RoSe : un framework pour la conception et l'exécution d'applications distribuées dynamiques et hétérogènes

Jonathan Bardin

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Jonathan M. BARDIN

Thèse dirigée par **Philippe LALANDA**
et codirigée par **Clément ESCOFFIER**

préparée au sein du **Laboratoire Informatique de Grenoble**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII)**

RoSe : Un framework pour la conception et l'exécution d'applications distribuées dynamiques et hétérogènes.

Thèse soutenue publiquement le **2 Octobre 2012**,
devant le jury composé de :

Mme Laurence NIGAY
Professeur à L'Université Joseph Fourier, Présidente

M. Iulian NEAMTIU
Assistant Professor University of California, Riverside, Rapporteur

M. Philippe ROOSE
Maître de Conférences à l'Université de Pau, Rapporteur

M. Ye-Qiong SONG
Professeur à l'Université de Lorraine / INPL, Examinateur

M. Philippe LALANDA
Professeur à l'Université Joseph Fourier, Directeur de thèse

M. Clément ESCOFFIER
Head of the Innovation Competence Center à akquinet tech@spree, Co-Directeur de thèse

Résumé

L'adaptation est aujourd'hui devenue un enjeu majeur en Génie Logiciel. Les ingénieurs sont en effet régulièrement confrontés à des demandes d'évolution qui peuvent prendre de nombreuses formes : mises à jour, nouvelles versions, besoins en nouvelles fonctionnalités, etc. Cette tendance est accrue par l'émergence de nouveaux domaines tels que l'informatique ubiquitaire ou le *cloud computing* qui exigent des changements dynamiques dans des environnements en constante évolution. Ainsi, dans ces domaines, les ressources sont souvent élastiques, volatiles et hétérogènes.

Cette thèse s'intéresse en particulier à la conception et à l'exécution d'applications distribuées composées d'entités hétérogènes et qui nécessitent d'être adaptées durant l'exécution. Notre approche s'appuie sur les modèles à composant orientés service et sur les styles d'architectures SOA et REST. Nous proposons un framework, nommé RoSe, qui permet l'import de ressources distantes dans un framework à composant orienté service et l'export de service locaux. RoSe permet aux développeurs et aux administrateurs de gérer la distribution des applications de manière totalement indépendante et dynamique grâce à un langage de configuration et d'une API dite *fluent*. Le framework lui-même est modulaire et flexible et supporte l'ajout et le retrait de composants durant l'exécution.

L'implantation de RoSe est hébergée au sein du projet OW2 Chameleon et est aujourd'hui utilisée dans plusieurs projets industriels et académiques.

Abstract

Adaptation has now become a major challenge in Software Engineering. Engineers are indeed regularly confronted with requests for changes that can take many forms : updates, new versions, new features need etc. This trend is enhanced by the emergence of new areas such as ubiquitous computing or cloud computing that require dynamic changes in rapidly constantly evolving environments. For instance, in these areas, resources are often elastic, volatile and heterogeneous.

This thesis focuses especially in the design and execution of distributed applications composed of heterogeneous entities which need to be adapted at runtime. Our approach is based on service-oriented component models and on the SOA and REST architectural styles. We propose a framework, named RoSe, which enables the import of remote resources in a service-oriented component framework and the export of local services. RoSe allows developers and administrators to manage the distribution of their application in a totally independent and dynamic way thanks to a configuration language and a fluent API. The framework itself is modular, flexible and supports the addition and removal of components during execution.

The implementation of RoSe is hosted by OW2 in the Chameleon project and is now used in several industrial and academic projects.

Remerciements

C'est porté par les soutiens et encouragements de nombreuses personnes que j'ai réalisé mes travaux. Je tiens en premier lieu à remercier les membres du Jury. Je remercie Philippe Roose et Iulian Neamtiu pour avoir rapporté ma thèse. Je remercie également Laurence Nigay d'avoir accepté la présidence du Jury et Ye-Qiong Song pour avoir accepté d'examiner mon travail.

Cette thèse n'aurait jamais pu aboutir sans le soutien, l'encadrement et les encouragements de mes directeurs de thèse Phillipe Ladanda et Clément Escoffier, et ce, malgré des emploies du temps très encombrés et la distance qui parfois nous séparait. Je tiens aussi à remercier Ichiro Satoh qui m'a accueilli au sein du NII à Tokyo pour trois mois, et Iulian Neamtiu qui en plus de rapporter mes travaux m'a accueilli et encadré pendant trois mois à Riverside. Une expérience qui m'a beaucoup apporté, aussi bien sur le plan professionnel que personnel.

Je souhaite aussi remercier l'ensemble des membres de l'équipe ADELE passés et actuels, qui m'ont supporté pendant 5 ans et avec qui j'ai partagé de très bon moments. Pierre, Mehdi, Noé, Diana, Gabriel, Kiev, qui ont été non seulement de très agréable collègue de travail, mais aussi et surtout de très bons amis.

Je ne peu terminer sans un grand merci à Walter et Marcia qui m'ont donné bien plus qu'une collocation, un deuxième foyer. Un grand merci à Lan et Ludo pour leur précieuse amitié. Et bien sûr, un autre immense merci à l'ensemble de ma famille, et en particulier, mes parents et mes sœurs qui ont toujours été là pour moi. Et enfin j'aimerais remercier Mi, qui m'a apporté bien plus qu'elle n'imagine partageant mes joies et mes peines, sans qui ces dernières années auraient été bien rude.

Liste des publications

Les travaux discutés dans cette thèse ont été présentés précédemment.

- fANFARE : Autonomic Framework for Service-based Pervasive Environment, SCC '12 [97],
- Improving user experience by infusing web technologies into desktops, SPLASH '11 [11],
- Towards an Automatic Integration of Heterogeneous Services and Devices, APSCC '10 [10],
- Developing User-Centric Applications with H-Omega, MWMOSA '09 [43].

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

> "When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem." *Edsger W. Dijkstra*

1.1 Introduction

L'informatique est caractérisée aujourd'hui par des évolutions considérables qui s'effectuent à un rythme effréné. La généralisation d'Internet, la prolifération des équipements connectés intelligents, l'émergence d'une intelligence ambiante (*pervasive computing*) et l'apparition du *cloud computing* bouleversent le paysage informatique et soulèvent des problèmes de Génie Logiciel critiques. Ces changements affectent radicalement la manière dont sont conçus, exécutés et administrés les logiciels. Dans cette partie introductive, nous présentons rapidement ces nouvelles tendances, et tentons d'en déduire les défis associés.

L'un des plus grands bouleversements de l'informatique est l'apparition d'Internet. Internet est un média de communication. Il permet la diffusion d'informations à grande échelle. Il est ainsi possible de diffuser une information, une photo ou une vidéo en quelques secondes depuis et vers différents média (téléphone portable, ordinateur, tablette, etc.). Cette facilité pose des problèmes importants pour les logiciels réceptionnant ces informations. Tout d'abord, il s'agit, par nature, d'applications distribuées. Cet aspect oblige

l'application à gérer les problèmes de connectivité, d'utiliser un protocole de communication, d'éviter la perte de donnée... Une deuxième difficulté est la charge qui peut évoluer (augmenter ou baisser) très rapidement. Le stockage et le traitement d'une telle quantité de données (recherche, indexation, conversion) s'avèrent également être des activités très complexes. Nous voyons ainsi apparaître des bases de données contenant des téraoctets d'informations. Le Web, a également introduit un nouveau mode de consommation. Les sites de e-commerce font déjà partie de notre vie quotidienne. La conception de ces sites web n'est pas simple. En plus des problèmes de sécurité, d'interface utilisateur, de gestion en temps réel de stocks, de suivi des colis, il faut également garantir un temps de réponse acceptable et la disponibilité du site. Les concepteurs de ces applications doivent continuellement proposer de nouvelles fonctionnalités afin de satisfaire les attentes grandissantes des utilisateurs. Cette flexibilité change en profondeur la conception des applications web et impacte non seulement l'application, mais aussi les intergiciels utilisés (tel que les serveurs d'applications, les gestionnaires de bases de données, et les gestionnaires de communications réseaux), ainsi que les outils de développement (gestionnaires de sources, outils de construction), et les méthodes de développement (*Agilité*, déploiement continu, développement basé sur les tests). Afin de répondre a toutes ces demandes, de nouvelles technologies apparaissent tel que les bases de données noSQL (*not only SQL*), de nouveaux types de serveurs d'applications, ou encore l'évolution de technologies éprouvées comme Java Enterprise Edition 7. L'un des changements majeurs actuels est le développement d'HTML 5, permettant, entre autre, de développer des interfaces utilisateurs plus réactives, interactives et adaptables.

