initiale du système J2EE dynamique, composé initialement comme dans l'expérimentation précédente d'une instance de serveur Tomcat ainsi que d'une instance de serveur MySQL.

La figure 10.10 présente l'allure de la charge de travail que nous appliquons au service Internet Rubis. L'expérimentation débute avec une charge modérée correspondant à 100 utilisateurs du service. Trois minutes (soit 180 secondes) après le début de l'expérimentation, la charge de travail monte brutalement à 500 clients, émulant ainsi un pic de charge que le système ne peut convenablement tolérer avec sa capacité de traitement initiale. La figure 10.11 présente le comportement du service Internet en réponse au pic de charge ainsi généré, d'une part lorsque le système est auto-optimisé et dynamiquement approvisionné, et d'autre part lorsque le système est non auto-optimisé et statiquement approvisionné. Le comportement du service Internet est caractérisé sur cette figure par les temps de réponse des requêtes des utilisateurs du service. Plus spécifiquement, un point représenté sur la figure correspond exactement à une requête utilisateur soumise au service ; l'abscisse du point correspond à la date à laquelle la requête a été soumise au service, tandis que son ordonnée représente le temps mis par le service Internet Rubis pour traiter la requête et transmettre une réponse au client.

La charge de travail initialement appliquée au service Internet est correctement satisfaite par le système, ce qui se traduit sur la figure par des temps de réponse faibles. Dès la survenue du pic de charge, nous observons une nette et franche augmentation des temps de réponse. En réalité, cette augmentation brutale des temps de réponse révèle l'écroulement du système qui n'est alors plus capable de traiter convenablement les requêtes des utilisateurs du service. Lorsque le système est statiquement approvisionné, l'écroulement du système est très marqué, si bien que de nombreuses requêtes soumises après la survenue du pic de charge n'obtiennent une réponse qu'après la fin de l'expérimentation. Par exemple, les requêtes soumises 200 secondes après le début de l'expérimentation montrent des temps de réponse dépassant les 1400 secondes, signifiant ainsi que la réponse à ces requêtes a été délivrée 1600 secondes après le début de l'expérimentation, c'est-à-dire encore 700 secondes après la fin de l'expérimentation. En comparaison, le système auto-optimisé et dynamiquement approvisionné détecte rapidement le pic de charge par l'observation des temps de réponse aux requêtes

des utilisateurs. L'auto-optimisation réagit au pic de charge et déclenche immédiatement l'approvisionnement de deux serveurs supplémentaires assignés à l'étage bases de données. Lorsque ces serveurs sont effectivement intégrés au système en fonctionnement, les requêtes des utilisateurs sont à nouveau correctement traitées, ce qui se traduit sur la figure 10.11 par des temps de réponse à nouveau faibles à partir de 300 secondes après le début de l'expérimentation.

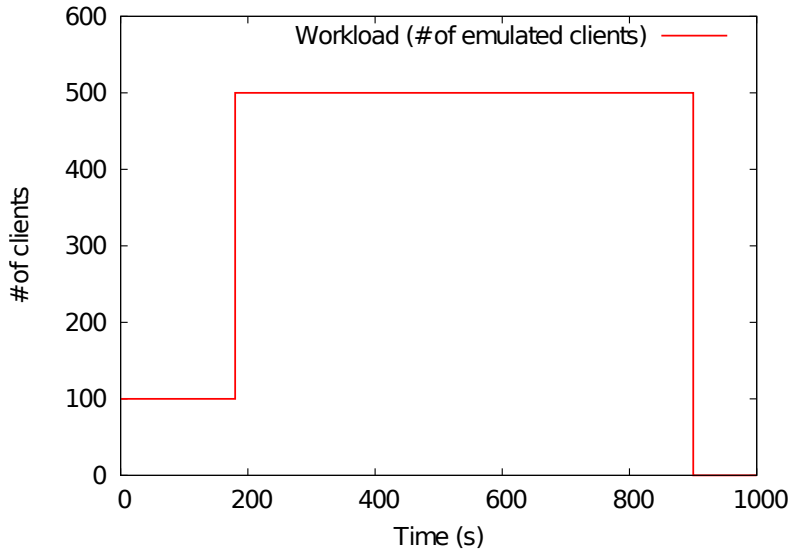


FIG. 10.10 – Variation brutale de la charge appliquée au service Internet dynamique administré.

Surcoût sur les performances du système administré

Afin de mesurer le surcoût induit par la mise en œuvre de l'auto-optimisation et par l'utilisation de la plate-forme d'administration autonome Jade, nous avons comparé l'exécution du système sur la plate-forme Jade intégrant la boucle d'auto-optimisation avec l'exécution d'un système équivalent non optimisé et en l'absence de la plate-forme Jade. Durant cette expérimentation, le système est soumis à une charge de travail modérée qui n'entraîne aucune opération d'approvisionnement dynamique.

	Débit (req./s)	Temps de réponse (ms)	Consommation CPU (%)	Consommation mémoire (%)
Avec plate-forme autonome	12	89	12.74	20.1
Sans plate-forme autonome	12	87	12.42	17.5

TAB. 10.1 – Surcoût lié à la plate-forme autonome sur le système administré.

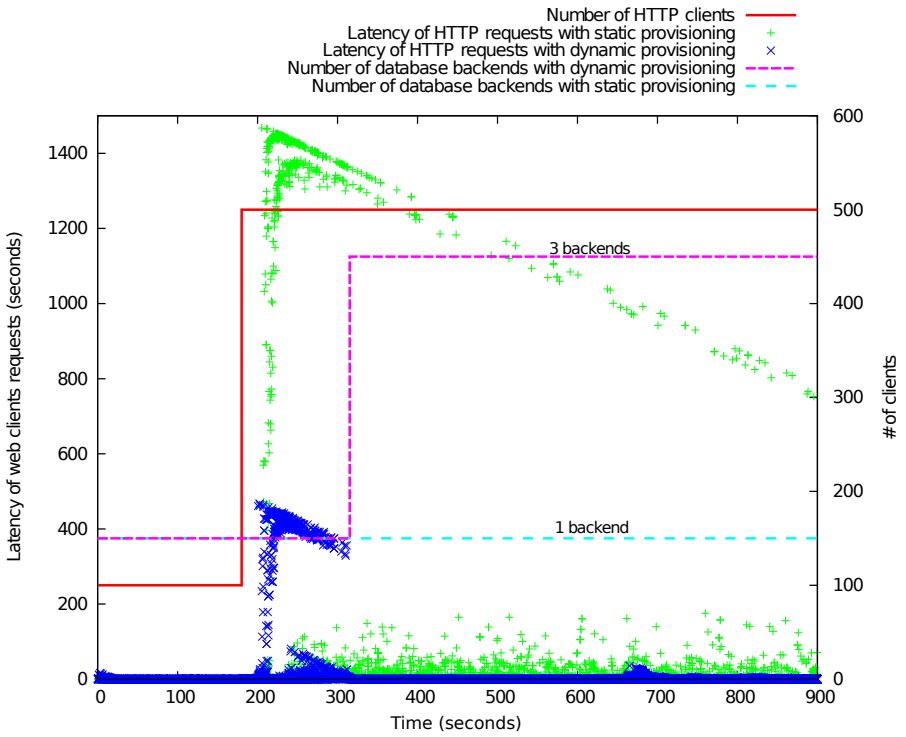


FIG. 10.11 – Temps de réponse des requêtes utilisateur en réaction du service Internet Rubis à un pic de charge intervenant à t=180s.

Les résultats présentés dans la table 10.1.2 ne montrent pas de surcoût significatif en termes de temps de réponse ou de débit de requêtes traitées par le système administré. En revanche, nous pouvons toutefois noter un léger surcoût sur la consommation mémoire du système (20.1% en présence de la plate-forme, contre 17.5% sans). Cette surconsommation mémoire doit être reliée avec la mise en œuvre de la représentation du système administré sous forme de composants logiciels administrables par la plate-forme d'administration autonome Jade. Enfin, nous n'observons aucun surcoût en terme de consommation CPU, ce qui s'explique par l'absence d'interception des communications entre les systèmes administrés.

Les précédentes évaluations montrent l'intérêt des politiques d'optimisation améliorant au mieux les performances des systèmes administrés en s'appuyant sur des heuristiques. Nous évaluons dans la suite des politiques dont l'objectif est cette fois de garantir un niveau de performance sur le système administré.

10.2 Modélisation pour garantie d'optimalité

10.2.1 Environnement de simulation

Nous présentons ici une évaluation destinée à valider les différents modèles établis dans la section 7.2.2 (page 89). Cette évaluation repose sur un ensemble de simulations illustrant les comportements des modèles. Nous cadrons ces simulations dans la perspective d'appliquer ces modèles à notre service de surveillance participant à la mise en œuvre d'un système administré auto-optimisé et dont le nombre de ressources évolue dynamiquement durant l'exécution. Nous envisageons ici deux scénarios : l'un garantissant l'empreinte du service sur les ressources du système, l'autre garantissant certains niveaux de performances.

10.2.2 Garantie de bande passante maximale.

Dans un premier scénario, nous souhaitons que l'empreinte du système de surveillance soit totalement caractérisée et maîtrisée. Nous voulons de plus obtenir dans ces conditions les meilleures performances possibles. Nous voulons en quelque sorte pouvoir considérer que l'infrastructure de surveillance fait partie de l'environnement du système et considérer que son empreinte sur le système est parfaitement connue et maîtrisée.

La figure 10.12 présente, sous forme d'un planificateur de capacité du système de surveillance, le modèle présenté en section 7.2.2 (page 89) relatif à la garantie de la consommation de la bande passante réseau. Ce planificateur prend comme paramètres la limite $B_{\max}$ de consommation de la bande passante réseau d'une machine par le système de surveillance, le nombre N de machines composant le système à surveiller et enfin la fréquence τ de rafraîchissement des informations de surveillance. Les résultats du planificateur sont la configuration architecturale (forme de l'arbre hiérarchique) optimale du système de surveillance (w, d), ainsi que la latence optimale L^* obtenue dans cette configuration.

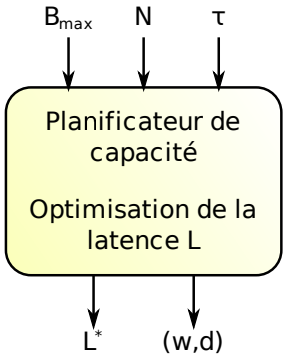


FIG. 10.12 – Planificateur de capacité avec garantie de bande passante consommée et optimisation de la latence de la surveillance réseau.

La figure 10.13 présente les résultats issus de l'évaluation du modèle garantissant la consommation de la bande passante réseau. Dans cette évaluation, nous limitons la consommation de la bande passante réseau à 10Ko/s sur chaque machine du système, et nous fixons arbitrairement la fréquence de rafraîchissement des informations à 1.7Hz. Nous faisons varier le nombre de machines composant le système entre 1 et 10000, ce qui correspond à la taille d'une infrastructure telle que Grid'5000 [57].
La figure comporte 4 graphes en fonction du nombre de machines N composant le système à surveiller : en bas, la configuration architecturale du système de surveillance (à gauche, l'arité de l'arbre hiérarchique, et à droite, sa profondeur) ; en haut, les performances du système de surveillance (à gauche, la bande passante réseau consommée sur chaque machine, et à droite, le temps de latence de la surveillance ainsi réalisée). Notons que l'arité de l'arbre est directement reliée à la consommation de bande passante (visible sur les deux graphes de gauche) ; et que la profondeur de l'arbre est directement reliée à la latence de la surveillance (visible sur les deux graphes de droite). Notons, enfin, que les courbes présentées ont des échelles logarithmiques.
La figure présente la configuration calculée par notre planificateur de capacité (courbe noire pleine), ainsi que 4 configurations fournies à titre de comparaison : deux configurations où l'arité de l'arbre est fixée manuellement, et deux configurations où la profondeur de l'arbre est fixée manuellement. Deux des configurations correspondent à des cas extrêmes (courbes en traits pointillés) : lorsque la profondeur de l'arbre est forcée à 1 (Depth=1), il s'agit d'une configuration à plat où toutes les machines du système sont directement connectées à une machine centrale ; lorsque l'arité de l'arbre est forcée à 1 (Arity=1), l'arbre est en réalité une liste, ce qui permet de réaliser la consommation de bande passante réseau minimale pour chacune des machines du système. Les deux dernières configurations correspondent à des cas intermédiaires (courbes de couleur en trait plein) : un arbre à deux niveaux (Depth=2) et un arbre binaire (Arity=2).

Étant donnée la fréquence de rafraîchissement des informations que nous avons fixée, la contrainte de bande passante maximum impose l'arité de l'arbre de propagation des informations de surveillance. La profondeur de l'arbre de propagation croît à mesure que le nombre de machines du système augmente. Enfin, la latence d'accès aux informations de surveillance augmente en fonction de la profondeur de l'arbre de propagation.

10.2.3 Garantie de latence maximale.

Dans un second scénario, nous voulons garantir la latence d'accès aux informations de surveillance, dans l'objectif de mieux calibrer les boucles d'auto-optimisation (voir section 7.1.2, page 84). Dans ce contexte, nous minimisons l'impact de l'infrastructure de surveillance sur les ressources du système administré.