Le développement de l'Internet est également dû à la popularisation d'appareils connectés comme les *Smartphones*. Le web est accessible à tout moment de n'importe où. Les applications mobiles est généralement composée d'une partie cliente exécutée sur le périphérique mobile, mais aussi d'une partie serveur. Le développement de telles applications reste un challenge et, à cause des spécificités liées aux périphériques mobiles, demande de revoir les méthodes usuelles de construction d'applications dites – client-serveur –. Par exemple, la conception d'interfaces utilisateurs doit prendre en compte des écrans tactiles et de petite taille. De plus, l'architecture de ces applications doit prendre en compte des problématiques telles que la perte de connectivité, l'adaptation et la réduction des données échangées. L'apparition d'applications mobiles proposant un mode « non-connecté » et un

protocole de synchronisation illustre ces impacts architecturaux. La conception d'application mobile est un domaine encore jeune et évolue de manière continue depuis une dizaine d'années. Des nouveautés apparaissent régulièrement comme la géolocalisation et l'utilisation de la voix comme interface utilisateur. Ces évolutions sont poussées non seulement par les éditeurs d'applications, toujours plus innovants, mais aussi par les capacités des appareils.

Bien que les *Smartphones* et les tablettes soient les dispositifs connectés les plus communs, une autre classe d'appareils a fait son apparition. Ces appareils, dit *communicants*, sont contrôlables à distance et peuvent communiquer entre eux. Ainsi, nous avons vu par exemple le développement d'appareils compatibles Bluetooth [1] . D'autres protocoles ont également vu le jour comme UPnP, DPWS et RFID. Ces protocoles ne reposent pas nécessairement sur Internet, et ciblent généralement une communication locale (réseaux domestiques ou d'entreprise). Cependant, l'émergence d'un – Internet des Choses – (*Internet of Things*), un réseaux d'applications web et de dispositifs connectés et communicants, montre un attrait important autour de ces applications. Cet intérêt grandissant est principalement dû à la capacité de contrôler un environnement physique grâce à une interface logicielle et donc à la capacité d'automatiser ce contrôle. Cette automatisation est un enjeu crucial dans le développement de solutions domestiques (pour améliorer le confort ou la sécurité d'un habitat) ou de machine à machine (M2M) comme pour le suivi de marchandises. Ainsi nous voyons émerger une nouvelle classe d'applications reposant à la fois sur du logiciel mais également sur des appareils communicants déployés dans le monde réel. Le développement de ces applications reste un défi majeur. Les méthodologies et les technologies à mettre en place sont complexes. La disponibilité dynamique des dispositifs physiques (pouvant apparaître et disparaître), les communications réseaux instables, les différents protocoles de communication, l'administration de ces systèmes hybrides pouvant atteindre des tailles considérables, la prise en charge du flux de données important et hétérogène, font de la mise en place de ces applications un véritable challenge.

1. Le protocole Bluetooth est un regroupement de plusieurs protocoles qui définissent des règles de très bas niveau comme la bande de fréquences des ondes électromagnétiques, des règles de plus haut niveau comme l'organisation logique d'un réseau en maitres et esclaves, jusqu'à des règles de haut niveau définissant des profils de matériels, la manière de découvrir les dispositifs accessibles, etc.

Afin de résoudre les problèmes de montée en charge et de réduire le cout d'administration des applications, le *Cloud Computing* apparaît comme la solution en vogue. Le *Cloud* consiste a utiliser des ressources mises a disposition par une entreprise disposant de nombreuses machines, comme Amazon, Google, Sales Force ou OVH. Il est ainsi possible de louer des machines virtuelles (CPU, mémoire, stockage) afin d'héberger d'autres applications. Ce type de Cloud appelé *Infrastructure as a Service* permet de réduire les coûts d'achat de machines, tout en proposant une grande flexibilité en permettant de louer d'autres machines pour faire face à une demande accrue. Avec *l'Infrastructure as a Service*, il est aussi possible de louer des environnements d'exécution prêt à l'emploi comme un serveur d'application JavaEE, ou un GoogleApp Engine afin d'exécuter des applications tout en s'abstrayant des couches sous-jacentes (système d'exploitation). L'administration, souvent complexe de ces intergiciels est déléguée. Il est également possible de déployer (dynamiquement) une application sur plusieurs instances d'un environnement afin de mieux supporter la charge. Enfin, il existe un dernier type de *Cloud* appelé *Software as a Service* proposant un service logiciel, évitant ainsi les coûts d'installation et d'administration. Par exemple, il est possible de louer une instance d'un logiciel de compatibilité. Cependant, bien que le *Cloud* soit devenu populaire, et bien qu'il réduise effectivement les coûts d'administration, le passage à l'échelle n'est pas garanti. En effet, il est généralement nécessaire d'avoir conçu son application de manière à utiliser l'élasticité proposée par le *Cloud*. Il est également souvent difficile de laisser gérer des éléments cruciaux d'un système à une entreprise extérieure, et ainsi perdre une partie du contrôle.

Ces différentes évolutions majeures mettent en évidence l'aspect de plus en plus distribué des logiciels modernes. Cela soulève bien évidemment de formidables problèmes de génie logiciel liés en particulier aux aspects suivants :

- Les logiciels aujourd'hui sont rarement conçus et mis en opération de façon isolée. Ils reposent sur des applications logicielles écrites, mises en place et administrées par des parties tierces.
- Les logiciels modernes utilisent des données produites par diverses applications distantes. Ces données sont caractérisées par des tailles, des complexités et des hétérogénéités croissantes.

– Les logiciels sont potentiellement utilisés par une masse d'utilisateurs qui interagissent via des équipements toujours plus nombreux, plus hétérogènes, plus mobiles, etc.

En d'autres termes, la notion de machine d'exécution évolue et se confond de plus en plus avec le réseau, un réseau mondial, global. Ceci soulève de grands problèmes au niveau du développement des applications, mais aussi de leur maintenance.

1.2 Problématique

Au cours des dix dernières années, le domaine informatique a ainsi beaucoup évolué. Comme nous l'avons vu, l'explosion d'Internet, l'apparition des applications mobiles, des appareils communicants, et le développement du *Cloud* sont en train de changer radicalement la manière dont sont conçues, exécutées et administrées les applications logicielles. Ces changements s'accompagnent d'une tendance persistante depuis de nombreuses années : la croissance continue de la taille et de la complexité des logiciels.

A notre sens, trois propriétés majeures caractérisent nombre de logiciels modernes : distribution, dynamisme et hétérogénéité. Plus précisément :

– **Distribution** - les applications modernes sont fréquemment distribuées. Souvent cette distribution est très large puisqu'elle peut aller de capteurs embarqués jusqu'à Internet. Par ailleurs, les architectures de distribution ne suivent pas forcément une topologie *client-serveur* classique. Les patterns utilisés sont souvent complexes, ce qui rend les évolutions et plus généralement la gestion des applications plus délicates.
– **Dynamisme** - ce nouveau défi a un gros impact sur le cycle de vie du logiciel. Celui-ci n'est plus conçu comme une brique monolithique fixe, mais comme un ensemble de briques flexibles, pouvant apparaître, disparaître ou évoluer au cours du temps. Ces briques, en outre, sont de plus en plus distribuées. La complexité engendrée par le dynamisme est grande et s'applique à plusieurs niveaux comme la conception, le développement et l'exécution.
– **Hétérogénéité** - un autre défi est de prendre en compte l'importance de la grande diversité des technologies et des services mis en œuvre au sein d'une application. Un

développeur doit alors comprendre et savoir utiliser un grand nombre de technologies différentes, notamment pour traiter la distribution.

On peut ajouter deux propriétés également très structurantes : la complexité et la disponibilité. La complexité des logiciels est liée aux besoins propres des métiers qu'ils adressent. Un autre facteur est la complexité de la plate-forme technologique. C'est le cas des applications reparties, où la mise en place de services non fonctionnels tels que la sécurité, les transactions et la reprise sur panne s'ajoute à la complexité propre aux domaines applicatifs. Les applications doivent dorénavant être disponibles en permanence et répondre rapidement quel que soit le nombre d'utilisateurs ou la quantité de données à traiter. Bien que cette propriété ne soit pas nouvelle, son importance a grandi. L'élasticité offerte par le *Cloud* est en train de changer la manière dont les applications sont fabriquées pour faire face aux fluctuations de demandes.

Ces problématiques sont étudiées depuis de nombreuses années. Cependant, les récents changements dans le monde informatique réduisent l'efficacité des solutions actuelles, les rendant même obsolètes dans certains cas. De nombreuses technologies et méthodologies ont récemment vu le jour afin de faire face aux nouvelles difficultés liées aux développement d'applications modernes, apportant ainsi une complexité supplémentaire et rendant les environnements logiciels encore plus hétérogènes. Par exemple, il n'est plus rare de voir des applications utilisant à la fois des bases de données relationnelles (SQL) et des bases de données *noSQL*. La mise en place d'environnements d'exécution adaptés aux applications modernes de manière efficace est aujourd'hui un problème crucial.

Cette thèse s'intéresse à la gestion de la distribution dans les applications dynamiques et hétérogènes. Plus précisément, elle cherche à faciliter, voire rendre transparent dans certains cas, la gestion de la distribution pour des applications requérant et exportant des ressources hétérogènes et évoluant en cours d'exécution de façon imprévisible.

Concrètement, l'objectif de cette thèse est de fournir un cadre conceptuel et programmatique pour le développement d'applications distribuées, dynamiques et hétérogènes. L'informatique ubiquitaire constitue un cadre d'application privilégié puisque ce domaine se caractérise par un dynamisme et une hétérogénéité importants. Mais l'informatique ubiquitaire n'est pas le seul cadre applicatif, comme nous le verrons tout au long de ce

document. Les applications Internet, en particulier, sont également fortement considérées.

Un des principes fondamentaux de notre travail est de séparer clairement le code applicatif du code lié à la distribution. Pour atteindre nos objectifs, nous proposons d'introduire un nouveau patron de conception architecturale et un framework associé, nommé RoSe.

Notre travail propose en particulier :
- Un cadre de développement permettant de spécifier de façon très simple la publication et la consommation de ressources logicielles distribuées de façon dynamique,
- Un environnement d'exécution gérant de manière transparente le code de distribution lié à des ressources dynamiques et hétérogènes (et distantes),
- Des mécanismes d'introspection et de reconfiguration afin de comprendre et manipuler à chaud l'architecture de ces applications distribuées lors de leur exécution.

Le framework associé au patron de conception architecturale, RoSe, est disponible en open source sur le projet OW2 Chameleon (`chameleon.ow2.org`). Il est aujourd'hui utilisé dans plusieurs projets industriels et académiques. En particulier, RoSe a été mis en œuvre au sein d'Orange, de Schneider Electric et d'Akquinet.

1.3 Plan du document

Après cette introduction, le manuscrit de thèse est divisé en deux grandes parties : un état de l'art et la présentation de notre contribution.

L'état de l'art comprend deux chapitres :

- le chapitre 2 présente la notion d'architecture logicielle. Après un certain nombre de définitions, nous présentons les défis associés à ce domaine et les patrons architecturaux les plus communément utilisés aujourd'hui. Nous nous concentrons ensuite sur les patrons propres aux architectures distribuées faiblement couplées qui constituent le cœur de cette thèse. Nous détaillons en particulier les architectures fondées sur RPC, CORBA, les services WEB et REST. Nous développons également l'idée que ces approches ne traitent qu'imparfaitement les nouvelles applications logicielles mentionnées précédemment.

– le chapitre 3 traite de la dynamicité en informatique. Après avoir défini cette notion fondamentale, nous décrivons les moyens de mettre en œuvre la dynamicité en logiciel en nous positionnant à différents niveaux de granularité. Nous nous attardons particulièrement sur les frameworks OSGi et iPOJO qui permettent, dans le monde Java, de concevoir des applications dynamiques à un niveau – composant –.

La contribution se divise quant à elle en trois parties :

– le chapitre 4 rappelle la problématique, expose les objectifs et donne une vision d'ensemble de notre approche. Nous introduisons le patron proposé dans cette thèse et le framework associé, appelé RoSe. RoSe permet d'automatiser l'intégration et publication de ressources distantes, dynamiques et hétérogènes (au niveau des protocoles en particulier).
– le chapitre 5 détaille l'implémentation de Rose. Il s'agit plus précisément de présenter l'architecture globale et de détailler l'implantation des composants majeurs. Ces composants sont développés au dessus des technologies OSGi/iPOJO. Dans son état actuel, RoSe peut être intégré avec du code applicatif développé en iPOJO (Java/OSGi) et H-ubu (JavaScript).
– le chapitre 6 traite de la validation du framework. Dans un premier temps nous évaluons le coût d'utilisation de RoSe durant la phase de développement, puis nous évaluons les performances durant l'exécution. Enfin, nous présentons plusieurs applications qui s'appuient sur RoSe pour la gestion de la distribution dans les domaines de l'informatique ubiquitaire, en particulier la domotique, ainsi que dans le domaine des applications Web.

Enfin, le chapitre 7 synthétise les idées principales de notre proposition. Nous rappelons à cet effet les points principaux de notre contribution et nous décrivons les perspectives envisagées pour de futurs travaux.

Chapitre 2

Architectures logicielles distribuées

"On croit que le style est une façon compliquée de dire des choses simples, alors que c'est une façon simple de dire des choses compliquées." *Jean Cocteau*

2.1 Architecture logicielle

2.1.1 Origines

L'architecture peut se définir comme l'art et la science de bâtir des édifices. Dans son traité *De Architectura*, Vitruve, architecte romain du premier siècle av. J.-C, décrit l'architecture de la façon suivante [148] :

> "L'architecture est une science qui embrasse une grande variété d'études et de connaissances ; elle connaît et juge de toutes les productions des autres arts. Elle est le fruit de la pratique et de la théorie. La pratique est la conception même, continuée et travaillée par l'exercice, qui se réalise par l'acte donnant à la matière destinée à un ouvrage quelconque, la forme que présente un dessin. La théorie, au contraire, consiste à démontrer, à expliquer la justesse, la convenance des proportions des objets travaillés."

Vitruve énonce également trois principes fondamentaux qui restent encore très pertinents aujourd'hui pour la construction d'architectures, à savoir la beauté, la solidité et l'utilité :

– Une architecture doit répondre à des critères esthétiques basés sur la symétrie, l'élégance des proportions et l'adéquation avec la fonction de l'édifice,

- Une architecture doit être pérenne et reposer sur des fondations solides et des matériaux de qualité,
- Une architecture doit organiser un édifice de telle sorte qu'on puisse en disposer de manière aisée, en distribuant chaque chose d'une manière commode et pertinente.

Ces principes sont encore très pertinents aujourd'hui et ont influencé de nombreux architectes au cours du temps. Par exemple, des architectes tels que Pier Luigi Nervi [108] se sont appropriés ces principes en leur donnant une forme plus actuelle. Pour ce dernier, le principe d'utilité devient la fonction, la solidité devient la structure et la beauté devient la forme. Alexander [2], architecte américain du vingtième siècle, a défini des modèles d'architectures composables permettant d'atteindre de façon quasi automatique ces propriétés de beauté, de solidité et d'utilité. Même si ce travail a été beaucoup raillé à l'époque à cause de l'uniformité résultante, il a fortement influencé des architectes modernes mais aussi le domaine de l'informatique. Nous y reviendrons.

L'architecture est ainsi une science majeure permettant d'organiser des systèmes complexes tout en atteignant des propriétés de beauté, de solidité et d'utilité. Ces propriétés sont difficilement quantifiables : elles sont souvent très subjectives et sont jugées différemment en fonction des époques. Rappelons la protestation signée de la majorité des artistes et intellectuels de l'époque à propos de la Tour Eiffel (Protestation contre la tour de M. Eiffel) :

> "Nous venons, écrivains, peintres, sculpteurs, architectes, amateurs passionnés de la beauté jusqu'ici intacte de Paris, protester de toutes nos forces, de toute notre indignation, au nom du goût français méconnu, au nom de l'art et de l'histoire française menacés, contre l'érection, en plein cœur de notre capitale, de l'inutile et monstrueuse tour Eiffel que la malignité publique, souvent empreinte de bon sens et d'esprit de justice a déjà baptisée du nom de Tour de Babel."

Aujourd'hui la notion d'architecture s'étend à l'édification de systèmes complexes en général. Notamment, elle joue désormais un rôle majeur en informatique. C'est ce que nous examinons dans les sections suivantes. Nous verrons que les notions de structure, de beauté, d'utilité, de solidité, de modèles réutilisables, et de subjectivité y prennent un sens très réel.

2.1.2 Définition de la notion d'architecture logicielle

Les chercheurs n'hésitent plus à faire des analogies entre la structure des systèmes virtuels et l'architecture très concrète des édifices qui nous entourent. Bien que cette analogie ait certaines limites, il apparaît que la définition de l'architecture et les principes fondamentaux présentés ci-avant s'appliquent parfaitement à la structure des logiciels. En effet, de la même façon que la création d'un bâtiment commence par la définition de son architecture, tout logiciel important par sa taille ou sa complexité requiert une étude de son architecture en premier lieu [14]. Comme l'a fait remarquer Grady Booch, on ne construit pas un gratte-ciel de manière *ad hoc* comme on le ferait pour la niche d'un chien [1].