La figure 10.14 présente, sous forme d'un planificateur de capacité du système de surveillance, le modèle présenté en section 7.2.2 (page 89) relatif à la garantie de latence de la disponibilité des informations de surveillance. Ce planificateur prend comme paramètres la limite $L_{\max}$ de latence de disponibilité des informations de surveillance, le

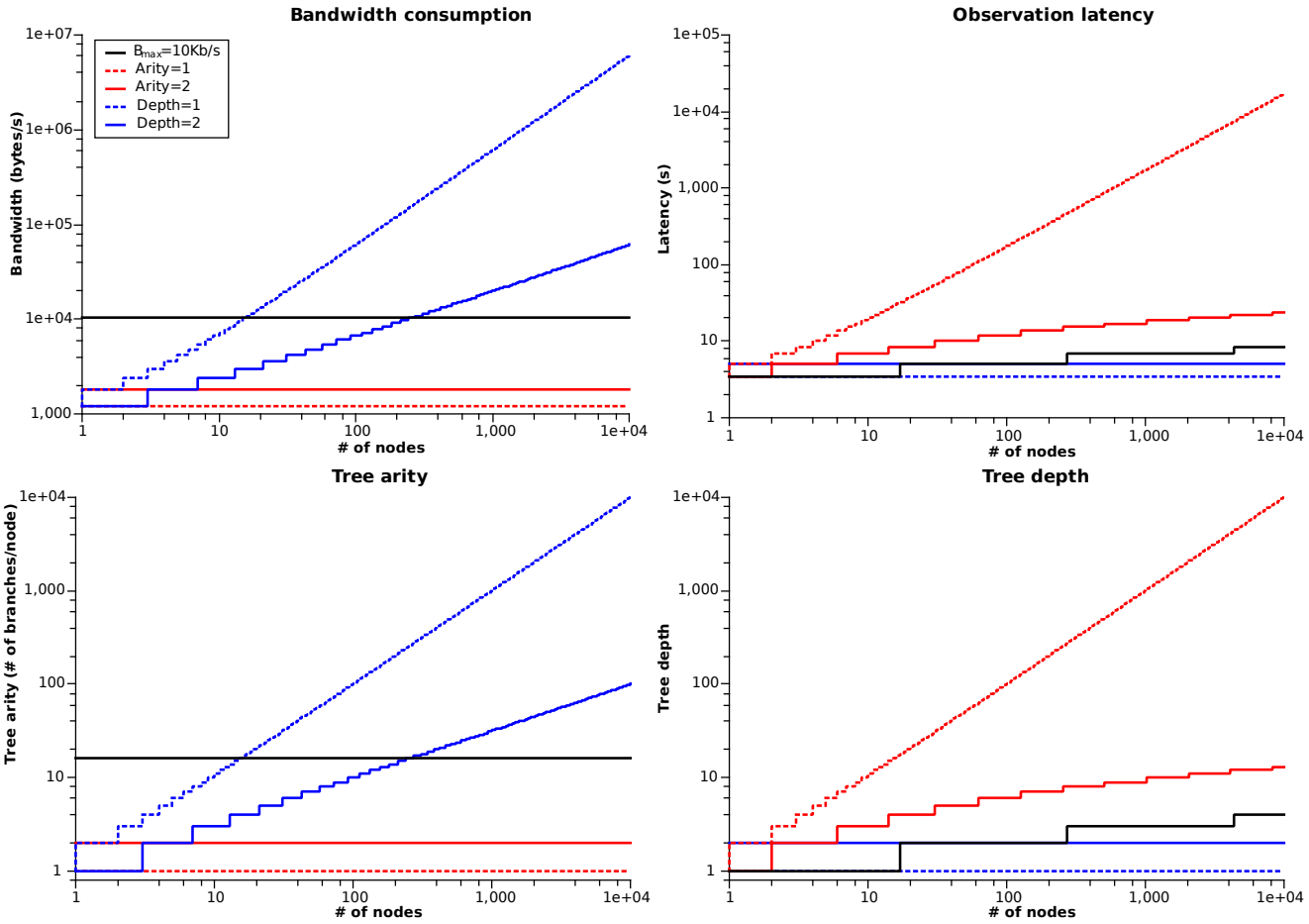


FIG. 10.13 – Garantie de bande passante maximale consommée.

nombre N de machines composant le système à surveiller et enfin la fréquence τ de rafraîchissement des informations de surveillance. Les résultats du planificateur sont la configuration architecturale (forme de l'arbre hiérarchique) optimale du système de surveillance (w, d), ainsi que la bande passante réseau maximale B^* consommée sur chaque machine dans cette configuration.

La figure 10.15 présente, sous la même forme que précédemment, les résultats obtenus par l'évaluation du modèle relatif à la garantie de la latence d'accès aux informations. Dans cette évaluation, nous limitons la latence d'accès aux informations de surveillance à 10s. Comme précédemment, nous fixons arbitrairement le taux de rafraîssissement des informations collectées à 1.7Hz, et nous faisons varier le nombre de machines composant le système entre 1 et 10000.

Étant donnée la fréquence de rafraîssissement des informations que nous avons fixée, la contrainte de latence d'accès aux informations de surveillance impose la profondeur maximum de l'arbre de propagation. L'arité de l'arbre de propagation croît

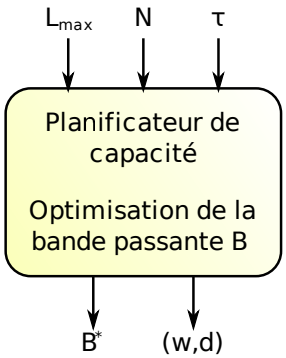


FIG. 10.14 – Planificateur de capacité avec garantie de latence de la surveillance réseau et optimisation de la bande passante consommée.

alors à mesure que le nombre de machines du système augmente, ainsi que la bande passante réseau consommée, qui est directement reliée cette arité.

10.2.4 Travaux en cours et à venir.

Dans un scénario réel d'utilisation de ces modèles, nous appliquerions ces modèles à la mise en œuvre d'un service de surveillance de composants redimensionnables auto-optimisés. Ces composants s'étendent sur des ensembles de machines dont la taille varie dynamiquement au cours de l'exécution du système, et qui nécessitent une surveillance de chacun des machines qui les composent afin de mettre en œuvre l'auto-optimisation. La garantie relative à la consommation des ressources, la bande passante réseau dans notre évaluation, apporte la maîtrise de l'intrusivité du système de surveillance sur le système administré. La garantie relative à la latence d'accès aux informations de surveillance permet de mieux caractériser les latences des reconfigurations dynamiques, et permet donc d'envisager une meilleure calibration des boucles autonomes.

Il reste néanmoins à étudier la mise en œuvre des transitions entre deux configurations optimales différentes successives, la plus immédiate et primitive des techniques consistant à simplement redéployer le service de surveillance dans sa globalité. Le chapitre qui suit évalue justement notre proposition relative à la mise en œuvre de telles actions de reconfiguration dynamique sur le système administré.

10.3 Synthèse

Nous avons présenté dans ce chapitre plusieurs résultats d'expérimentation concernant les différentes politiques d'auto-optimisation que nous avons mises en œuvre. Les évaluations réalisées démontrent la faisabilité et l'intérêt de l'auto-optimisation par approvisionnement dynamique des systèmes distribués. L'algorithme d'auto-optimisation que nous implantons est simple et réutilisable dans différents contextes applicatifs. Il

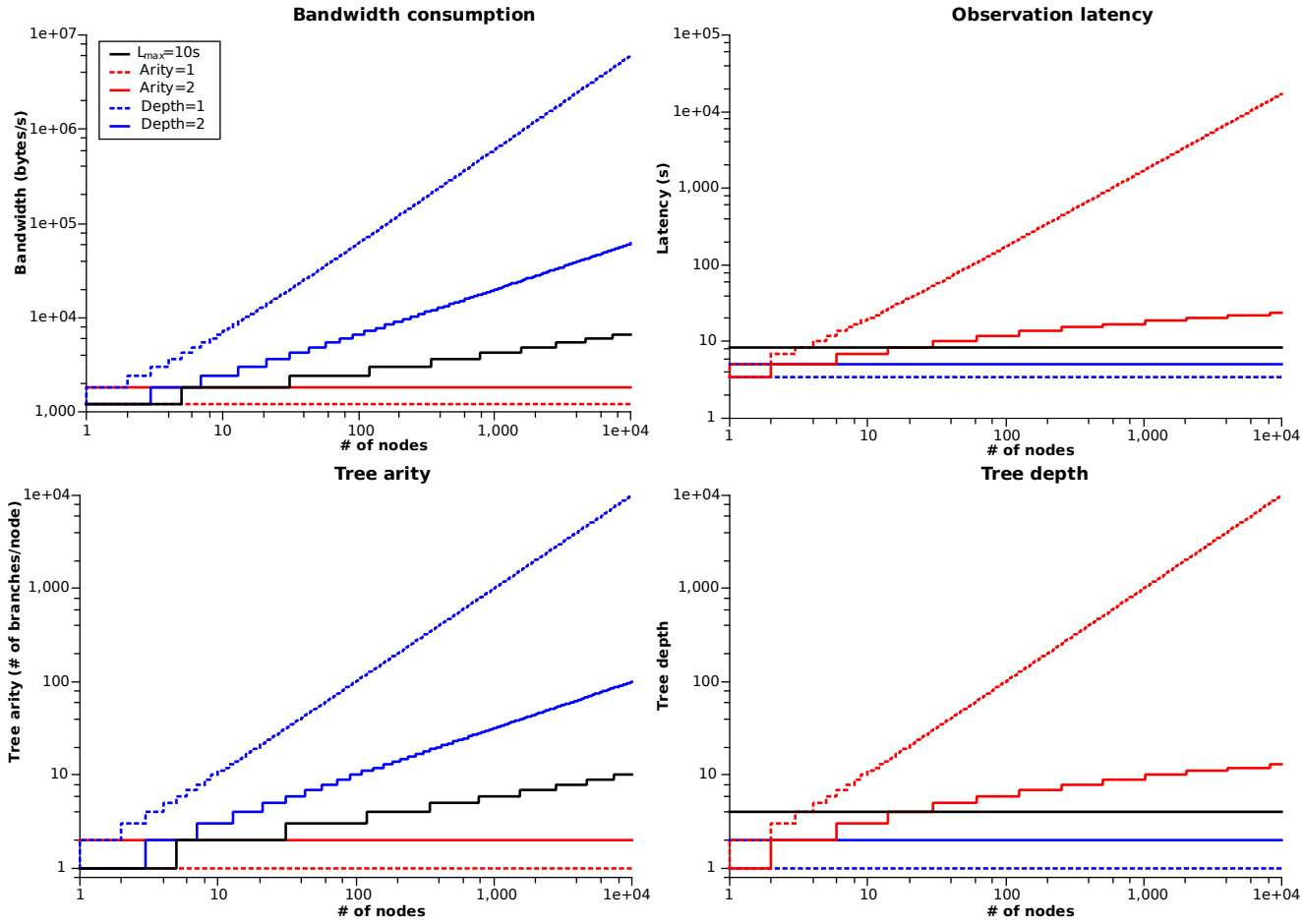


FIG. 10.15 – Garantie de latence maximale d'accessibilité aux informations de surveillance.

peut en outre être adapté de sorte de traiter aussi bien des variations graduelles de charge que des pics de charge.

Nous présentons également une évaluation par simulation de l'approche par modélisation du système administré. Cette évaluation valide le modèle que nous avons réalisé et la capacité à nous en servir pour générer des configurations architecturales optimales du système respectant des garanties de performances.

Le chapitre suivant présente une évaluation de notre contribution relative aux actions et aux observations sur les systèmes administrés.

Chapitre 11

Évaluation de FructOz et LactOz pour la construction de systèmes distribués fondés sur des architectures dynamiques

Ce chapitre s'intéresse à l'évaluation de notre contribution pour la mise en œuvre des actions et des observations sur les systèmes distribués fondés sur des architectures dynamiques. Nous présentons dans ce chapitre une évaluation des performances des différentes stratégies de déploiement distribué étudiées dans le cadre de l'application du canevas FructOz. Nous présentons ensuite une évaluation qualitative du canevas FructOz qui prend la forme d'une étude comparative avec la plate-forme de déploiement et de gestion de cycle de vie SmartFrog. Enfin, nous montrons comment combiner le canevas FructOz et la bibliothèque LactOz pour décrire et mettre en œuvre des systèmes distribués réels fondés sur des architectures dynamiques. Nous revenons à ce titre sur les services à messages, les services Internet et les services de surveillance des systèmes distribués.

11.1 Évaluation de performance des stratégies de déploiement distribué

Nous présentons ici une évaluation des performances des différentes stratégies de déploiement distribué exprimées dans le cadre du canevas FructOz et présentées à la section 8.4.5 (page 109).

11.1.1 Environnement d'évaluation

Environnement matériel. Cette évaluation a été conduite sur une grappe de 16 machines compatibles x86 issues de l'infrastructure Grid'5000 [57], équipées de bi-processeurs

Opteron 252 fonctionnant à 2.6GHz, équipées de 4Go de mémoire vive et interconnectées par un réseau Ethernet 1Gb/s.

Environnement logiciel. La grappe de machines fonctionne sur un environnement Linux 2.6 exécutant la plate-forme Mozart/Oz [30] en version 1.3.2.

Application. L'évaluation consiste à mesurer la vitesse de déploiement du composant composite CompositePackage présenté à la section 8.4.5 (page 109) en fonction de la taille de ce composant, qui correspond au nombre N de composants à instancier en tant que sous-composants du composite. Nous évaluons et comparons ici les performances des déploiements distribués de ce composant au moyen des stratégies centralisée séquentielle, centralisée parallèle et enfin distribuée hiérarchique.

11.1.2 Résultats de performance des stratégies de déploiement

La figure 11.1 synthétise les résultats issus de l'évaluation que nous avons menée. La figure 11.1-(a) présente les stratégies centralisée séquentielle et centralisée parallèle. Nous observons une amélioration intéressante de la vitesse de déploiement de l'ordre de 25% dans le cas de la stratégie parallèle par rapport à la stratégie séquentielle. Cette amélioration est liée à la parallélisation des temps d'attente causés par les latences réseau lors des exécutions à distance. La figure 11.1-(b) présente les performances obtenues grâce à la stratégie de déploiement distribuée hiérarchique, dans le cas d'arbres de déploiement binaire, ternaire et enfin quaternaire. Nous observons une amélioration considérable de la vitesse de déploiement dans le cas des stratégies distribuées hiérarchiques, avec un temps total de déploiement environ 5 fois plus court que dans le cas des déploiements parallèles précédents. Par exemple, pour déployer 500 composants, alors qu'il faut environ 60s avec un déploiement séquentiel, et environ 40s avec un déploiement parallèle, il ne faut qu'un peu plus de 7s avec un déploiement distribué en arbre quaternaire.