L'apparition de la notion d'architecture en logiciel est toutefois assez récente. En fait, c'est l'augmentation continue de la complexité des logiciels qui a rendu nécessaire ce niveau d'abstraction [62] :

> "as the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation : designing and specifying the overall system structure emerges as a new kind of problem… This is the software architecture level of design."

Cependant, il n'existe pas de réel consensus sur la définition de l'architecture logicielle. De nombreuses définitions ont été avancées pour caractériser une architecture logicielle. L'une des premières a été apportée par Mary Shaw et David Garlan [130] :

> "Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns."

Cette définition présente une architecture logicielle comme la description abstraite des éléments constituant un système logiciel, de leurs interactions, des patrons de conception utilisés, ainsi que des contraintes associées. La notion de patron est aussi centrale dans la définition de Grady Booch [22], mais ce dernier retient plutôt le terme de style d'architecture plutôt que patron.

Franck Buschmann fournit la définition suivante [27] :

1. La version 7.2 du système d'exploitation Red Hat Linux 7.1 à été estimé par David A Wheeler à 8000 homme/ans (`http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`), pour comparaison l'édification de l'empire state building nécessita 3500 homme/ans

> "Software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and nonfunctional properties of a software system."

Dans cette définition, l'architecture décrit également la structure d'un système, mais, cette fois, sous la forme explicite de sous-systèmes et de composants. La structure est définie suivant différentes vues, incluant des aspects fonctionnels et non fonctionnels. Les propriétés du système peuvent être introduites sous la forme d'attributs. Ces attributs, appelés propriétés externes dans d'autres définitions [14], sont pris en compte et classés en deux catégories. Certains types d'attributs tels que la performance, la sécurité, la disponibilité, la fonctionnalité et l'utilité sont observables lors de l'exécution. D'autres, au contraire, sont invisibles au cours de l'exécution de systèmes logiciels (tels que la modifiabilité, la portabilité, la réutilisabilité, l'intégrabilité et la testabilité) et s'adressent plus aux développeurs ou ingénieurs chargés de la maintenance.

Dwayne Perry et Alexander Wolf [119] définissent une architecture logicielle comme un triplet *{Elements, Form, Rationale}*. Ils présentent donc l'architecture logicielle comme un ensemble d'éléments de conception agencés suivant une forme particulière. Le terme *rationale* correspond au *pourquoi* de l'architecture : il regroupe l'ensemble des motivations qui ont dirigé le choix de structuration. Perry et Wolf distinguent trois différents types d'éléments architecturaux : les éléments de traitement, les éléments de données et les éléments connecteurs. Les éléments de données contiennent l'information utilisée et transformée par les éléments de traitements. Les connecteurs sont les liens qui permettent aux éléments de communiquer entre eux. Ces types éléments sont souvent conceptualisés en deux concepts architecturaux fondamentaux, les composant et connecteurs, aux dépens des éléments de données.

Bass, Clements, et Kazman proposent la définition suivante [14] :

> "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. By 'externally visible' properties, we are referring to those assumptions

> other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. The intent of this definition is that a software architecture must abstract away some information from the system (otherwise there is no point looking at the architecture, we are simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and risk reduction."

Ici, l'architecture logicielle est présentée comme la structure d'un système informatique. C'est une structure de haut niveau définie comme un assemblage de composants logiciels, les propriétés de ces composants et les relations entre composants. L'attention est portée sur la difficulté à définir le niveau d'abstraction que doit permettre l'architecture ; elle doit être suffisamment abstraite pour permettre une vue d'ensemble du système mais suffisamment précise pour permettre l'analyse du système, la prise de décision et par conséquent de réduire les risques.

L'organisation IEEE fournit une définition proche [79] en ajoutant la notion d'évolution :

> "Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution."

Malgré l'absence de consensus et de légères divergences lexicales, toutes ces définitions semblent découler de notions et concepts communs. La recherche en architecture logicielle est un champ bien établi [128] et, même si les définitions évoluent aux fils des avancées du domaine, toutes s'appuient sur les mêmes fondements. M. Shaw affirme d'ailleurs que l'architecture logicielle est aujourd'hui arrivée à son âge d'or [128] et est au cœur de la conception du système logiciel, au cœur de son développement et de plus en plus au cœur de son exécution [12].

Pour résumer, une architecture logicielle est une abstraction structurelle qui permet de raisonner sur la construction et l'évolution d'un système informatique. Elle définit l'organisation d'un système sous la forme d'un assemblage de composants de traitement, de composants de données et de connecteurs. Les composants encapsulent les fonctionnalités ou des données cohérentes et les connecteurs représentent les liaisons entre les composants.

Une architecture inclut les contraintes et principes sous-jacents à sa construction de façon à permettre sa compréhension et son évolution. Enfin, une architecture peut être définie en suivant des principes et des contraintes spécifiés par un style ou patron d'architecture particulier. Nous abordons ce dernier point dans la suite de ce chapitre.

2.1.3 Défis liés aux architectures logicielles

Comme avancé précédemment, certains auteurs considèrent que l'architecture logicielle est aujourd'hui dans son *âge d'or*. Néanmoins, si l'importance de l'architecture est désormais bien reconnue, la communauté s'accorde sur le fait que des problèmes majeurs et complexe sont encore à traiter dans ce domaine. Attardons-nous sur trois défis majeurs :

- La description des architectures,
- L'évaluation des architectures,
- La conception des architectures.

Les deux premiers points ont donné lieu à de nombreuses recherches sur les ADL, c'est-à-dire les Langages de Description d'Architecture. Un ADL peut être informel (utilisation de schémas par exemple), semi-formel (en UML par exemple) ou formel. Un langage de description d'architecture fournit les éléments pour représenter une ou plusieurs vues architecturale se focalisant sur une préoccupation particulière. De plus, la description d'une architecture à l'aide d'un ADL peut être compréhensible par une machine, ou transformable en un format compréhensible. Les ADLs ont pour but de favoriser la compréhension des architectures, leur dissémination mais aussi leur évaluation. Une description peut ainsi être analysée et utilisée afin d'automatiser l'implémentation de certains aspects.

Par ailleurs, chaque langage de description d'architecture se distingue par ses capacités de modélisation qui proviennent directement des objectifs poursuivis par cet ADL [100]. Par exemple, Aesop [59] spécifie les architectures d'applications en se basant sur un ensemble de styles architecturaux. SADL [104] est un langage de description d'architecture structural. Il se concentre sur le raffinement des architectures du système sur différentes couches d'abstraction de façon formelle. Cela permet la traçabilité des variantes et des évolutions du système. UniCon [129] permet de générer du code de liaison afin de construire et vérifier les architectures à partir d'éléments architecturaux prédéfinis utilisant les protocoles communs. MetaH [145] se concentre sur la conception, la vérification et la validation

de l'assemblage et du déploiement de systèmes temps réel critique. Il est utilisé pour modéliser les architectures en particulier dans le domaine de la navigation et du contrôle.

Certains langages de description d'architecture permettent de décrire des interactions collaboratives entre les éléments structuraux du système à l'exécution. En d'autres termes, ils permettent de modéliser, simuler et analyser le comportement dynamique d'un système. C2, Darwin, Rapide et Wright constituent des exemples de tels ADLs. C2 [139] se focalise sur la description des architectures d'application réparties et dynamiques. Darwin [96] traite de la configuration et de l'instanciation des systèmes distribués et dynamique de façon formelle. Rapide [95] est un langage permettant de simuler le comportement dynamique à l'aide de poset (partial ordered event) [2]. Il permet de vérifier des propriétés telles que la synchronisation et la concurrence sur des assemblages de composants. Comme Rapide, Wright [4] est utilisé pour modéliser et analyser formellement le comportement dynamique des systèmes concurrents. Mais, Wright se concentre plus sur la vérification de conformité de l'assemblage des éléments architecturaux et la détection d'inter-blocages des systèmes concurrents au cours de l'exécution. Il emploie le concept de CSP (*Communicating Sequential Process*) [75] pour décrire l'architecture du système logiciel.

Un constat s'impose par rapport à ces travaux sur les ADLs : ils n'ont pas percés et ne sont pas utilisés du tout au delà de quelques laboratoires. De nombreuses avancées sont encore nécessaires pour une modélisation effective des architectures. Le troisième défi mentionné ici est lié à l'activité de conception des architectures logicielles. La tâche de conception reste en effet très complexe et soulève de nombreux problèmes. Il est en fait très urgent de faire émerger des solutions pragmatiques et *scalables* (capables de passer à l'échelle) facilitant le travail des architectes (et concepteurs au sens large).

Un architecte doit produire une architecture répondant aux exigences du projet considéré tout en assurant un ensemble de qualités logicielles fondamentales. En particulier, une architecture doit respecter les propriétés suivantes :

- Abstraction,
- Modularité,
- Cohérence,

2. `http://en.wikipedia.org/wiki/Partially_ordered_set`

– Couplage,
– Evolutivité.