Ces résultats justifient la pertinence des capacités d'expression et de mise en œuvre de différentes stratégies de déploiement et de synchronisation qu'autorise le canevas FructOz. À titre de comparaison, nous confrontons dans la section suivante notre canevas FructOz à la plate-forme de déploiement distribué SmartFrog représentative de l'état de l'art dans ce domaine.

11.2 Comparaison avec SmartFrog

Nous nous comparons ici à la plate-forme de déploiement distribué SmartFrog, qui repose sur un modèle de composant et intègre la bibliothèque de composants SmartFlow spécialisée pour le contrôle des flots d'exécution (*workflow*). Nous effectuons une comparaison qualitative de notre prototype FructOz avec la plate-forme SmartFrog. Cette comparaison a pour objectif de mettre en avant les avantages et les défauts de

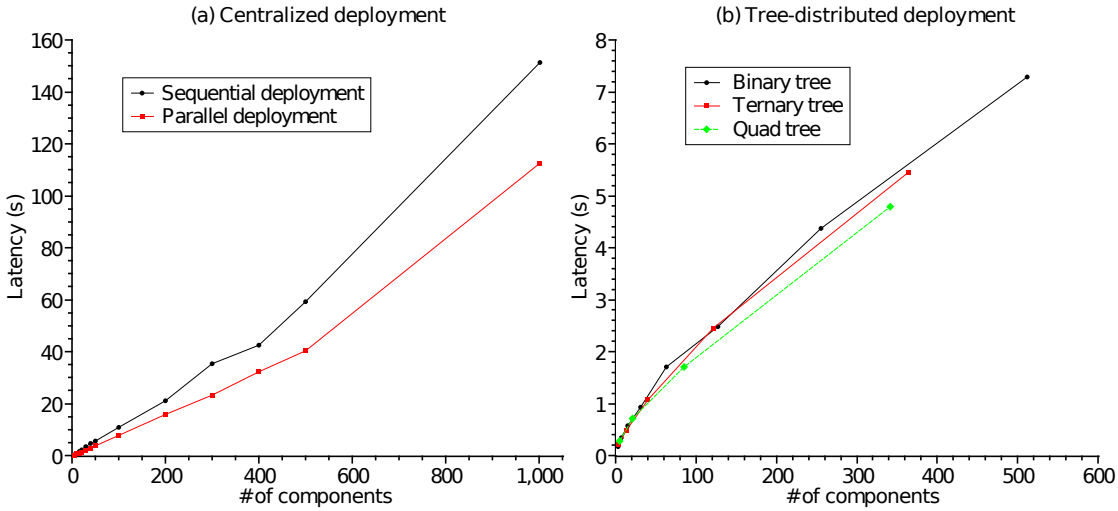


FIG. 11.1 – Comparaison des performances des différentes stratégies de déploiement distribué.

nos plates-formes respectives. À cette fin, nous faisons intervenir certains des scénarios de déploiement présentés précédemment permettant de concrétiser les différences mentionnées.

Dans un premier temps, nous présentons rapidement les caractéristiques de la plate-forme SmartFrog. Nous en donnons ensuite quelques exemples d'utilisation pour réaliser des déploiements distribués et pour coordonner des actions. Enfin, nous illustrons sur quelques scénarios les limitations de SmartFrog.

11.2.1 Critères de comparaison

Nous concentrons notre étude sur les deux critères de comparaison suivants. (1) D'une part, la capacité de paramétrage du langage de description d'architecture (ADL). Cette faculté est essentielle pour abstraire et réutiliser des schémas d'architecture génériques dans différents contextes. (2) Et, d'autre part, le contrôle des processus de déploiement distribués. Cela comporte le contrôle de la distribution de l'architecture et aussi du processus de déploiement, ainsi que la coordination et la synchronisation (workflow) au sein du processus de déploiement distribué.

11.2.2 Présentation de SmartFrog

Nous présentons ici la plate-forme SmartFrog [9, 56] plus en détail, en focalisant sur les points permettant d'établir une comparaison avec notre canevas FructOz.

SmartFrog est une plate-forme spécialisée pour les déploiements distribués et la gestion du cycle de vie des systèmes. SmartFrog s'articule autour d'un modèle de composant permettant de concevoir des systèmes distribués, d'un langage de description

d'architecture (ADL), et enfin d'une bibliothèque de composants, SmartFlow, spécialisée pour la coordination d'actions distribuées (workflow). Les systèmes conçus à l'aide de la plate-forme SmartFrog associent deux langages : le langage de programmation Java permettant de décrire l'implantation des composants SmartFrog, et l'ADL SmartFrog permettant de décrire l'architecture du système en assemblant des composants SmartFrog.

Modèle de composant. Les composants SmartFrog sont étroitement liés aux objets Java. Un composant SmartFrog est implanté par une classe Java et exporte généralement une interface Java accessible à distance par Java RMI. Un composant SmartFrog contient un ensemble d'attributs. La valeur associée à un attribut peut-être une donnée immédiate (entier, chaîne de caractère, etc), un composant ou encore une référence. Un attribut dont la valeur est un composant correspond à un sous-composant, dont le cycle de vie est alors lié à celui de son composant parent. En particulier, le déploiement, le démarrage ou encore l'arrêt d'un composant implique celui de ses sous-composants. Les attributs références permettent, entre autres, d'établir une référence vers n'importe quel composant de l'architecture, réalisant ainsi les liaisons entre composants. Enfin, certains attributs ont des rôles spéciaux, comme par exemple l'attribut sfClass qui permet de spécifier la classe Java d'implantation du composant, ou encore l'attribut sfProcessHost qui permet de définir la machine sur laquelle le composant doit être déployé.

Langage de description. L'ADL SmartFrog est un langage purement descriptif. Il repose sur le prototypage de composants et sur leur spécialisation (similaire à l'héritage). Un descripteur de composant liste essentiellement l'ensemble de ses attributs. Ainsi, l'ADL SmartFrog permet de décrire textuellement l'arborescence des composants et des sous-composants formant l'architecture. Les références permettent d'établir des liaisons vers n'importe quel composant de l'architecture. La redéfinition de l'attribut sfProcessHost permet de contrôler la distribution de l'architecture sur l'ensemble des machines de l'environnement distribué.

L'ADL SmartFrog ne contient pas de structure de contrôle. Il est cependant possible d'obtenir l'équivalent de certaines structures grâce aux fonctions SmartFrog, qui permettent d'établir la valeur d'un attribut comme le résultat de l'évaluation d'une fonction. Il est ainsi possible de définir et d'invoquer une fonction en la modélisant sous la forme d'un composant qui sera remplacé lors de l'analyse syntaxique ou lors de l'instanciation par le résultat de l'invocation de la fonction.

L'analyse syntaxique d'une description ADL SmartFrog produit une représentation en mémoire de la description d'architecture, sous forme d'un graphe d'objets Java de classe ComponentDescription. Cette structure est utilisée par le moteur de déploiement SmartFrog pour piloter l'instanciation effective de l'architecture. C'est lors de cette phase du déploiement que les références sont résolues pour établir les liaisons vers les composants instanciés.

Coordination et workflow. SmartFrog inclut des mécanismes permettant de contrôler et de coordonner l'exécution d'un ensemble actions (workflow). Ces mécanismes reposent essentiellement sur un système évènementiel distribué. SmartFrog intègre la bibliothèque SmartFlow de composants spécialisés dans l'expression de workflows qui s'appuient sur ce système évènementiel. Les composants SmartFlow implantent divers schémas de coordination usuels (séquence, parallélisme, boucle, temporisation, délai de garde, etc) qui opèrent sur les actions représentées par leurs sous-composants. Il est ainsi très simple de décrire des workflows complexes en composant plusieurs composants SmartFlow.

Les actions dont on souhaite contrôler et synchroniser l'exécution doivent être représentées sous forme de composants SmartFrog pour être intégrées aux composants SmartFlow. Les composants SmartFlow reposent alors sur le cycle de vie et principalement sur la terminaison des sous-composants représentant les actions pour mettre en œuvre les synchronisations et les coordinations.

Enfin, il est possible d'exprimer des synchronisations plus élaborées en exploitant directement le système évènementiel. La bibliothèque SmartFlow intègre plusieurs composants permettant d'émettre des évènements (composant EventSend) et de réaliser une synchronisation sur la réception d'évènements (par exemple, au moyen des composants OnEvent ou EventCounter).

11.2.3 Déploiements distribués et coordinations avec SmartFrog et FructOz

Nous présentons maintenant quelques exemples représentatifs des fonctionnalités offertes par la plate-forme de déploiement et de gestion de cycle de vie SmartFrog. À titre de comparaison, nous fournissons une traduction sur le canevas FructOz des descriptions obtenues.

Coordination d'actions. Nous montrons dans cet exemple comment utiliser la bibliothèque SmartFlow pour exprimer et coordonner des actions distribuées. L'exemple consiste à se synchroniser sur l'exécution en parallèle de deux actions sur deux machines distinctes. La première action réalise un téléchargement et est soumise à un délai de garde de 60 secondes. La seconde action invoque un script shell et éventuellement un second script shell de compensation dans le cas où le premier script aurait échoué.

SmartFrog

```
sfConfig extends Parallel {
   action1 extends Sequence {
      sfProcessHost 'hostname1';
      timeout extends Timeout {
         time 60000;
         download extends Downloader {
            url 'http://hostname/file';
            toFile '/tmp/file';
         }
      }
   }
   action2 extends Sequence {
      sfProcessHost 'hostname2';
      try extends Try {
         normal extends BashShellScript {
            cmd 'my-shell-script ';
         }
         abnormal extends BashShellScript {
            cmd 'compensation-script';
         }
      }
   }
}
```

FructOz

```
functor Procedure
export Membrane
define
   proc {Action1}
      functor Action
      export Membrane
      define
         proc {DownloadAction} % download a file
            {Download "http://hostname/file" "/tmp/file"}
         end
         {Timeout DownloadAction 60000}
      end
   in
      {RemoteDeploy Host 1Action}
   end
   proc {Action2}
      functor Action
      import OS
      export Membrane
      define
         proc {Action} % execute the given shell−script
            if ({OS.system "my−shell−script"} \= 0) then
               raise shellError end
            end
         end
         proc {OnError} % execute a compoensation shell−script
            {OS.system "compensation−script"}
         end
         {HandleError Action OnError}
      end
   in
      {RemoteDeploy Host2 Action}
   end
   {Barrier [Action1 Action2]}
end
```

La version FructOz s'appuie sur des schémas de synchronisation détaillés en annexe (page 199).

Déploiement séquentiel. Nous montrons dans cet exemple comment réaliser un déploiement séquentiel d'un ensemble de composants. Il s'agit ici de déployer un composant composite contenant 3 sous-composants identiques distribués sur des machines distinctes. Il s'agit d'une instance du composant composite {CompositePackage Cluster N DeployProc} présenté à la section 8.4.5 (page 109) pour laquelle nous connaissons à l'avance le nombre de sous-composants et la distribution des sous-composants sur les machines formant l'environnement distribué. L'algorithme de déploiement utilisé pour déployer les sous-composants est décrit dans le code d'implantation du cycle de vie du composant Compound (méthode sfDeploy, sfStart et sfTerminate).

SmartFrog

```
sfConfig extends Compound {
   subComp1 extends SubComponentType {
      sfProcessHost 'hostname1'; ...
   }
   subComp2 extends SubComponentType {
      sfProcessHost 'hostname2'; ...
   }
   subComp3 extends SubComponentType {
      sfProcessHost 'hostname3'; ...
   }
}
```

FructOz

```
functor Pakage
export Membrane
define
   C = {CNew nil}
   SC1 = {RemoteDeploy Host1 SubComponentPkg}
   {CAddSubComponent C SC1}
   SC2 = {RemoteDeploy Host2 SubComponentPkg}
   {CAddSubComponent C SC2}
   SC3 = {RemoteDeploy Host3 SubComponentPkg}
   {CAddSubComponent C SC3}
   Membrane = C
end
```

Déploiement parallèle. Nous montrons ici comment paralléliser le déploiement d'un ensemble de composants. Nous reprenons et ajustons l'exemple précédent afin de paralléliser le déploiement des 3 sous-composants. La plate-forme SmartFrog intègre dans la bibliothèque SmartFlow le composant Parallel dont l'objet est précisément de paralléliser le déploiement de l'ensemble de ses sous-composants.

SmartFrog

```
sfConfig extends Parallel {
   subComp1 extends SubComponentType {
      sfProcessHost 'hostname1'; ...
   }
   subComp2 extends SubComponentType {
      sfProcessHost 'hostname2'; ...
   }
   subComp3 extends SubComponentType {
      sfProcessHost 'hostname3'; ...
   }
}
```

FructOz

```
functor Pakage
export Membrane
define
   C = {CNew nil}
   thread
      SC1 = {RemoteDeploy Host1 SubComponentPkg} in
      {CAddSubComponent C SC1}
   end
   thread
      SC2 = {RemoteDeploy Host2 SubComponentPkg} in
      {CAddSubComponent C SC2}
   end
   thread
      SC3 = {RemoteDeploy Host3 SubComponentPkg} in
      {CAddSubComponent C SC3}
   end
   Membrane = C
end
```

Déploiement hiérarchique. Enfin, nous montrons comment procéder au déploiement hiérarchique d'un ensemble de composants. Dans une infrastructure distribuée SmartFrog, chaque composant composite est responsable du déploiement de ses sous-composants. Ainsi, il est possible d'obtenir un déploiement distribué hiérarchique en composant et en distribuant selon une structure arborescente les composants. Notons que l'architecture du système à composants produite par cette description impose une relation de composition hiérarchique entre les composants SmartFrog.