Par nature, une architecture est une abstraction. Elle définit un ensemble de composants au travers d'interfaces, de propriétés et de documentations diverses. Ces éléments représentent une enveloppe, un cahier des charges qui doit ensuite être implanté. Il en va de même pour les connexions qui sont définies de façon abstraite. L'abstraction est un élément fondamental : il donne la latitude nécessaire aux concepteurs/développeurs d'implanter un système dans le langage cible et sur les plates-formes cibles tout en disposant d'un canevas de travail.

La modularité d'une architecture repose sur la notion de composant. Cette notion, si correctement appliquée, permet de mettre en œuvre le principe d'encapsulation et de séparer les notions de traitement et de communication. Les composants néanmoins visent la décomposition structurelle et ne traitent pas de façon explicite les autres types de décompositions. Parnas [116] a indiqué il y a déjà longtemps qu'un logiciel est caractérisé par de nombreuses dimensions (dont certaines non fonctionnelles comme la sécurité par exemple). La prise en compte des différentes propriétés non fonctionnelles peut rapidement alourdir les composants au travers de la définition de nombreuses interfaces difficiles à gérer. Une bonne modularité est donc très dépendante du problème abordé et difficilement réutilisable : idéalement, elle doit répondre uniquement au problème considéré !

Ceci nous amène de façon naturelle vers la notion de cohérence. Un défi majeur de toute conception est de créer des artefacts cohérents, regroupant des concepts sémantiquement proches dans le contexte considéré. Ceci permet de limiter les communications entre composants et de favoriser l'évolution en limitant les effets de bord lors de modifications. Ici aussi, la cohérence des artéfacts dépend fortement du problème abordé et est difficilement réutilisable.

Le couplage est complémentaire à la cohérence. Il caractérise la force des liens entre les artefacts logiciels, les composants dans le cas d'une architecture logicielle, ainsi que la nature des liens et leurs nombre. Il est impératif de maîtriser le nombre et la complexité de ces liens. Ceci permet une meilleure lisibilité, ou compréhension, du code et favorise l'évolution. Des liens de couplage bien maîtrisés limitent la propagation des modifications

lorsqu'on doit traiter une évolution.

Enfin, un défi majeur pour tous les architectes est l'évolutivité des logiciels. Bien sûr, il n'est pas possible de structurer un système de façon à prévoir et préparer tous les changements possibles. Mais, il est également acquis que de nombreux systèmes actuels sont amenés à s'adapter fréquemment à de nouveaux contextes ou de nouveaux besoins utilisateur. Dans certains cas, ces évolutions doivent se faire de façon dynamique, c'est-à-dire lors de l'exécution d'un système logiciel. Dans ces cas là, des mécanismes permettant l'évolution sont nécessaires.

2.1.4 Styles d'architecture

Pour aider la description mais aussi la conception des architectures, la notion de style architectural a été introduite [63]. Cette notion reprend la notion de pattern, introduite par Gamma [57] dans le contexte de la programmation orientée objet(OOP), s'inspirant lui-même des travaux de Alexander [3], architecte précédemment mentionné. Les styles architecturaux sont des techniques facilitant l'identification et la définition des composants et des connecteurs. Ils constituent un des piliers de l'architecture logicielle moderne [60].

Plus précisément, un style d'architecture est défini comme suit par le SEI [3] :

> "A specialization of element and relation types, together with a set of constraints on how they can be used."

Cette définition exprime clairement le fait qu'un style architectural spécialise les notions de composants et de connecteurs et ajoute des contraintes d'utilisation. Un style délimite la manière dont les composants et les connecteurs interagissent de façon à atteindre certaines propriétés telles que l'évolutivité, la réutilisation ou la performance. Un style architectural est ainsi une spécification générique pouvant s'appliquer à différents systèmes possédant des caractéristiques communes. Il permet de décrire les contraintes des règles/conditions topologiques en termes de structures des systèmes. Chaque style architectural synthétise ainsi un ensemble de décisions de conception cohérentes. Ils sont formulés et développés de façon à réutiliser facilement des conceptions logicielles éprouvées. Un patron de conception fournit un ensemble de modèles décrivant de façon plus ou moins formelle les

3. Software Engineering Institute - Carnegie Mellon University

solutions correspondant à un ensemble de problèmes bien définis dans un contexte donné. Les styles architecturaux ont été utilisés avec succès depuis plusieurs années et sont entrés dans le bagage commun des architectes logiciels [26].

Les styles architecturaux sont considérés comme la dénotation des langages de modélisation des systèmes logiciels [80]. En outre, les styles architecturaux communs peuvent être classifiés simplement en fonction des types de systèmes logiciels, tels que les systèmes de flux de données, les systèmes d'appels et de retours, et les systèmes se basant sur un courtier.

Dans [63], les auteurs ont défini un ensemble de styles génériques généralement axé sur les modes de communication (client/server, orienté messages, *publish/subscribe*). Certains styles abordent la structuration comme le style en couche ou le style *pipe-and-filter*. Ce dernier style, par exemple, est utilisé dans le cas où de nombreuses transformations de données sont nécessaires et, éventuellement, lorsque ces transformations doivent pouvoir être agencées de façon flexible. Ce style qui permet la création de flots de traitement de données facilite la mise en œuvre des opérations d'ordonnancement, de synchronisation, et de communications nécessaires à la mise en place de transformations successives de données.

Ces styles architecturaux très génériques définissent un ensemble de principes fondamentaux. Ils restent néanmoins très généraux et, finalement, nécessitent d'être raffinés dans la plupart des applications réelles. Pour cette raison, des styles plus spécifiques, associés à des domaines plus limités, ont été définis [46]. Ils ont reçu un accueil enthousiaste et sont largement utilisés dans les communautés concernées [66].

Dans certains cas, les styles architecturaux viennent avec des *frameworks* , c'est-à-dire du code implantant les concepts du style. Les frameworks permettent de basculer de façon plus rapide et plus sûre vers la phase d'implantation.

La notion de framework a été particulièrement travaillée par la communauté objet. Joseph W.Yoder propose la définition suivante [157] :

> "A framework is a reusable design of an application or subsystem and is represented as a set of abstract classes and the way their instances collaborate. Some frameworks are more technology frameworks (vertical) solving problems

> such as GUI's and object persistence, where other frameworks are more domain specific (horizontal) solving say manufacturing problems. Frameworks prescribe how to decompose a problem. There are white-box frameworks where you need to overwrite behavior or subclass in order to get them to work, and there are black-box frameworks where you customize by configuring which happens through polymorphism and parameterization."

Cette définition réaffirme l'idée de *réutilisation de conception* qui est à la base des notions de patrons, motifs et de styles. On retrouve également dans cette définition la distinction usuelle entre les styles génériques (appelés verticaux ici) et les styles centrés sur un domaine (dénommés horizontaux dans cette définition).

Un framework peut être assimilé à une architecture de référence dans un domaine donné. Il fournit le cadre de développement des composants réutilisable et garantit leur incorporation appropriée. C'est en quelque sorte un guide d'assemblage des artéfacts réutilisables pendant la phase de conception et de développement d'un logiciel. De ce fait, les frameworks permettent la construction de logiciels individuels, c'est-à-dire répondant à un besoin particulier, par la réutilisation d'artéfacts prévus à cet effet.

Des frameworks ont été développés pour les styles les plus populaires représentant des enjeux industriels. Par exemple, JavaEE [111] et CORBA [110] sont deux styles architecturaux à composants largement utilisés dans l'industrie qui ont donné lieu à de nombreux frameworks. JavaEE fournit un framework d'intergiciels supportant l'implémentation d'applications d'entreprises. CORBA se focalise plus sur l'interopérabilité des éléments d'applications se basant sur l'architecture orientée objets. Le framework JavaEE ou le framework CORBA permettent aux développeurs de se détacher des détails du développement des protocoles de communications et du développement des parties techniques. La mise en œuvre d'applications se conformant aux deux frameworks est semi-automatique. En d'autres termes, la programmation de la partie technique des applications est produite automatiquement à l'aide des plates-formes associées.

Il faut néanmoins noter que la mise en place d'un framework demande un investissement initial important. Tout d'abord, l'identification des points de variabilité est difficile et coûteux. Cela demande une analyse minutieuse des différentes déclinaisons possibles

d'un artéfact et de maîtriser les techniques de modélisation et de codage permettant d'introduire un certain niveau de variabilité. Cela ne peut être réalisé que par des ingénieurs et architectes ayant des connaissances très larges. Par ailleurs, la phase de maintenance est rendue plus difficile. Il ne s'agit plus de gérer les versions successives d'un même produit mais, maintenant, de gérer de façon conjointe l'évolution de plusieurs produits similaires et de leurs artéfacts réutilisables. Il faut ainsi gérer l'évolution des éléments du domaine ainsi que les liens de dépendances avec les différents produits déjà construits.