SmartFrog

```
root extends Parallel {
   sfProcessHost "hostname0";

   % Deploy a component locally
   subComp0 extends SubComponentType {
      ...
   }

   % First deployment branch on hostname1,3,4
   left extends Parallel {
      sfProcessHost "hostname1";
      subComp1 extends SubComponentType {
         ...
      }
      left extends Parallel {
         sfProcessHost "hostname3";
         ...
      }
      right extends Parallel {
         sfProcessHost "hostname4";
         ...
      }
   }

   % Second deployment branch on hostname2,5,6
   right extends Parallel {
      sfProcessHost "hostname2";
      ... % Similar to left with different hostnames
   }
}
```

FructOz

```
functor Pakage
export Membrane
define
   C = {CNew nil}

   % Deploy a component locally
   thread
      Local = {Deploy SubComponentPkg}
   in
      {CAddSubComponent C Local}
   end

   % Spread a deployment branch on Host1,3,4
   functor LeftPkg
   export Membrane
   define
      thread
         Membrane = {Deploy SubComponentPkg}
         {CAddSubComponent C Membrane}
      end

      functor LeftPkg ... end
      Left = thread {RemoteDeploy Host3 LeftPkg} end

      functor RightPkg ... end
      Right = thread {RemoteDeploy Host4 RightPkg} end
   end
   Left = thread {RemoteDeploy Host3 LeftPkg} end

   % Spread another deployment branch on Host2,5,6
   functor RightPkg
      ... % similar to LeftPkg except for the HostX
   end
   Right = thread {RemoteDeploy Host2 RightPkg} end

   Membrane = C
end
```

11.2.4 Limitations de SmartFrog

Les exemples précédents montrent la simplicité d'utilisation ainsi que la concision des descriptions SmartFrog pour mettre en œuvre des systèmes distribués à composants ainsi que des actions distribuées coordonnées. Toutefois, l'ADL SmartFrog n'est pas un langage de description d'ordre supérieur. Cette caractéristique limite le degré de paramétrage des architectures qu'il est possible de décrire. En pratique, l'ADL SmartFrog permet essentiellement la description d'une architecture immédiate, c'est-à-dire la description d'une architecture entièrement connue et explicitée.

Par exemple, dans le cadre de la mise en œuvre d'un service dupliqué en grappe tel qu'un service à messages ou un service Internet en grappe, nous décrivons l'architecture d'un service dupliqué en grappe au moyen d'un composant composite dont les sous-composants représentent les multiples instances du service dupliqué. Le nombre de ces instances est amené à prendre diverses valeurs, si bien qu'il est intéressant de paramétrer cette architecture par le nombre de ces sous-composants, ainsi que nous

l'avons réalisé dans FructOz avec le composant {CompositePackage Cluster N DeployProc} présenté à la section 8.4.5 (page 109).

Nous montrons dans les exemples qui suivent comment mettre en œuvre le paramétrage du nombre de sous-composants à l'aide de l'ADL SmartFrog. Nous présentons plusieurs variations selon la stratégie de déploiement (séquentielle, parallèle ou hiérarchique) des sous-composants que nous cherchons à mettre en œuvre.

Déploiement séquentiel paramétré. Lorsque le nombre de sous-composants N n'est pas connu a priori ou doit pouvoir être facilement changé, ce composite s'exprime au moyen de l'opérateur de répétition SmartFlow Repeat. En effet, le langage de description d'architecture SmartFrog n'inclut aucune structure syntaxique permettant de réaliser cette répétition. Cela rend nécessaire l'utilisation du composant Repeat pour produire cette répétition.

SmartFrog	FructOz

n 3;
sfConfig extends Repeat {
repeat n;
action extends SubComponentType {
sfProcessHost extends evalHostName {
sfFunctionLazy true;
...
}
...
}
}
	functor Pakage
	export Membrane
	define
	C = {CNew nil}
	for Host **in** [Host1 Host2 Host3] do
	SC = {RemoteDeploy Host SubComponentPkg}
	in
	{CAddSubComponent C SC}
	end
	Membrane = C
	end

La machine sur laquelle déployer chaque sous-composant est déterminée par l'évaluation la fonction-composant evalHostName. Afin que cette évaluation n'ait lieu qu'au moment de l'exécution de l'opérateur Repeat et non au moment de l'analyse syntaxique du descripteur d'architecture, nous devons forcer l'évaluation paresseuse de cette fonction par l'attribut sfFunctionLazy.

Il s'agit ici d'une utilisation détournée des composants SmartFlow. En effet, l'exécution de l'instruction Repeat se synchronise sur la terminaison du sous-composant action, afin de passer à l'itération suivante. Comme on souhaite activer plusieurs instances du même sous-composant, nous sommes alors contraints de déconnecter l'exécution du sous-composant de celle du composant Repeat, au moyen d'un second composant SmartFlow DetachingCompound, aboutissant finalement à la description suivante.

```
n 3;
sfConfig extends Repeat {
  repeat n;
  action extends DetachingCompound {
    detachUpwards true;
    subComp extends SubComponentType {
      sfProcessHost extends evalHostName { sfFunctionLazy true; ... }
      ...
    }
  }
}
```

De cette manière, l'exécution du sous-composant peut se poursuivre après l'exécution de l'opérateur Repeat. En revanche, cette manipulation par l'opérateur DetachingCompound présente l'inconvénient de transformer chaque sous-composant en composant racine (*top-level*) indépendant. Cela brise la relation de composition avec les sous-composants générés.

Déploiement parallèle paramétré. Notons en revanche que l'opérateur Parallel se synchronise sur la terminaison de l'ensemble des sous-composants. Pour réaliser une synchronisation sur l'état "déployé" des sous-composants, il est alors nécessaire de mettre en œuvre un mécanisme de synchronisation ad hoc par barrière. Cela prend la forme d'un composant SmartFrog EventCounter qui reçoit des évènements en provenance de chacun des sous-composants pour signaler leur état. Ces évènements sont émis au moyen du composant EventSend, une fois le composant déployé.

SmartFrog

```
/* Sub-component prototype */
SubComponentType extends Sequence {
   /* Deploy the component content */
   ...
   /* Send the barrier synchronization notification */
   notify extends EventSend {
      sendTo:a LAZY ROOT:...:barrier:barrier;
      /* identification of the barrier EventCounter */
      event "ready";
   }
}

/* Main SmartFrog configuration */
sfConfig extends Parallel {
   ...
   barrier extends Sequence {
      /* before barrier synchronization */
      barrier extends EventCounter {
         count 3;
      } /* EventCounter */
      /* after barrier synchronization */
   }
   ...
}
```

FructOz

```
functor Pakage
export Membrane
define
   C = {CNew nil}
   fun {DeployAction Host}
      proc {$}
         SC = {RemoteDeploy Host SubComponentPkg}
      in
         {CAddSubComponent C SC}
      end
   end
   {Barrier
      [{DeployAction Host1}
       {DeployAction Host2}
       {DeployAction Host3}]}
   Membrane = C
end
```

L'identification du composant EventCounter destiné à recevoir les messages de synchronisation est particulièrement complexe, étant donné le mécanisme de résolution des références adopté par SmartFrog, qui dépend fortement du contexte d'utilisation du prototype de composant SubComponentType. En comparaison, les synchronisations FructOz reposent généralement sur le mécanisme de liaison des variables, dont la description et la résolution correspondent simplement à la portée lexicale naturelle des déclarations de variables Oz. Nous exploitons ici le schéma de synchronisation par barrière déjà présenté (voir annexe, page 199).

Déploiement hiérarchique paramétré. Nous n'avons pas trouvé de moyen d'exprimer un déploiement hiérarchique paramétré sous forme d'un descripteur ADL Smart-

Frog. Une piste pour mettre en œuvre le déploiement hiérarchique paramétré consiste à développer un nouveau composant SmartFlow. Ce composant prend la forme d'un composite TreeParallel dont le code d'instanciation et de démarrage est spécialisé (à l'instar des composants SmartFlow existants : Parallel, Sequence, Repeat, etc), afin de contrôler l'instanciation et le démarrage des sous-composants selon un arbre hiérarchique. La mise en œuvre concrète de ce composant associe le prototype ADL du composant TreeParallel et le code Java d'implantation du composant. Le code d'implantation Java manipule les représentations Java des descriptions ADL des sous-composants à instancier sous forme d'objets ComponentDescription. Il s'agit de construire programmatiquement une description adaptée en fonction de paramètres interprétés par l'exécution de programmes Java. Plus précisément, le programme Java génère une représentation de description adaptée en construisant un objet ComponentDescription, par exemple par copie et modification d'un objet ComponentDescription produit par l'analyseur syntaxique de l'ADL SmartFrog. Le programme Java contrôle alors le déploiement de l'architecture en invoquant le moteur de déploiement SmartFrog sur les objets ComponentDescription ainsi générés.

11.2.5 Synthèse

La réalisation de ces divers exemples révèle plusieurs différences notables entre la plate-forme SmartFrog et notre canevas FructOz.

Avant tout, il convient de noter les différences significatives entre le modèle de composant Fractal sur lequel nous reposons dans FructOz et le modèle de composant SmartFrog. Ces différences limitent l'étendue de la comparaison que nous pouvons établir entre ces deux plates-formes. La plus importante différence concerne le lien très étroit existant entre la composition et le cycle de vie des composants imposé par SmartFrog. D'autre part, les interfaces des composants SmartFrog sont implicitement déterminées par les interfaces Java qu'implantent les classes Java associées aux composants et ne sont pas explicitées ni manipulables dans l'ADL SmartFrog. Enfin, les liaisons sont implicites limitant ainsi toute comparaison relative, par exemple, au paramétrage d'un schéma d'interconnexion.

L'ADL SmartFrog est particulièrement efficace pour réaliser des descriptions d'architecture dont nous connaissons parfaitement tous les détails. Il offre de plus certaines formes de paramétrages, mais qui sont relativement simples et limitées. En réalité, l'ADL SmartFrog n'est pas d'ordre supérieur. Cette caractéristique bride fortement les capacités de paramétrage de cet ADL, ainsi que l'ont montré certains de nos exemples. Les capacités d'abstraction et de réutilisation de schémas d'architectures (paramétrés) sont donc fortement limitées.

SmartFrog comprend la bibliothèque de composants SmartFlow spécialisés dans la coordination et la synchronisation d'actions distribuées. L'utilisation de cette bibliothèque se révèle très simple et efficace pour coordonner des ensembles d'actions représentées sous forme de composants SmartFrog. Cependant, les composants SmartFlow ne sont pas exploitables pour représenter et ainsi contrôler les déploiements de sys-

tèmes à composants SmartFrog. Cette limitation est une conséquence de la forte dépendance des composants SmartFlow envers le cycle de vie des composants représentant les actions qu'ils contrôlent. Comme le montrent nos derniers exemples, l'usage des composants SmartFlow dans le but de contrôler les déploiements de systèmes à composants SmartFrog nous contraint à tordre le modèle de composant de manière non satisfaisante. L'usage de la bibliothèque SmartFlow ne permet donc pas en l'état de réaliser des déploiements distribués d'architectures SmartFrog intégrant des synchronisations et des coordinations.

11.3 Applications réelles de FructOz et LactOz

Nous démontrons dans cette section comment combiner le canevas FructOz spécialisé dans la construction des systèmes distribués fondés sur des architectures dynamiques avec la bibliothèque LactOz de calcul et de navigation dynamique. Nous utilisons de façon conjointe le canevas FructOz et la bibliothèque LactOz pour décrire des systèmes distribués fondés sur des architectures dynamiques qui mettent en œuvre des systèmes réels. Dans cet objectif, nous reprenons les différents contextes applicatifs que nous avons considérés dans le cadre de l'implantation de politiques d'auto-optimisation : les services à messages, les services Internet et les services de surveillance des systèmes distribués.

11.3.1 Service à messages en grappe

Les services à messages en grappe ont une architecture qui correspond exactement à celle d'un composant redimensionnable (voir la section 6.2.1, page 68 et la section 7.1.3, page 85). En supposant que nous disposons d'un composant représentant un serveur JMS nommé JMSPkg, nous pouvons construire un service à messages en grappe en nous appuyant sur le composant générique redimensionnable ResizablePkg présenté en section 9.4.2 (page 125). L'architecture que nous décrivons ci-dessous est paramétrée par l'ensemble Cluster des machines à utiliser pour déployer les instances des serveurs JMS, par le package du composant JMSPkg représentant un serveur JMS et par des spécifications de qualité de service QoSSpecs associées à l'architecture (comme par exemple une liste de contraintes, dont un nombre maximal de clients que l'architecture doit pouvoir supporter, etc).

```
fun {MessageService Cluster JMSPkg QoSSpecs}
   %% Simple host allocator: we assume here that JMSPkg components are tagged with 'JMS'
   SHosts = {CGetSubComponents Cluster}
   SFreeHosts = {SFilter SHosts
      fun {$ Host} {SIsEmpty {SFilter {CGetSubComponents Host} {TaggedWith 'JMS'}}} end}

   %% GetHost randomly chooses a host from the free hosts list
   fun {GetHost}
      LHosts = {SFreeHosts toList($)}
   in
      case LHosts
```

```
        of nil then raise noMoreResources end
        [] H | T then H
        end
    end

    %% Deploy a sub−component instance on a random free host
    fun {ClusterDeploy Package}
        Host = {GetHost}
    in
        {RemoteDeploy Host Package}
        %% The Host component is automatically removed from the free host list SFreeHosts.
    end

    %% Remove an existing sub−component and release the corresponding host
    proc {ClusterUndeploy Comp}
        Host = {SGetSingleton {SFilter {CGetSuperComponents Comp} {TaggedWith 'host'}}}
    in
        %% Removing the component instance requires to remove its reference from the Host component.
        %% The component instance will then be collected on the next GC.
        %% As a result, the Host component will be considered free again in SFreeHosts.
        {CRemoveSubComponent Host Comp}
    end

    %% Capacity planning
    %% Determine the correct number of JMS servers matching the given QoS constraints
    NJMSServers = {JMSCapacityPlanning QoSSpecs}
in
    %% Resizable architecture
    {ResizablePkg JMSPkg NJMSServers ClusterDeploy ClusterUndeploy}
end
```

Le composant JMSPkg peut directement implanter un serveur JMS en Oz, ou bien encapsuler un serveur JORAM dont il contrôle la configuration et l'exécution, par exemple en invoquant des commandes shell grâce à la fonction système fourni par la plate-forme Mozart : {OS.system command}.

Cette description s'appuie sur une fonction réalisant la planification de la capacité du service à messages en grappe JMSCapacityPlanning. Cette fonction, qui n'est pas précisée ici, peut par exemple exploiter une modélisation des serveurs JMS afin d'en déterminer le nombre d'exemplaires nécessaires pour satisfaire les spécifications de qualité de service indiquées, comme par exemple un nombre de clients.

11.3.2 Service Internet en grappe

De façon similaire, l'architecture d'un service Internet en grappe correspond à un pipeline de composants redimensionnables (voir la section 6.2.2, page 70 et la section 7.1.3, page 87). En nous inspirant de l'architecture exhibant un schéma d'interconnexion dynamique entre deux composants redimensionnables que nous avons présentée en section 9.4.2 (page 127), nous construisons l'architecture d'un service Internet composé de trois étages. Comme dans l'exemple précédent, l'architecture décrite ci-dessous est paramétrée par l'ensemble Cluster des machines sur lesquelles instancier les

divers composants et par une spécification de qualité de service QoSSpecs. L'architecture est également paramétrée par les descriptions (packages) des trois types de serveurs correspondant aux trois étages du service à mettre en œuvre : WebPkg, AppPkg et DBPkg, ainsi que par deux schémas d'interconnexion intervenant respectivement entre l'étage Web et l'étage applications, et entre l'étage applications et l'étage bases de données.

```
fun {InternetService Cluster WebPkg AppPkg DBPkg WebAppBindScheme AppDBBindScheme QoSSpecs}
   %% Simple host allocator
   ... % similar to the previous example (MessageService)

   %% Determine the correct number servers matching the given QoS constraints
   NWebServers NAppServers NDBServers
   {JavaEECapacityPlanning QoSSpecs NWebServers NAppServers NDBServers}
in
   functor
   export Membrane
   define
      Comp = {CNew nil}

      %% Pipeline of resizable components
      WebSizable = {Deploy {ResizablePkg WebPkg NWebServers ClusterDeploy ClusterUndeploy}}
      {CAddSubComponent Comp WebSizable}
      AppSizable = {Deploy {ResizablePkg AppPkg NAppServers ClusterDeploy ClusterUndeploy}}
      {CAddSubComponent Comp AppSizable}
      DBSizable = {Deploy {ResizablePkg DBPkg NDBServers ClusterDeploy ClusterUndeploy}}
      {CAddSubComponent Comp DBSizable}

      %% Locate the set of processing units of each resizable component
      SWebServers = {SFilter {CGetSubComponents WebSizable} {TaggedWith 'Processing␣Unit'}}
      SAppServers = {SFilter {CGetSubComponents AppSizable} {TaggedWith 'Processing␣Unit'}}
      SDBServers = {SFilter {CGetSubComponents DBSizable} {TaggedWith 'Processing␣Unit'}}

      %% Apply the dynamic interconnection scheme
      {ApplyBindingScheme WebAppBindScheme SWebServers SAppServers}
      {ApplyBindingScheme AppDBBindScheme SAppServers SDBServers}

      Membrane = Comp
   end
end
```

De façon analogue, les composants WebPkg, AppPkg et DBPkg peuvent implanter des composants d'encapsulation Jade des systèmes patrimoniaux, par exemple d'un serveur Apache HTTPD pour WebPkg, d'un serveur Apache Tomcat pour AppPkg, etc. Ces composants pourraient aussi directement implanter en Oz les fonctionnalités qu'ils représentent, comme par exemple un serveur Web ou un serveur de bases de données. Nous pouvons réutiliser le schéma d'interconnexion DynamicFullInterconnect présenté en section 9.4.2 mettant en œuvre une interconnexion dynamique totale entre deux ensembles de composants.

11.3.3 Service de surveillance des systèmes distribués

Nous montrons ici comment décrire l'architecture hiérarchique d'un service de surveillance d'un système distribué (voir section 6.2.3, page 72). Cette architecture intègre des composants SensorPkg implantant les sondes du système de surveillance, et des composants AggregatorPkg qui agrègent plusieurs sources d'informations. Une sonde effectue périodiquement de nouvelles mesures sur la machine sur laquelle elle est déployée. Chaque machine à surveiller sera ainsi équipée d'une sonde, et chaque machine correspondant à un nœud non terminal de l'architecture hiérarchique (i.e. qui n'est pas une feuille de l'arbre) accueillera également un agrégateur dont les sources d'informations sont la sonde présente sur la machine ainsi que les agrégateurs présents sur les nœuds correspondant aux branches issues du nœud.

Pour le déploiement, nous utilisons une fonction MakeTree qui génère un arbre paramétré par son arité et par un ensemble de machines. L'arbre généré est représenté par des enregistrements Node = node(host:Host branches:Branches) pour lesquels Node.host désigne un composant représentant une machine et Node.branches est un tuple (éventuellement vide) listant les nœuds issus du nœud Node. La procédure récursive DeployTree parcourt la structure d'arbre ainsi générée, déploie les différents composants sur les machines en fonction de leur position dans l'architecture hiérarchique et établit les liaisons nécessaires. La figure 11.2 illustre l'architecture hiérarchique que nous mettons en œuvre ici sur un arbre binaire de 7 machines. Notons que tous les composants (sondes et agrégateurs) du service de surveillance sont placés dans un même composant composite représentant l'ensemble du système de surveillance. Ce composite n'est pas représenté sur la figure.

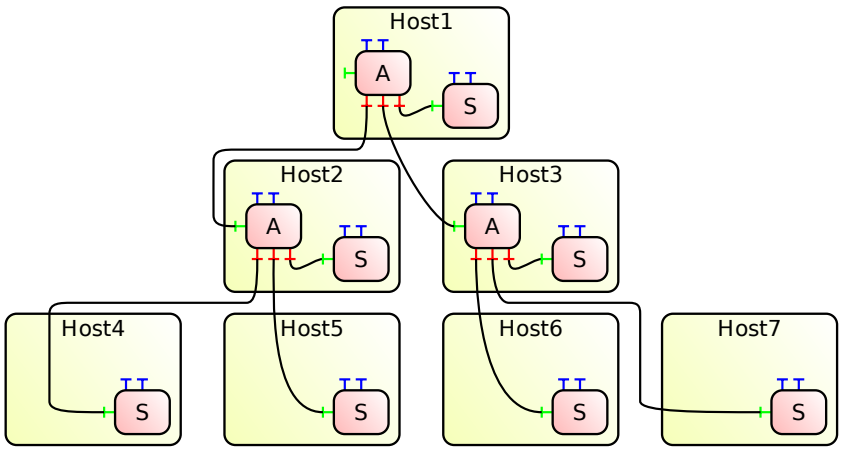


FIG. 11.2 – Architecture hiérarchique du service de surveillance. Sur cet exemple, il s'agit d'un arbre binaire s'étendant sur 7 machines. Les composants A désignent les agrégateurs, et les composants S désignent les sondes.

```
fun {MonitoringService SHosts QoSSpecs}
   %% Determine the correct parameters (tree and monitoring interval) for the given QoS constraints
```

```
    Arity Depth Tau
    {TreeMonitoringCapacityPlanning SHosts QoSSpecs Arity Depth Tau}

    %% Compute a tree from the given set of hosts
    TreeHosts = {MakeTree SHosts Arity}
in
    functor
    export Membrane
    define
        Comp = {CNew nil}

        proc {DeployTree ParentNode Node}
            Host = Node.host
            Branches = Node.branches
            %% Deploy a sensor locally, start it, and ask it for a new client interface
            Sensor = {RemoteDeploy Host SensorPkg}
            {CAddSubComponent Comp Sensor}
            {{IResolve {CGetInterface Sensor [sensor control]}} start}
            IClient = {{IResolve {CGetInterface Sensor [sensor control]}} addClient($)}
        in
            if (ParentNode == nil) then % Root node: deploy an aggregator locally
                Aggregator = {RemoteDeploy Host AggregatorPkg}
                {CAddSubComponent Comp Aggregator}
                %% Start the aggregator and bind it the to local sensor
                {{IResolve {CGetInterface Aggregator [sensor control]}} start}
                Binding = {BNew IClient {CGetInterface Aggregator [sensor input]}}
            in
                %% Deploy node branches
                {Record.forAll Branches proc {$ Branch} {TreeDeploy Node Branch} end}
            else % Internal or leaf node: get the parent aggregator
                ParentHost = ParentNode.host
                ParentAggregator = {SGetSingleton
                    {SFilter {CGetSubComponents ParentHost} {TaggedWithAll [monitoring aggregator]}}}
                IParentAggregatorInput = {CGetInterface ParentAggregator [sensor input]}
            in
                if ({Record.width Children} == 0) then % Leaf node: no local aggregator, use the parent's one
                    %% Bind the local sensor to the parent aggregator
                    Binding = {BNew IClient IParentAggregatorInput}
                else % Internal node: deploy an aggregator locally
                    Aggregator = {RemoteDeploy Host AggregatorPkg}
                    {CAddSubComponent Comp Aggregator}
                    IAggregatorClient = {{IResolve {CGetInterface Aggregator [sensor control]}} addClient($)}
                    %% Start the periodic aggregator, bind it to the parent aggregator and to the local sensor
                    {{IResolve {CGetInterface Aggregator [sensor control]}} start}
                    Binding1 = {BNew IAggregatorClient IParentAggregatorInput}
                    Binding2 = {BNew IClient {CGetInterface Aggregator [sensor input]}}
                in
                    %% Deploy node branches
                    {Record.forAll Branches proc {$ Branch} {TreeDeploy Node Branch} end}
                end
            end
        end
```

```
        {TreeDeploy nil TreeHosts}

        Membrane = Comp
    end
end
```

11.4 Implantation de l'auto-optimisation avec FructOz et LactOz

Nous montrons dans cette section comment implanter l'auto-optimisation par approvisionnement dynamique dans le cadre du canevas FructOz et de la bibliothèque LactOz. Nous reprenons et étendons les exemples de mise en œuvre des services à messages et des services Internet présentés dans la section précédente.

11.4.1 Observation d'un ensemble dynamique de composants

Nous reprenons dans cette section l'objectif illustrant l'intérêt des calculs dynamiques et de la bibliothèque LactOz. Cet objectif consiste à formuler et mettre en œuvre des propriétés de la forme : « Chaque machine hébergeant une instance de service dupliqué doit accueillir une sonde de surveillance qui est raccordée à un coordinateur central ».

Notre objectif est ici de décrire l'architecture d'un service de surveillance pour un composant redimensionnable. Il s'agit donc de décrire l'architecture du service de surveillance de telle sorte qu'elle s'adapte automatiquement aux évolutions de l'architecture du composant redimensionnable, comme illustré sur la figure 11.3.

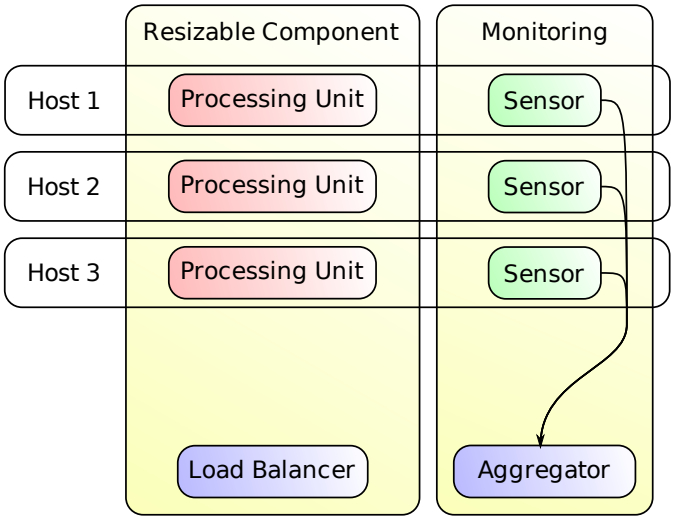


FIG. 11.3 – Architecture dynamique implicite du composant Monitoring de surveillance associé à un composant redimensionnable.

Dans cet objectif, nous implantons la fonction ClusteredDeploy : $\mathcal{S}\{\mathcal{C}\} \rightarrow \mathcal{S}\{\mathcal{C}\}$. Cette fonction construit l'ensemble dynamique implicite SComponents = {ClusteredDeploy SHosts Pkg} formé de composants Pkg qui sont distribués sur l'ensemble dynamique de machines SHosts. Cette fonction implante un observateur sur l'ensemble dynamique SHosts : lorsqu'une machine est ajoutée à l'ensemble SHosts, un nouveau composant Pkg est instancié sur cette machine, puis ajouté à l'ensemble dynamique implicite SComponents ; symétriquement, lorsqu'une machine est enlevée de l'ensemble SHosts, le composant Pkg déployé sur cette machine est supprimé de l'ensemble implicite puis oublié.

```
%% Helper navigation function:
%% Get the host component representing the machine on which the given component is deployed
fun {CGetHost Comp}
    {SGetSingleton {SFilter {CGetSuperComponents Comp} {TaggedWith 'Host'}}}
end

%% Deploy the given component (Package) on each host in the given set of hosts
%% Each component instance is made sub−component of the given Parent component
fun {ClusteredDeploy SHosts Package}
    %% Class for an implicit component set mapped over SHosts
    class ComponentSet from ImplicitSet
        meth init
            ImplicitSet,init
            %% Create and register a listener that forwards updates to the methods addHosts and removeHosts
            Listener = {New SetListenerForwarder init(self SHosts sync addHosts removeHosts)}
            {SHosts listen(Listener)}
        end

        %% Adding some hosts
        meth addHosts(SElements)
            {FSet.forAll SElements % for every new host
                proc {$ Host}
                    SubComp = {RemoteDeploy Host Package} % remotely deploy a component instance
                    Set,add(SubComp) % add the component to the implicit set
                end}
        end

        %% Removing some hosts
        meth removeHosts(SElements)
            {FSet.forAll SElements % for every removed host
                proc {$ Host}
                    Set,filterInPlace( % find and remove any component deployed on the removed host
                        fun {$ Comp} {CGetHost Comp} == Host end)
                end}
        end
    end

in
    %% Instantiate the implicit set
    {New ComponentSet init}
end
```

En nous appuyant sur la fonction ClusteredDeploy, nous construisons un service de

surveillance dynamique autonome, prenant la forme d'un composant composite dont l'architecture est pilotée par les évolutions de l'ensemble des composants dont on souhaite surveiller l'environnement d'exécution.

```
%% Dynamic monitoring architecture
fun {MonitoringPkg SComponents}
    %% Set of nodes hosting the components
    SHosts = {SMap SComponents CGetHost}
in
    functor $
    export Membrane
    define
        C = {CNew nil}

        %% Deploy the monitoring information aggregator
        Aggregator = {Deploy AggregatorPkg}
        {CAddSubComponent C Aggregator}

        %% Deploy a dynamic set of sensors on every host
        SSensors = {ClusteredDeploy SHosts SensorPkg}

        %% Bind all sensors to the aggregator
        SAggregators = {SSingleton Aggregator}
        {ApplyBindingScheme FullInterconnect SSensors SAggregator}

        Membrane = C
    end
end
```

En résumé, nous avons décrit l'architecture dynamique d'un service de surveillance en fonction d'un ensemble dynamique arbitraire de composants que l'on souhaite surveiller. Il s'agit ainsi du paramétrage d'une architecture dynamique par une autre architecture dynamique.