2.2 Architecture distribuées faiblement couplées

2.2.1 Définition

Les architectures distribuées forment un sous ensemble des architectures logicielles. Elles ne forment pas un style, néanmoins, car elles demeurent trop diverses. Les notions d'architectures distribuées, et de systèmes distribués, ne sont pas nouvelles en logiciel et il existe de nombreuses définitions. L'une des plus connues est fournie par Tanenbaum [138] :

> "A distributed system is a collection of independent computers that appears to its users as a single coherent system."

Cette définition, courte et élégante, met clairement en avant la notion d'utilisateurs. Un système distribué est ainsi défini comme un ensemble indéterminé de machines d'exécution qui apparaît aux utilisateurs comme un système unique et cohérent. Bien sûr, les différentes machines jouissent d'une certaine autonomie mais elles collaborent de façon à donner l'illusion de l'unicité . Cette idée de collaboration est au cœur des systèmes distribués. Cette définition soulève néanmoins le problème de la transparence. En effet, elle impose à un système distribué de rendre les problématiques de distribution transparente afin d'apparaitre aux utilisateurs comme un système unique et cohérent. Cet objectif de transparence soulève de réels problèmes. D'une part, elle est souvent irréalisable en pratique. D'autre part, il ne facilite pas le travail des programmeurs. Il est nécessaire de fournir des mécanismes ou des outils permettant aux programmeurs d'applications distribuées de traiter la nature distribuée de leurs applications de façon explicite.

Une seconde définition est proposée par Coulouris et ses collègues [31] :

> "We define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages."

Cette proposition est assurément beaucoup plus technique que la définition précédente. Elle introduit la notion générique de composant qui peut représenter aussi bien des éléments logiciels que des éléments matériels. Elle met également en avant la notion de collaboration entre ces différents éléments. Les collaborations entre composants apparaissent comme le résultat de communications et de coordinations basées sur l'unique principe d'échange de messages. Cette définition précise également que les composants, logiciels ou matériels, appartiennent à un même réseau informatique.

David Peleg donne quant à lui une définition plus détaillée que les précédentes. Il place la notion de système distribué dans une perspective plus large [118] :

> "In order to define a distributed system, one safe approach is to consider first what a centralized system is. We usually think of such a system as composed of a single controlling unit. Thus in a centralized system, at most one autonomous activity is carried on at any single moment. In contrast, the main characteristic of a distributed system is that there may be autonomous processors active in the system at any moment. These processors may need to communicate with each other in order to coordinate their actions and achieve a reasonable level of cooperation, but there are nonetheless capable of operating by themselves.
>
> Another significant characteristic of a distributed system is that is it rather non uniform. [...] The processors are often rather different in size and power, architecture, organizational membership and so on."

Peleg définit les systèmes distribués en les opposant aux systèmes centralisés. Selon lui, un système centralisé est constitué d'une entité de contrôle unique et est donc destiné à ne réaliser au plus qu'une activité autonome à un instant donné. Par contraste, un système distribué comprend plusieurs processeurs autonomes actifs au même moment. Ces différents processeurs peuvent communiquer et parvenir à un certain niveau de coordination, mais qui n'en restent pas moins autonomes. La seconde caractéristique relevée par Peleg est l'aspect non uniforme des systèmes distribués. En effet, les processeurs qui constituent un système distribué sont souvent hétérogènes ; ils diffèrent en taille, puissance,

architecture etc...

Ces différentes définitions établissent la multiplicité des processeurs au sein d'un système distribué et introduisent les notions de collaboration (coordination / synchronisation) entre ces processeurs. Le couplage entre ces processeurs est d'ailleurs un point fondamental. Comme nous l'avons déjà vu, le couplage est une mesure qualitative qui repose sur le nombre de connexions entre les processeurs et la nature de ces connexions. Le couplage influence grandement les propriétés logicielles d'un système logiciel comme l'évolutivité. Ceci reste bien sûr valable pour les systèmes distribués.

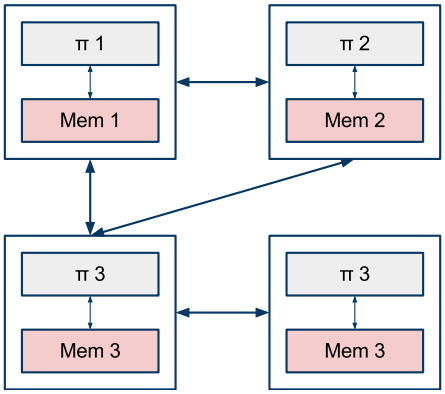


FIGURE 2.1 – Systèmes distribués faiblement couplé.

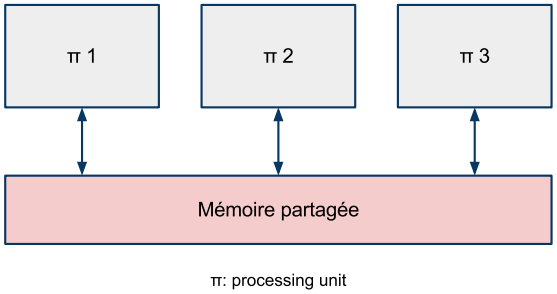


FIGURE 2.2 – Systèmes distribués fortement couplé (système parallèle).

Les systèmes distribués se caractérisent par différents niveaux de couplage. On peut mentionner ici deux formes de couplage extrêmes que l'on rencontre fréquemment et qui sont illustrés par les figures 2.1 et 2.2 :

- Certains systèmes distribués se caractérisent par un ensemble de processeurs très indépendants (i.e. pas de mémoire partagée), communiquant de façon peu fréquente et requièrent peu ou pas de synchronisation (figure 2.1). Ces systèmes sont très faiblement couplés.
- Certains systèmes distribués se caractérisent par un ensemble de processeurs fonctionnant en étroite synchronisation, partageant une large zone de mémoire et disposent de mécanismes de communication extrêmement fiables et véloces (figure 2.2). Ces systèmes sont très fortement couplés.

Clairement, les systèmes faiblement couplés sont souvent plus difficiles à synchroniser mais offrent des possibilités d'évolution beaucoup plus importantes. Ils sont privilégiés en l'absence de contraintes de performance fortes (sur la latence en particulier). Dans la suite de ce mémoire, nous nous concentrons sur les systèmes distribués faiblement couplés.

2.2.2 Remote Procedure Call

Aux premières heures de l'informatique distribuée, la conception et la construction de programmes distribués étaient une tâche extrêmement complexe, et seul un petit groupe d'experts était en mesure de relever un tel défi. C'est dans cette optique que la notion de *Remote Procedure Call* (RPC) a été créée. RPC est en effet né du besoin impératif de simplifier la création d'applications distribuées [17].

Néanmoins, cette exigence de simplicité d'utilisation fortement liée à RPC ne peut pas être dissociée de la nécessité de garantir des communications performantes et sécurisées. Ces trois objectifs ont ainsi dirigé la conception et les premières implémentations de l'approche RPC. Dans cette section, nous présentons les principes fondamentaux de l'approche et la façon dont ils sont mis en œuvre dans les systèmes d'aujourd'hui.

L'idée de base de l'approche RPC est très simple et a été clairement spécifiée par Birrell et Nelson [17]. RPC repose sur la notion de procédure, introduite dans les langages procéduraux [1], qui permet de réaliser des transferts de contrôle et de données au sein d'un même programme. Birrell et Nelson constatent que les mécanismes utilisés pour les appels de procédures en local, c'est-à-dire sur une même machine, sont parfaitement connus et maîtrisés par les programmeurs. Ils proposent alors d'étendre ce principe et les mécanismes sous-jacents afin de permettre le transfert de contrôle et de données aux travers d'un réseau, c'est-à-dire entre des programmes s'exécutant sur des machines indépendantes (des processeurs) peuplant un même réseau.

Ainsi, un des principes fondamentaux de la conception de RPC est que la sémantique des appels de procédures distantes (*remote procedure calls*) doit être aussi proche que possible de la sémantique des appels de procédures locales de façon à donner l'illusion de l'unicité. De cette manière, RPC garantit aux programmeurs d'applications distribuées la

même simplicité d'utilisation que celle atteinte par les programmeurs d'applications centralisées.