11.4.2 Auto-optimisation d'un composant redimensionnable

Nous mettons en application immédiate l'architecture dynamique du système de surveillance détaillée dans la section précédente. Cette architecture est paramétrée par l'ensemble dynamique des composants que nous souhaitons surveiller. Nous l'appliquons précisément à la mise en œuvre d'un composant redimensionnable auto-optimisé par approvisionnement dynamique et dirigé par des seuils. Le système de surveillance permet de surveiller l'état des ressources sous-jacentes aux instances dupliquées qui forment le composant redimensionnable. Il s'agit d'identifier l'ensemble des machines qui hébergent des instances du système dupliqué formant le service à messages en grappe. Ces instances sont identifiées parmi les sous-composants du composant redimensionnable modélisant le service à messages au moyen du marqueur "Processing Unit". Le système de surveillance ainsi mis en œuvre agrège ces informations et produit un indicateur synthétique global de la charge sur l'ensemble du composant redimensionnable. Cet indicateur est soumis à un seuil minimum et un seuil maximum afin de

piloter les opérations d'approvisionnement dynamique.

```
fun {SelfOptResizablePkg Package PkgDeploy PkgUndeploy LoadMin LoadMax Inhibitor InhibitionDelay}
    functor
    export Membrane
    define
        %% Initial number of processing unit instances
        Size = {VNew 1}
        Resizable = {ResizablePkg Pkg Size PkgDeploy PkgUndeploy}

        %% Deploy the dynamic monitoring infrastructure

        %% Set of processing units composing the resizable component
        SUnits = {SFilter {CGetSubComponents ResizableComp} {TaggedWith "Processing␣Unit"}}
        %% Monitor all the machines hosting a processing unit
        Monitor = {Deploy {MonitoringPkg SUnits}}

        %% Complete the loop

        %% Turn the aggregated monitoring data into a dynamic floating−point number
        Aggregator = {SGetSingleton {SFilter {CGetSubComponents Monitor} {TaggedWith 'Aggregator'}}}
        Load = {GetDynamicLoad Aggregator}
        %% Threshold−based controller driving the size of the resizable component
        %% with a stabilization delay
        {Threshold Load LoadMin LoadMax
            %% Underload action
            proc {$ Load}
                {Decrement Size} % remove one resource
                {Inhibitor.start InhibitionDelay} % stabilization delay
            end
            %% Overload action
            proc {$ Load}
                {Increment Size} % add one resource
                {Inhibitor.start InhibitionDelay} % stabilization delay
            end
            %% Activation precondition
            fun {$ Load}
                {Not {Inhibitor.isInhibited}}
            end
        }

        Membrane = Resizable
    end
end
```

Le composant redimensionnable auto-optimisé est paramétré par : Package, la description du sous-composant dupliqué ; PkgDeploy et PkgUndeploy, deux procédures chargées d'instancier et de supprimer les instances du sous-composant dupliqué ; LoadMin et LoadMax, les seuils minimum et maximum de charge déclenchant les opérations d'approvisionnement dynamique ; et enfin, Inhibitor, un gestionnaire d'inhibition pour assurer la stabilisation et prévenir les oscillations du composant redimensionnable, associé au délai d'inhibition InhibitionDelay. Le gestionnaire d'inhibition fournit une procédure {Inhibitor.start Delay} qui déclenche un délai d'inhibition pour la durée indiquée en para-

mètre, et une fonction {Inhibitor.isInhibited} qui indique si l'inhibition est activée ou non.

La fonction GetDynamicLoad construit un nombre flottant dynamique dont la valeur effective évolue régulièrement pour refléter l'état courant du système rapporté par les dernières mesures du système de surveillance. La fonction Threshold implante une réaction au dépassement de seuils minimum et maximum.

```
fun {Threshold Value Min Max UnderMinAction OverMaxAction Precondition}
    class ValueObserver from VariableListener
        attr current
        ...
        meth update(NewValue)
            lock
                if {Precondition @current} then
                    if (NewValue > Max) then
                        {UnderMinAction NewValue}
                    elsif (NewValue < Min) then
                        {OverMaxAction NewValue}
                    end
                end
                current := NewValue
            end
        end
    end
end
```

Nous externalisons le gestionnaire d'inhibition du composant redimensionnable afin de pouvoir partager l'inhibiteur entre plusieurs instances de composants redimensionnables. Nous générons un gestionnaire d'inhibition comme suit :

```
fun {NewInhibitor}
    Inhibited = {NewCell false}
    proc {StartStabilizationDelay Duration}
        Inhibited := true
        thread
            {Delay (Duration * 1000)}
            Inhibited := false
        end
    end
    fun {IsInhibited}
        @Inhibited
    end
in
    inhibitor(
        start: StartStabilizationDelay
        isInhibited: IsInhibited
    )
end
```

11.4.3 Auto-optimisation d'un service à messages

L'application du composant redimensionnable auto-optimisé à la construction d'un service à messages auto-optimisé est immédiate. Nous instancions l'architecture para-

métrée SelfOptResizablePkg. Nous utilisons un allocateur de machines simple (fonctions ClusterDeploy et ClusterUndeploy) comme celui présenté à la section 11.3.1 (page 168). Nous utilisons également un inhibiteur comme celui présentés à l'instant. Pour être pleinement exploitable, il reste à définir les seuils minimum et maximum déclenchant les reconfigurations dynamiques, ainsi que la longueur du délai d'inhibition pour la stabilisation du système auto-optimisé.

```
fun {MessageService Cluster JMSPkg}
    ... % Simple host allocator (procedures ClusterDeploy and ClusterUndeploy)

    Inhibitor = {NewInhibitor} % create a local inhibition manager
    LoadMin = ... % minimum and maximum load thresholds..
    LoadMax = ... % ..(manually adjusted)
    InhibitionDelay = ...
in
    {SelfOptResizablePkg JMSPkg ClusterDeploy ClusterUndeploy LoadMin LoadMax Inhibitor InhibitionDelay}
end
```

11.4.4 Auto-optimisation d'un service Internet multi-étagé

De façon analogue, nous mettons en œuvre un service Internet multi-étagé auto-optimisé par approvisionnement dynamique de chacun des différents étages. L'allocateur de machines ainsi que le gestionnaire d'inhibition que nous implantons dans cette architecture sont partagés par les trois composants redimensionnables correspondant aux trois étages du système en pipeline. Cela permet précisément d'implanter la gestion des oscillations de notre politique d'auto-optimisation décrite à la section 7.1.2 (page 84). Pour être exploitable, il reste à définir, pour chacun des étages composant le système en pipeline, les seuils minimum et maximum déclenchant les reconfigurations dynamiques, ainsi que la longueur du délai d'inhibition pour la stabilisation de l'auto-optimisation.

```
fun {InternetService Cluster WebPkg AppPkg DBPkg WebAppBindScheme AppDBBindScheme}
    ... % Shared host allocator (procedures ClusterDeploy and ClusterUndeploy)
    Inhibitor = {NewInhibitor} % create a shared inhibition manager

    %% Tunable thresholds (manually adjusted)
    WebLoadMin = ...
    WebLoadMax = ...
    WebInhibitionDelay = ...
    AppLoadMin = ...
    AppLoadMax = ...
    AppInhibitionDelay = ...
    DBLoadMin = ...
    DBLoadMax = ...
    DBInhibitionDelay = ...
in
    functor
    export Membrane
        Comp = {CNew nil}
```

```
        %% Pipeline of resizable components
        WebSizable = {Deploy {SelfOptResizablePkg WebPkg ClusterDeploy ClusterUndeploy
            WebLoadMin WebLoadMax Inhibitor WebInhibitionDelay}}
        {CAddSubComponent Comp WebSizable}
        AppSizable = {Deploy {SelfOptResizablePkg AppPkg ClusterDeploy ClusterUndeploy
            AppLoadMin AppLoadMax Inhibitor AppInhibitionDelay}}
        {CAddSubComponent Comp AppSizable}
        DBSizable = {Deploy {SelfOptResizablePkg DBPkg ClusterDeploy ClusterUndeploy
            DBLoadMin DBLoadMax Inhibitor DBInhibitionDelay}}
        {CAddSubComponent Comp DBSizable}

        %% Locate the set of processing units of each resizable component
        SWebServers = {SFilter {CGetSubComponents WebSizable} {TaggedWith 'Processing␣Unit'}}
        SAppServers = {SFilter {CGetSubComponents AppSizable} {TaggedWith 'Processing␣Unit'}}
        SDBServers = {SFilter {CGetSubComponents DBSizable} {TaggedWith 'Processing␣Unit'}}

        %% Apply the dynamic interconnection scheme
        {ApplyBindingScheme WebAppBindScheme SWebServers SAppServers}
        {ApplyBindingScheme AppDBBindScheme SAppServers SDBServers}

        Membrane = Comp
    end
end
```

11.5 Synthèse

Nous avons présenté dans ce chapitre trois éléments d'évaluation du canevas FructOz de construction de systèmes distribués fondés sur des architectures dynamiques. (1) Notre évaluation des performances de différentes stratégies de déploiement distribué montre l'intérêt offert par le canevas FructOz de pouvoir spécialiser et programmer précisément les procédures de déploiement des architectures. (2) Notre comparaison avec la plate-forme de déploiement distribué et de gestion de cycle de vie SmartFrog révèle plusieurs lacunes de SmartFrog issues principalement de l'absence de paramétrage et des limitations d'usage des coordinations et synchronisations pour contrôler les déploiements distribués. Ces éléments montrent l'apport de notre canevas FructOz par rapport à la plate-forme SmartFrog représentative de l'état de l'art en matière de déploiements distribués de systèmes à composants. Enfin, (3) nous exhibons des descriptions d'architectures réelles correspondant à des services à messages, des services Internet ou encore des services de surveillance reposant sur une architecture hiérarchique. Le canevas FructOz offre un haut niveau d'expression pour ces différentes formes d'architectures en permettant notamment d'abstraire plusieurs schémas d'architectures paramétrées qui peuvent ainsi être réutilisées et spécialisées dans différents contextes.

Au cours de ces différentes études de cas, nous avons construit plusieurs briques de base de systèmes distribués fondés sur des architectures dynamiques complexes. Ces briques architecturales prennent la forme d'abstractions (fonctions génériques) qui peuvent être largement réutilisées et spécialisées dans différents contextes, conférant

alors à la plate-forme intégrant le canevas FructOz avec la bibliothèque LactOz un très haut niveau d'expressivité, de réutilisabilité et d'extensibilité.

Notons enfin que les différents extraits de code présentés dans cette évaluation ont été simplifiés afin d'en faciliter la lecture et la compréhension. En conséquence, ces extraits de code ne sont pas nécessairement optimaux et pourraient faire l'objet d'améliorations diverses.

Le chapitre suivant présente nos conclusions sur l'ensemble des travaux réalisés au cours de notre étude.

Quatrième partie

Conclusion

Chapitre 12

Conclusions et perspectives

Résumé des contributions

Cette thèse a pour objectif l'étude des techniques de mise en œuvre de systèmes autonomes, et plus spécifiquement de systèmes auto-optimisés. L'analyse des besoins liés à ces systèmes nous a conduit à articuler notre étude autour de trois problématiques, contribuant chacune à des techniques complémentaires pour la construction de systèmes autonomes auto-optimisés.

Nous nous interrogeons tout d'abord sur les politiques d'optimisation des performances des systèmes distribués permettant de réduire leur consommation d'énergie, de maximiser la rentabilité des infrastructures et d'optimiser leurs performances en particulier lorsque l'environnement dans lequel le système évolue change au cours du temps, par exemple, en raison de variations de charges progressives ou brutales, etc.

Les politiques d'auto-optimisation étudiées s'appuient sur des adaptations des systèmes distribués, comme par exemple des opérations d'approvisionnement dynamique. Ces actions dirigées sur le système administré doivent être contrôlées et coordonnées afin notamment d'assurer la cohérence globale des adaptations commandées. Cela nous conduit à étudier les techniques de description de systèmes fondés sur des architectures (ADL) dynamiques, autorisant l'expression d'évolutions complexes.

La construction de systèmes aux architectures dynamiques introduit une source supplémentaire de complexité. La nature éphémère des architectures dynamiques rend les techniques usuelles de calcul inexploitables, car les résultats produits sont immédiatement périmés. Nous nous interrogeons donc sur la manière d'observer et de nommer l'architecture dynamique des systèmes administrés, de façon à pouvoir exprimer des opérations, même si ces architectures évoluent dynamiquement.

Nous répondons à chacune de ces problématiques comme suit.

Nous mettons en œuvre dans la plate-forme d'administration autonome Jade des politiques d'auto-optimisation au moyen de boucles de commande. Ces boucles de commande sont dirigées par la surveillance d'indicateurs divers, dont notamment les consommations des ressources. Ces indications commandent des adaptations par redi-

mensionnement dynamique des systèmes administrés. L'algorithme d'auto-optimisation est intéressant par sa généralité et sa simplicité de mise en œuvre. Nos expérimentations démontrent son efficacité et sa mise en pratique sur des systèmes variés.