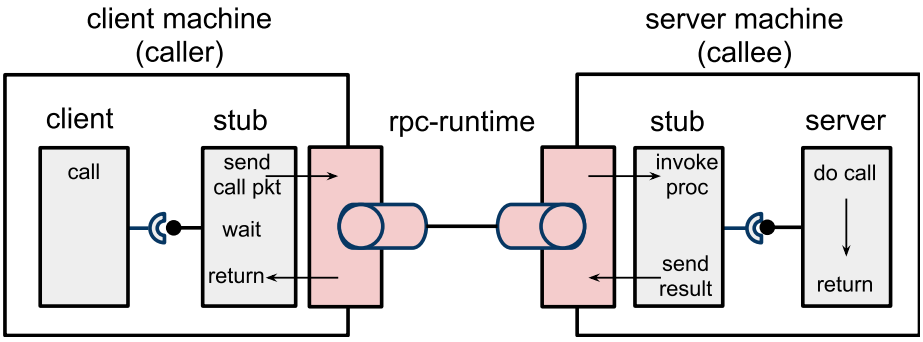


FIGURE 2.3 – Principes de RPC.

Comme explicité par la figure 2.3, le modèle de programmation RPC s'articule autour de cinq composants :

- le client qui représente le programme appelant,
- le *stub* coté client qui assure le lien vers le *runtime* défini ci-après,
- le *runtime* RPC qui assure le transfert de contrôle et de données,
- le *stub* coté serveur qui assure le lien entre runtime et serveur,
- le serveur qui représente le programme appelé.

La dynamique de l'approche RPC est la suivante :

1. le programme client effectue un appel de procédure locale sur le *stub* coté client et suspend ses traitements en attente de la réponse,
2. le *stub* coté client sérialise les paramètres en un ou plusieurs paquets et demande au *runtime* RPC de les transmettre au serveur,
3. le *stub* coté serveur dé-sérialise les données et appelle la procédure appropriée sur le serveur,
4. le *stub* coté serveur reçoit le résultat de la procédure appelée et, après l'avoir sérialisé, demande au *runtime* RPC de transmettre le résultat au client appelant,
5. le *stub* coté client dé-sérialise le résultat et le transmet au client. Le processus suspendu en attente du retour reprend son exécution.

Dans l'approche RPC et la plupart des implantations existantes, les clients et les serveurs font partie intégrante de l'application distribuée et sont développés par les programmeurs. En revanche, les *stubs* client et serveur sont générés par un programme utilitaire. La génération est rendue possible par l'utilisation d'interface permettant de spécifier le nom des procédures ainsi que le type de leurs arguments et valeurs de retour. Ces informations sont suffisantes pour permettre de compiler indépendamment les programmes serveur et client ainsi que de générer proprement la séquence d'appels. Néanmoins, une contrainte forte est que les données qui transitent entre les appels doivent être sérialisables.

On dit d'un module/composant implémentant une telle interface qu'il exporte cette interface. Réciproquement on dit d'un module/composant appelant des procédures depuis l'interface qu'il importe l'interface. En s'affranchissant de la création des stubs, le programmeur n'a pas besoin de coder la partie communication distante de l'application. Une fois l'interface spécifiée, le programmeur doit seulement implémenter le composant client qui importe l'interface et le composant serveur qui exporte l'interface. Ensuite, les stubs sont générés de manière statique. Par convention le *client-stub exporte* l'interface et le *server-stub importe* l'interface.

RPC a donné lieu et donne encore lieu à de nombreuses implantations sous forme de frameworks, encore appelés middleware. Tout d'abord, RPC a servi de fondement à la création des premiers systèmes de fichiers distribués et localisation agnostique (ex. NFS [126] et AndrewFS [76]). Dans le système de fichier Andrew [76], l'implémentation de RPC est en dehors du noyau du système et peut ainsi supporter plusieurs milliers de clients par serveur, le passage à l'échelle étant rendu possible par l'utilisation de caches et de services de résolution de nom. La partie communication de ces systèmes est basée sur les paradigmes RPC mais utilisent leur propre protocole optimisé mais fondamentalement incompatible avec les autres implémentations. Aujourd'hui, de nouvelles implémentations de middleware de programmation basés sur RPC sont encore proposées, en particulier dans le domaine des grilles de calculs [28, 137] ou encore dans les systèmes d'exploitations mobile pour permettre des communications inter-processus entre applications [37]. En effet, de tels middlewares ciblent une empreinte mémoire faible et une relative simplicité, ce qui n'est pas le cas des technologies comme CORBA ou Java RMI. Ces deux approches que nous détaillons ci-après ciblent de fait un ensemble plus large d'applications.

La nature spécifique des données échangées dans les systèmes de grilles informatiques (c'est-à-dire de larges tableaux de donnés numériques) rendent l'utilisation de protocoles de transport tels que SOAP, basés sur un encodage XML, particulièrement mal adaptés aux grilles [70]. Néanmoins, des approches hybrides permettent aujourd'hui de construire des middlewares pour les grilles de calculs en se basant sur Services Web, notamment en s'affranchissent d'XML pour l'encapsulation des messages mais en le gardant pour la description de contrats [131]. Les Services Web sont également présentés ci-après.

2.2.3 ORB / CORBA

Dès le début des années 90, plusieurs entreprises ont développé leur propre framework pour simplifier la création d'applications distribuées de type RPC (ex. Sun ONC [64], Appollo NCS et DCE [87]). Néanmoins, la plupart de ces frameworks étaient liés au langage C et aux systèmes Unix et n'étaient pas interopérables. Ainsi, faire communiquer différents programmes s'exécutant sur des machines distinctes et potentiellement écrits suivant des langages de programmation différents relevait du cauchemar [72]. C'est dans ce contexte qu'est né CORBA, *the Common Object Request Broker Architecture*, proposé par le consortium industriel *Object Management Group* (OMG) [4].

La vision portée par CORBA s'inscrit dans le domaine des systèmes distribués orienté objet. Selon l'approche CORBA, il n'y a pas de distinction du point de vue du programmeur entre des objets qui partagent une zone mémoire et des objets appartenant à des machines distantes. Cette vision est similaire à celle apportée par RPC, mais adaptée au paradigme de l'orienté objet plutôt que procédural. Ainsi CORBA fournit un standard avec des *mappings* entre langages (d'abord C++, puis Java en 1998) et un ensemble d'outils permettant aux développeurs de concevoir des applications distribuées en suivant les principes de la programmation orientée objet. L'intérêt ici, en plus de celui relatif de la transparence de la distribution, est d'avoir un standard qui à l'aide de *mapping* permet de faire inter-opérer des clients et serveurs de différents langages de programmation orientés objet tels que Java ou C++.

CORBA s'articule autour d'un élément principal, nommé *Object Request Broker*(ORB). Un ORB permet de transmettre les requêtes des clients vers les implémentations objets

4. Bien que CORBA soit une initiative du consortium OMG, il hérite de concepts provenant de différents travaux de recherche [113, 18, 117]

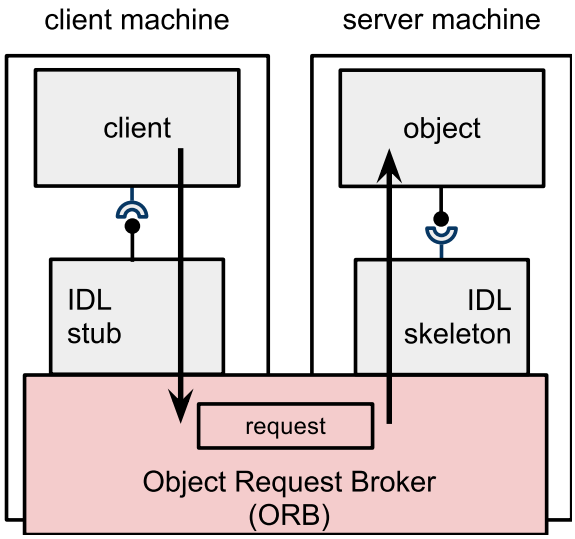


FIGURE 2.4 – Principes des ORB (requêtes client vers objet serveur).

visées et ce d'une manière transparente. Ainsi les interactions entre objets via un ORB se font exclusivement aux travers d'une interface définie suivant un langage générique nommé *Interface Definition Language* (IDL).

Un exemple d'une telle interface est présenté par le listing 2.1 ci-dessous.

```
1 module HelloWorldIDL
2 {
3     interface HelloWorld
4     {
5         string sayHello();
6     };
7 };
```

Listing 2.1 – Exemple de définition d'interface en IDL.

IDL est ainsi un langage standard pour la définition des interfaces d'objets. Similairement à RPC, les clients peuvent accéder aux objets seulement aux travers de leurs interfaces, seules les opérations de l'objet exposé par son interface IDL peuvent être appelées.