Afin d'adapter les systèmes administrés, nous décrivons et mettons en œuvre FructOz, un modèle de composant et un modèle de déploiement parfaitement intégrés au langage Oz et à sa plate-forme distribuée Mozart. Cette intégration confère instantanément à notre ADL des capacités de paramétrage et de synchronisation issues du langage Oz. Nous construisons ainsi un ADL dynamique d'ordre supérieur.

En soutien à la construction d'architectures dynamiques, nous implantons LactOz, un modèle de calcul dynamique supportant une bibliothèque de primitives de navigation dans les architectures. LactOz capture le dynamisme des architectures et élève le niveau d'expression en permettant de raisonner et de calculer sur des structures dynamiques.

Perspectives

Dans le sillage de l'informatique autonome, cette thèse ambitionnait la construction de systèmes autonomes auto-optimisés. Sans totalement atteindre cet objectif audacieux, nos réalisations constituent néanmoins une étape significative vers la construction de systèmes autonomes. Et cette étape ouvre la voie à de nombreuses extensions et perspectives que nous détaillons ici.

Nous mettons en œuvre des politiques d'auto-optimisation au moyen de boucles de commande qui doivent être paramétrées (par exemple, par des seuils de qualité de service, etc). Le paramétrage des boucles de commande est une tâche qui peut s'avérer complexe et délicate. L'automatisation de ce paramétrage constitue un défi majeur pour faciliter l'exploitation des systèmes auto-optimisés. En particulier, les modèles et notamment les techniques issues de la théorie de la commande semblent constituer une piste particulièrement prometteuse dans cet objectif. Dans un premier temps, cela peut prendre la forme d'outils d'aide à l'analyse, au diagnostic et au dimensionnement des systèmes administrés. Il semble également pertinent d'intégrer les latences des opérations d'approvisionnement dynamique et celles du système de surveillance associé à la boucle de commande afin d'améliorer la précision de l'auto-optimisation. Enfin, l'hypothèse d'homogénéité sur les infrastructures administrées est une limitation que nous pourrions tenter de lever en étudiant l'intégration des caractéristiques des machines composant l'infrastructure dans les décisions liées à l'approvisionnement dynamique.

Un autre ensemble de perspectives concernerait plus particulièrement FructOz. En effet, notre canevas à composants FructOz constitue une base solide à partir de laquelle nous pouvons pousser l'étude des mécanismes pour l'administration des systèmes distribués. Nous pouvons par exemple envisager de doter FructOz de capacités transactionnelles, qui pourraient être construites comme une généralisation des espaces de calculs Oz à tout type de calcul ainsi qu'aux environnements distribués. L'absence de typage du langage Oz est à la fois une force et une faiblesse. Il serait intéressant d'étudier le typage des systèmes à architectures, par exemple dans le cadre d'Alice ML, un

langage fortement typé et inspiré d'Oz et de ML. Pour finir, il faudrait finaliser l'intégration de FructOz avec le langage Oz par introduction d'éléments syntaxiques dans langage Oz, et enfin compléter FructOz par une bibliothèque complète de patrons de coordination.

Enfin, les perspectives concernant LactOz sont les suivantes. Les calculs dynamiques mis en œuvre dans LactOz reposent sur des structures distribuées complexes qui constituent en soi un système distribué qui pourrait faire l'objet de nombreuses optimisations. Nous pourrions en dernier lieu envisager une intégration des expressions dynamiques LactOz dans le langage Oz. Les expressions LactOz pourraient également tirer parti d'un système de typage permettant de vérifier leur validité.

Bibliographie

[1] MySQL AB. The MySQL Open Source Relational Database Management System, 1995. Available at URL : *http://www.mysql.com*.

[2] Takoua Abdellatif, Jakub Kornas, and Jean-Bernard Stefani. J2EE Packaging, Deployment and Reconfiguration Using a General Component Model. In *3rd International Working Conference on Component Deployment (CD'2005)*, volume 3798 of *Lecture Notes in Computer Science*. Springer, 2005.

[3] Tarek F. Abdelzaher and Nina Bhatti. Web Content Adaptation to Improve Server Overload Behavior. *Computer Networks*, 31(11-16) :1563–1577, 1999.

[4] Tarek F. Abdelzaher, Kang G. Shin, and Nina T. Bhatti. Performance Guarantees for Web Server End-Systems : A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1) :80–96, 2002.

[5] Stephen Adler. The Slashdot Effect, An Analysis of Three Internet Publications. *Linux Gazette*, 38, March 1999.

[6] Anatoly Akkerman, Alexander Totok, and Vijay Karamcheti. Infrastructure for Automatic Dynamic Deployment of J2EE Applications in Distributed Environments. In *3rd International Working Conference on Component Deployment (CD'2005)*, number 3798 in Lecture Notes in Computer Science. Springer, 2005.

[7] Amazon.com Inc. Amazon site, 2007. *http://www.amazon.com*.

[8] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Alan L. Cox, Sameh Elnikety, Romer Gil, Julie Marguerite, Karthick Rajamani, and Willy Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. In *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, November 2002.

[9] Paul Anderson, Patrick Goldsack, and Jim Paterson. SmartFrog Meets LCFG : Autonomous Reconfiguration with Central Policy Control. In *17th Conference on Systems Administration (LISA 2003)*. USENIX, 2003.

[10] Karen Appleby, Sameh A. Fakhouri, Liana L. Fong, Germán S. Goldszmidt, Michael H. Kalantar, S. Krishnakumar, Donald P. Pazel, John A. Pershing, and B. Rochwerger. Océano-SLA based management of a computing utility. In *Proceedings of Integrated Network Management*, pages 855–868, 2001.

[11] Jean Arnaud and Sara Bouchenak. Gestion de Ressources dans les Services Internet. In *6ème Conférence Française des Systèmes d'Exploitation (CFSE'6)*, 2008.

[12] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster Reserves : a mechanism for resource management in cluster-based network servers. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'00)*, pages 90–101, Santa Clara, California, United States, 2000.

[13] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems. *Software Quality Control Journal*, 15(3) :265–281, 2007.

[14] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers : A New Facility for Resource Management in Server Systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 45–58, New Orleans, Louisiana, February 1999.

[15] María Cecilia Bastarrica, Alexander A. Shvartsman, and Steven A. Demurjian. A Binary Integer Programming Model for Optimal Object Distribution. In *2nd International Conference on Principles Of Distributed Systems (OPODIS'98)*. Hermes, 1999.

[16] Josep M. Blanquer, Antoni Batchelli, Klaus Schauser, and Rich Wolski. Quorum : Flexible Quality of Service for Internet Services. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation (NSDI'05)*, pages 159–174, 2005.

[17] Sara Bouchenak, Fabienne Boyer, Sacha Krakowiak, Daniel Hagimont, Adrian Mos, Noel de Palma, Vivien Quéma, and Jean-Bernard Stefani. Architecture-Based Autonomous Repair Management : An Application to J2EE Clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005)*. IEEE Computer Society, 2005.

[18] Sara Bouchenak and Noel de Palma. Message Queuing Systems. *Springer Encyclopedia of Database Systems*, 2008. (to appear).

[19] Sara Bouchenak, Noel de Palma, Daniel Hagimont, and Christophe Taton. Autonomic Management of Clustered Applications. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing (Cluster)*, September 2006.

[20] Jeremy S. Bradbury, James R. Cordy, Jürgen Dingel, and Michel Wermelinger. A Survey of Self-Management in Dynamic Software Architecture Specifications. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004)*. ACM, 2004.

[21] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12) :1257–1284, September 2006.

[22] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The Fractal Component Model. *The ObjectWeb Consortium*, 2004. Available at URL : *http://fractal.objectweb.org*.

[23] Denis Caromel, Alexandre di Costanzo, and Christian Delbé. Peer-to-Peer and fault-tolerance : Towards deployment-based technical services. *Future Generation Computer Systems*, 23(7) :879–887, 2007.

[24] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-JDBC : Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference, Freenix track*, Boston, MA, June 2004. *http://c-jdbc.objectweb.org/*.

[25] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the Eleventh IEEE/ACM International Workshop on Quality of Service (IWQoS 2003)*, Monterey, CA, June 2003.

[26] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. Managing Energy and Server Resources in Hosting Centres. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP'01)*, pages 103–116, Banff, Alberta, Canada, October 2001.

[27] Antonin Chazalet and Philippe Lalanda. A Meta-Model Approach for the Deployment of Services-oriented Applications. In *IEEE International Conference on Services Computing (SCC 2007)*. IEEE Computer Society, 2007.

[28] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley R. Schmerl, and Peter Steenkiste. Rainbow : Architecture-Based Self-Adaptation with Reusable Infrastructure. In *1st International Conference on Autonomic Computing (ICAC 2004)*. IEEE Computer Society, 2004.

[29] CiteSeer.IST. CiteSeer : Scientific Literature Digital Library. Available at URL : *http://citeseer.ist.psu.edu*.

[30] Oz Consortium. The Mozart Programming System. Available at URL : *http://www.mozart-oz.org*.

[31] Oz Consortium. Mozart Documentation, 2004. Available at the URL : *http://www.mozart-oz.org/documentation*.

[32] The ObjectWeb Consortium. The JOnAS Open Source EE Application Server. Available at URL : *http://jonas.objectweb.org*.

[33] The ObjectWeb Consortium. JORAM : Java Open Reliable Asynchronous Messaging, 2002. Available at URL : *http://www.objectweb.org/joram*.

[34] The ObjectWeb Consortium. Julia : Fractal Composition Framework Reference Implementation, 2002. Available at URL : *http://www.objectweb.org/fractal*.

[35] W3C : World Wide Web Consortium. XML Path Language (XPath), Version 1.0, W3C Recommendation, November 1999. Available at URL : *http://www.w3.org/TR/xpath*.

[36] Cougaar Web site. Cougaar : An Open Source Agent Architecture for Large-scale, Distributed Multi-Agent Systems. Available at URL : *http://www.cougaar.org*.

[37] Geoff Coulson, Gordon S. Blair, Paul Grace, François Taïani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A Component Model

for Building Systems Software. *ACM Transactions on Computer Systems (TOCS)*, 26(1) :1–42, February 2008.

[38] Dailymotion. Dailymotion : a video streaming server, 2005. Available at URL : *http://www.dailymotion.com*.

[39] Pierre-Charles David. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspects d'adaptation*. PhD thesis, Université de Nantes, France, 2005.

[40] Pierre-Charles David and Thomas Ledoux. WildCAT : a generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing (MPAC '05)*, pages 1–7, New York, NY, USA, 2005. ACM.

[41] Noel de Palma, Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sylvain Sicard, and Christophe Taton. Jade : Un Environnement d'Administration Autonome. *Techniques et Sciences Informatiques (TSI)*, to appear.

[42] Frank Denis. PLB, the Pure Load-Balancer : A free high-performance load balancer for Unix. Available at URL : *http://plb.sunsite.dk*.

[43] Yixin Diao, Neha Gandhi, Joseph L. Hellerstein, Sujay Parekh, and Dawn M. Tilbury. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. In *IEEE/IFIP Network Operations and Management Symposium (NOMS 2002)*, pages 219–234, 2002.

[44] Yixin Diao, Joseph L. Hellerstein, Sujay Parekh, Rean Griffith, Gail E. Kaiser, and Dun Phung. A control theory foundation for self-managing computing systems. *Selected Areas in Communications, IEEE Journal on*, 23(12) :2213–2222, December 2005.

[45] eBay Inc. eBay Web site, 2007. Available at URL : *http://www.ebay.com*.

[46] Areski Flissi and Philippe Merle. A Generic Deployment Framework for Grid Computing and Distributed Applications. In *OTM Confederated International Conferences, Grid computing, high performAnce and Distributed Applications (GADA 2006)*, volume 4276 of *Lecture Notes in Computer Science*, pages 1402–1411. Springer, 2006.

[47] The Apache Software Foundation. The Apache HTTP Web server. Available at URL : *http://httpd.apache.org*.

[48] The Apache Software Foundation. The Apache Tomcat Connector. Available at URL : *http://tomcat.apache.org/connectors-doc*.

[49] The Apache Software Foundation. The Apache Tomcat Servlet container. Available at URL : *http://tomcat.apache.org*.

[50] Ethan Galstad. Nagios : an open source computer system and network monitoring application software, 2002. Available at URL : *http://www.nagios.org*.

[51] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design Patterns : Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference Object-Oriented Programming (ECOOP'93)*, pages 406–431, Kaiserslautern, Germany, July 1993.

[52] Alan G. Ganek and Thomas A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1) :5–18, January 2003.

[53] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. Rainbow : Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10) :46–54, October 2004.

[54] David Garlan, Robert T. Monroe, and David Wile. Acme : Architectural Description of Component-Based Systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[55] Sébastien Godard. SysStat : Performance monitoring tools for Linux, 1999. Available at URL : *http://pagesperso-orange.fr/sebastien.godard/*.

[56] Patrick Goldsack. SmartFrog : Configuration, Ignition and Management of Distributed Applications. Technical report, HP Research Labs, 2003. Available at URL : *http://www-uk.hpl.hp.com/smartfrog*.

[57] Grid'5000 Web site. Grid'5000 : the experimental grid platform. *http://www.grid5000.fr*.

[58] Frédéric Guidec, Yves Maheo, and Luc Courtrai. A Java Middleware Platform for Resource-Aware Distributed Applications. In *Proceedings of the 2nd International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 96–103, Oct 2003.

[59] Ashish Gupta and Inderpal S. Mumick. Maintenance of Materialized Views : Problems, Techniques, and Applications. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(2) :3–19, June 1995.

[60] Seif Haridi and Nils Franzén. Tutorial of Oz, 2004. Available at the URL : *http://www.mozart-oz.org/documentation/tutorial/index.html*.

[61] Red Hat. The JBoss Application server. Available at URL : *http://www.jboss.com*.

[62] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley - IEEE, 2004.

[63] Paul Horn. Autonomic Computing : IBM's Perspective on the State of Information Technology. IBM T.J.Watson Labs, October 2001. See URL : *http://www.ibm.com/research/autonomic*.

[64] VMware Inc. VMware : Virtualization via Hypervisor, Virtual Machine & Server Consolidation. Available at URL : *http://www.vmware.com*.

[65] Armann Ingolfsson and Ekkehard W. Sachs. Stability and sensitivity of an EWMA controller. *Journal of Quality Technology*, 25(4) :271–287, October 1993.

[66] Ackbar Joolia, Thais Batista, Geoff Coulson, and Antonio Tadeu A. Gomes. Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform. In *5th IEEE/IFIP Conference on Software Architecture (WICSA'05)*, volume 0, pages 131–140. IEEE Computer Society, 2005.

[67] Gail E. Kaiser, Janak J. Parekh, Philip Gross, and Giuseppe Valetto. Kinesthetics eXtreme : An External Infrastructure for Monitoring Distributed Legacy Systems. In *5th Annual International Workshop on Active Middleware Services (AMS 2003)*. IEEE Computer Society, 2003.

[68] Alexander Keller and Rémi Badonnel. Automating the Provisioning of Application Services with the BPEL4WS Workflow Language. In *15th IFIP/IEEE International Workshop on Distributed Systems : Operations and Management (DSOM)*, volume 3278 of *Lecture Notes in Computer Science*, pages 15–27. Springer, 2004.

[69] Alexander Keller and Heiko Ludwig. The WSLA Framework : Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1) :57–81, March 2003.

[70] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1) :41–50, 2003.

[71] Tatiana Kichkaylo and Vijay Karamcheti. Optimal Resource-Aware Deployment Planning for Component-Based Distributed Applications. In *13th International Symposium on High-Performance Distributed Computing (HPDC 2004)*. IEEE Computer Society, 2004.

[72] Derek Long and Maria Fox. PDDL2.1 : An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research (JAIR)*, 20 :61–124, 2003.

[73] Canonical Ltd. Upstart : an event-based replacement for the `init` daemon, August 2006. Available at URL : *http://upstart.ubuntu.com*.

[74] Chenyang Lu, Tarek F. Abdelzaher, John A. Stankovic, and Sang H. Son. A feedback control approach for guaranteeing relative delays in Web servers. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTTAS)*, pages 51–62, 2001.

[75] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system : design, implementation, and experience. *Parallel Computing*, 30(7) :817–840, July 2004.

[76] Nenad Medvidovic and Marija Mikic-Rakic. Architectural Support for Programming-in-the-Many. Technical Report USC-CSE-2001-506, University of Southern California, October 2001.

[77] Sun MicroSystems. J2EE : Java 2 Platform, Enterprise Edition, 2002. Available at URL : *http://java.sun.com/j2ee/docs.html*.

[78] Sun MicroSystems. Java Message Service Specification Final Release 1.1, Mars 2002. Available at URL : *http://java.sun.com/products/jms/docs.html*.

[79] Sun MicroSystems. JavaEE : Java Platform, Enterprise Edition, May 2006. Available at URL : *http://java.sun.com/javaee/technologies/*.

[80] Marija Mikic-Rakic, Sam Malek, and Nenad Medvidovic. Improving Availability in Large, Distributed Component-Based Systems Via Redeployment. In *3rd International Working Conference on Component Deployment (CD'2005)*, number 3798 in Lecture Notes in Computer Science. Springer, 2005.

[81] Marija Mikic-Rakic and Nenad Medvidovic. Architecture-Level Support for Software Component Deployment in Resource Constrained Environments. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD'2002)*, number 2370 in Lecture Notes in Computer Science. Springer, 2002.

[82] Marija Mikic-Rakic and Nenad Medvidovic. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. *Lecture Notes in Computer Science*, 2672 :162–181, 2003.

[83] Jesús M. Milan-Franco, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Bettina Kemme. Adaptive Middleware for Data Replication. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware (Middleware'04)*, volume 78, pages 175–194, Toronto, Canada, 2004.

[84] Berkeley Millenium Project, University of California. The Ganglia scalable distributed monitoring system, 2004. Available at URL : *http://ganglia.info/*.

[85] Robert T. Monroe. Capturing Software Architecture Design Expertise with Armani, 1998. Revised January, 2001.

[86] Justin Moore, David Irwin, Laura Grit, Sara Sprenkle, and Jeffrey S. Chase. Managing Mixed-Use Clusters with Cluster-on-Demand. Technical report, Internet Systems and Storage Group, Duke University, 2002. Available at URL : *http://issg.cs.duke.edu/cod-arch.pdf*.

[87] Jayaprakash Nagapraveen, Thierry Coupaye, Christine Collet, and Pierre-Charles David. Flexible Reactive Capabilities in Component-Based Autonomic Systems. In *5th IEEE Workshop on Engineering of Autonomic and Autonomous Systems (EASE 2008)*, pages 97–106, March 2008.

[88] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI'04)*, pages 61–76, Berkeley, CA, USA, 2004. USENIX Association.

[89] James Norris, Keith Coleman, Armando Fox, and George Candea. OnCall : Defeating Spikes with a Free-Market Application Cluster. In *1st International Conference on Autonomic Computing (ICAC'04)*, pages 198–205, New York, NY, USA, May 2004.

[90] Vivek S. Pai, Mohit Aron, Gaurov Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware Request Distribution in cluster-based network servers. In *Proceedings of the 8th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 205–216, San Jose, California, United States, 1998.

[91] Justin M. Paluska, Hubert Pham, Umar Saif, Grace Chau, Chris Terman, and Steve Ward. Structured Decomposition of Adaptive Applications. In *6th annual IEEE International Conference on Pervasive Computing and Communications (PerCom'2008)*, pages 1–10, March 2008.

[92] Marta Patiño-Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R : Consistent database replication at the middleware level. *ACM Transactions on Computer Systems (TOCS)*, 23(4) :375–423, 2005.

[93] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Computing Surveys (CSUR)*, 31(1) :63–103, 1999.

[94] PostgreSQL. The PostgreSQL Open Source Object-Relational Database Management System. Available at URL : *http://www.postgresql.org*.

[95] Vivien Quéma, Roland Balter, Luc Bellissard, David Féliot, André Freyssinet, and Serge Lacourte. ScalAgent, une plate-forme à composants pour applications asynchrones. *Techniques et Sciences Informatiques (TSI)*, 23 :253–274, 2004.

[96] Martin Reiser and Stephen S. Lavenberg. Mean-Value Analysis of Closed Multichain Queuing Networks. *Journal of the ACM (JACM)*, 27(2) :313–322, April 1980.

[97] Kai Shen, Hong Tang, Tao Yang, and Lingkun Chu. Integrated resource management for cluster-based internet services. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 225–238, Boston, Massachusetts, USA, December 2002.

[98] Kai Shen, Tao Yang, Lingkun Chu, Joanne Holliday, Douglas A. Kuschner, and Huican Zhu. Neptune : Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS'01)*, pages 197–208, San Francisco CA, March 2001.

[99] Kelvin C. W. So and Emin Gün Sirer. Latency and bandwidth-minimizing failure detectors. In *EuroSys '07 : Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 89–99, New York, NY, USA, 2007. ACM.

[100] Gokul Soundararajan and Cristiana Amza. Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers. Technical report, Department of Electrical and Computer Engineering, University of Toronto, 2005.

[101] Gokul Soundararajan, Cristiana Amza, and Ashvin Goel. Database Replication Policies for Dynamic Content Applications. In *First ACM SIGOPS EuroSys Conference (EuroSys 2006)*, Leuven, Belgium, April 2006.

[102] Jean-Bernard Stefani and Christophe Taton. Automating deployment with Oz. Technical report, INRIA, September 2008. *to appear*.

[103] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation (NSDI'2005)*, pages 71–84, 2005.

[104] Vanish Talwar, Qinyi Wu, Calton Pu, Wenchang Yan, Gueyoung Jung, and Dejan Milojicic. Comparison of Approaches to Service Deployment. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'2005)*, pages 543–552, Washington, DC, USA, 2005. IEEE Computer Society.

[105] Naoyuki Tamura. Cream : Class Library for Constraint Programming in Java, 2004. Available at URL : *http://bach.istc.kobe-u.ac.jp/cream*.

[106] Christophe Taton, Sara Bouchenak, Noël de Palma, and Daniel Hagimont. Self-Optimization of Internet Services with Dynamic Resource Provisioning. Technical Report RR-6575, INRIA - Sardes Project, July 2008.

[107] Christophe Taton, Noel de Palma, Daniel Hagimont, Sara Bouchenak, and Jérémy Philippe. Self-Optimization of Clustered Message-Oriented Middleware. In *International Conference on Distributed Objects and Applications (DOA)*, volume 4803 of *Lecture Notes in Computer Science*. Springer, 2007.

[108] Christophe Taton, Noel de Palma, Jérémy Philippe, and Sara Bouchenak. Improving the Performances of JMS-Based Applications. *International Journal of Autonomic Computing (IJAC)*, to appear.

[109] Chouky Tibermacine, Didier Hoareau, and Reda Kadri. Enforcing Architecture and Deployment Constraints of Distributed Component-Based Software. In *10th International Conference Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *Lecture Notes in Computer Science*. Springer, 2007.

[110] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. Analytic modeling of multitier internet applications. *ACM Transaction on the Web*, 1(1) :2, 2007.

[111] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant J. Shenoy, Mike Spreitzer, and Asser N. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, pages 291–302, Banff, Alberta, Canada, June 2005.

[112] Bhuvan Urgaonkar and Prashant Shenoy. Cataclysm : Handling Extreme Overloads in Internet Services. Technical Report TR03-40, Department of Computer Science, University of Massachusetts, November 2004.

[113] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, and Pawan Goyal. Dynamic Provisioning of Multi-tier Internet Applications. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC'05)*, Seattle, June 2005.

[114] Bhuvan Urgaonkar and Prashant J. Shenoy. Sharc : Managing CPU and Network Bandwidth in Shared Clusters. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(1) :2–17, January 2004.

[115] Bhuvan Urgaonkar and Prashant J. Shenoy. Cataclysm : policing extreme overloads in internet applications. In *Proceedings of the 14th international conference on World Wide Web (WWW'2005)*, pages 740–749, Chiba, Japan, May 2005.

[116] Bhuvan Urgaonkar, Prashant J. Shenoy, and Timothy Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI'02)*, Boston, Massachusetts, USA, December 2002.

[117] Giuseppe Valetto and Gail E. Kaiser. Combining Mobile Agents and Process-based Coordination to Achieve Software Adaptation. Technical report, Columbia University, Department of Computer Science, 2002.

[118] Giuseppe Valetto and Gail E. Kaiser. Using Process Technology to Control and Coordinate Software Adaptation. In *25th International Conference on Software Engineering (ICSE'2003)*. IEEE Computer Society, 2003.

[119] Giuseppe Valetto, Gail E. Kaiser, and Gaurav S. Kc. A Mobile Agent Approach to Process-based Dynamic Adaptation of Complex Software Systems. In *8th European Workshop on Software Process Technology*, volume 2077 of *Lecture Notes in Computer Science*. Springer, 2001.

[120] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1) :5–51, November 2003.

[121] André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf. Software Architecture, Configuration Management, and Configurable Distributed Systems : A Ménage a Trois. Technical report, CU-CS-849-98, Department of Computer Science, University of Colorado, Boulder, Colorado, USA, 1998.

[122] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling : Flexible Proportional-Share Resource Management. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, California, November 1994.

[123] Matt Welsh and David Culler. Adaptive Overload Control for Busy Internet Servers. In *Proceedings of the 4th USENIX Conference on Internet Technologies and Systems (USITS'2003)*, March 2003.

[124] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA : An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP'2001)*, pages 230–243, Banff, Alberta, Canada, October 2001.

[125] Steve R. White, James E. Hanon, Ian Whalley, David M. Chess, and Jeffrey O. Kephart. An Architectural Approach to Autonomic Computing. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'2004)*, 2004.

[126] YouTube. YouTube : a video streaming server, 2005. Available at URL : *http://www.youtube.com*.

[127] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-tier Applications. In *Proceedings of the 4th International Conference on Autonomic Computing (ICAC'2007)*, page 27, Jacksonville, Florida, USA, June 2007.

[128] Huican Zhu, Hong Tang, and Tao Yang. Demand-driven Service Differentiation in Cluster-based Network Servers. In *Proceedings of the 20th IEEE International Conference on Computer Communications (INFOCOM'01)*, volume 2, pages 679–688, Anchorage, Alaska, April 2001.

Bibliothèque de schémas de synchronisations

Barrière de synchronisation

```
%% {Barrier ActionList}
%% ActionList: list of procedures (no parameter)
%% Barrier synchronization pattern
proc {Barrier ActionList}
   N = {List.length ActionList}
   Bar = {Tuple.make barrier N}
   I = {NewCell 1}
in
   for Action in ActionList do
      J = @I
   in
      thread
         {Action}
         Bar.J = true % ready signal once the procedure is over
      end
      I := J+1
   end
   {Record.forAll Bar Wait} % wait for all N ready signals
end
```

Délai de garde

```
%% {Timeout Action Length}
%% Action: procedure (no parameter)
%% Length: timeout in milliseconds
%% Allow execution of procedure Action during Length milliseconds
proc {Timeout Action Length}
   Timeout  % timeout synchronization variable
   Completed  % completion synchronization variable
   ActionThread
in
   thread
      ActionThread = {Thread.this}
      {Action}
      Completed = true
   end
   thread
      {Delay Length}
      Timeout = true
   end
   {WaitOr Completed Timeout}
```

```
    if {IsDet Timeout} then
        {Thread.terminate ActionThread}
    end
end
```

Gestion d'erreur

```
%% {HandleError Action Handler}
%% Action: procedure (no parameter)
%% Handler: procedure (no parameter)
%% Execute Action. On error, execute the given Handler
proc {HandleError Action Handler}
    try
        {Action}
    catch Exn then
        {Handler}
    end
end
```