En pratique, une interface IDL est compilée en *stubs* client (côté client) et *skeleton* objet (côté serveur). Les *stubs* et *skeleton* servent de proxys entre le client et le serveur. Dès lors que les interfaces sont définies avec le standard IDL, un *stub* et un *skeleton* programmés suivant une même interface IDL peuvent s'interfacer, même s'ils sont compilés dans des langages différents ou s'ils s'exécutent sur des ORBs de vendeurs différents. En effet, des *mappings* IDL permettent de lier les définitions et types IDL vers des langages comme C++, Java ou encore Smalltalk. Par conséquent, tous les programmes basés sur CORBA devraient, en théorie du moins, pouvoir interopérer indépendamment du système d'exploitation, du langage de programmation ou encore du type de réseaux.

En pratique, dans CORBA, chaque instance d'objet à une unique référence d'objet (*ObjRef*). Les clients utilisent ces références d'objets pour guider leurs appels vers la bonne instance. Le client agit comme s'il effectuait l'appel sur l'instance de l'objet alors qu'en réalité l'appel est effectué sur le *stub*. Comme dans les approches de type RPC, l'appel est propagé du *stub* aux travers de l'ORB vers le *skeleton* pour enfin s'effectuer sur l'objet ciblé. Quand l'ORB traite une référence d'objet correspondant à un objet distant, il transfert l'invocation par le réseau jusqu'à l'ORB de l'objet distant. La communication inter ORB est possible grâce à l'utilisation d'un protocole commun appelé *Internet Inter-ORB Protocol* (ou IIOP). Du point de vue du client, le fait que l'objet soit distant est totalement masqué, seul l'ORB sait que la référence demandée est celle d'un objet distant. De ce fait, le code du client reste le même, que l'objet soit local ou distant.

Dés à la fin des années 90, un nombre important de middleware et de frameworks, aussi bien académiques qu'industriels, se sont appuyés sur CORBA et DCOM [5] pour répondre à des besoins d'adaptation. Il s'agissait par exemple de middlewares adaptatifs et réflexifs visant les applications mobiles, les applications ubiquitaires [132, 124, 19] ou encore les systèmes de combat distribués [127]. Ce type de middleware étend les modèles à composants distribués proposés par CORBA ou DCOM en leur ajoutant des mécanismes de réflexion [20]. Fort de ces mécanismes, les applications peuvent être reconfigurées et recomposées à chaud [30]. Néanmoins, CORBA et DCOM n'ont pas réussi le passage à l'échelle du Web. C'est d'ailleurs pour cette raison que Microsoft à abandonné DCOM au profit des web-services. Michi Henning [72] donne un très bon résumé des problèmes de l'approche CORBA, et notamment ces problèmes de passage à l'échelle dus a la complexité des choix architecturaux (ex. les *Interoperable Object Reference*) mais aussi les problèmes de sécurité et de versionnement (indispensables à l'évolution) ou encore l'absence de spécification concernant la gestion des *threads*.

2.2.4 REST

Le style architectural *Representational State Transfer* (ou REST) peut être vu comme un modèle abstrait de l'architecture du Web [48]. En effet, REST a été spécifiquement créé pour répondre aux besoins du Web, conjointement au protocole HTTP/1.1 qui constitue une réification du style REST.

REST s'articule autour du concept de *ressources*. Chaque ressource est référencée par un identifiant global, par exemple avec un URI dans le cas de HTTP. Les ressources sont essentiellement des sources primaires d'information. Les différents composants présents sur le réseau (agents utilisateurs, serveurs) manipulent ces ressources aux travers de leur représentation. Les représentations de ressources peuvent être considérées comme des documents qui contiennent l'information définie par une ressource. Ainsi, les composants peuvent échanger ces représentations en communicant via une interface standard (comme celle définit par HTTP). De plus, un certain nombre de connecteurs, tels que des proxies, des systèmes de cache ou encore des *pipes* peuvent servir de médiateur pour acheminer une requête entre deux composants. Un des principes fondamentaux est que cette médiation doit

5. DCOM est un modèle à composant distribué proposé par Microsoft et était un des principaux concurrents de CORBA.

être transparente. En effet, une application REST doit pouvoir interagir avec un ressource en connaissant uniquement son identifiant et l'action à effectuer. Elle ne doit pas avoir à connaitre les éventuels intermédiaires (cache, proxy, passerelle, firewall, etc.) entre elle et le serveur contenant l'information. En revanche, l'application doit être capable de comprendre le format de l'information (la représentation) retourné par le serveur, tel qu'une page HTML ou des documents JSON ou XML.

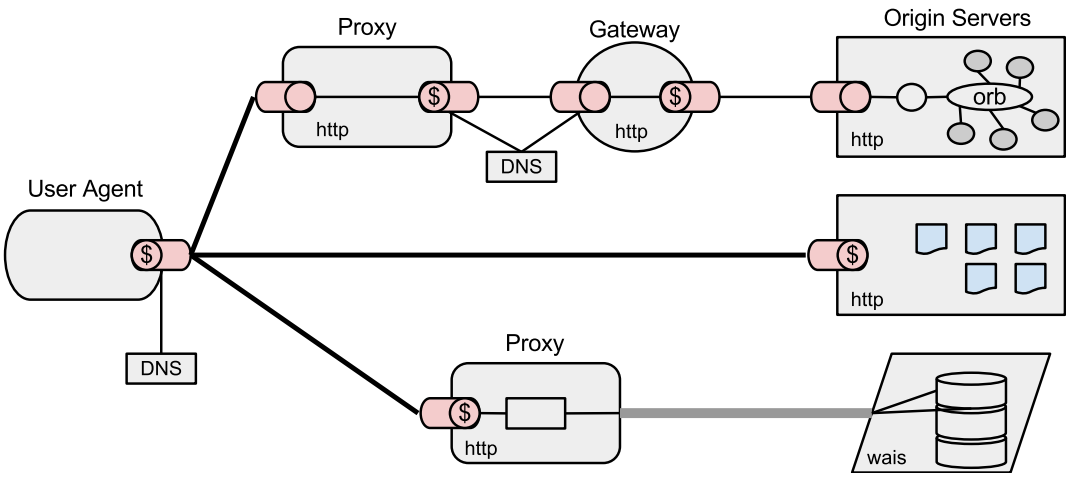


FIGURE 2.5 – Vue processus d'une application basée sur REST (adaptée de [48]).

Les concepts proposés par REST découlent de l'assemblage de contraintes dérivées de plusieurs styles architecturaux, à savoir *Replicated Repository*, *cache*, *client-server*, *layered system*, *stateless*, *virtual machine*, *code on demand* et *uniform interface*. Ainsi la séparation des préoccupations via une interface uniforme découle des styles *client-server* et *uniform-interface* et permet aux serveurs et client d'évoluer indépendamment les uns des autres. Une application REST est *stateless*, c'est-à-dire que les serveurs ne contiennent aucune information liée au contexte des clients entre chaque requête. Les requêtes doivent donc contenir l'ensemble des informations nécessaires à leur traitement. Néanmoins, les serveurs peuvent posséder un état si les états côté serveurs sont publiés comme ressource. REST hérite aussi des styles *cacheable* et *layered system*, des intermédiaires tels des caches sur les réponses peuvent être placés de manière transparente entre les clients et les serveurs. De ce fait, des intermédiaires

peuvent aisément être ajoutés pour permettre le passage à l'échelle ou de meilleures performances. De par cette nature hybride, REST permet la substitution dynamique de composants, la mise en cache et réutilisation d'interactions passées et le traitement d'actions par des intermédiaires.

Une des particularités de REST est qu'il s'abstrait des détails d'implantation des composants et de la syntaxe des protocoles pour se concentrer principalement sur le rôle de ces composants et des contraintes sur leurs interactions avec d'autres composants.

Pour résumer REST s'articule autour de six principes fondamentaux [140] :

1. L'information est abstraite sous la forme d'une *ressource*, identifiée par une URL. Toute information qui peut être nommée peut constituer une ressource.
2. Une ressource est représentée sous la forme d'une séquence d'octets. Cette représentation est complétée par un ensemble de métadonnées pour décrire ces octets. Les composants REST peuvent s'aligner sur une forme particulière de représentation.
3. Toutes les interactions sont indépendantes du contexte. Cela signifie que chaque interaction contient l'information nécessaire au traitement de la requête, indépendamment de tout historique.
4. Seul un petit nombre d'opérations bien définies peut être effectué par les composants sur les ressources. Ces opérations doivent être communes et supportées par toutes les ressources conformes à une réification spécifique de REST. Par exemple, toutes les ressources exposées via HTTP doivent supporter ces opérations de manière équivalente.
5. Les opérations idempotentes et les métadonnées des représentations sont fortement encouragées pour permettre la mise en cache et la réutilisation des représentations.
6. La présence d'intermédiaire est encouragée. En effet, des intermédiaires du type filtrage ou redirection peuvent aussi utiliser les métadonnées et les représentations, dans les requêtes ou réponses, pour augmenter, restreindre, ou modifier les requêtes et réponses d'une façon qui est transparente à la fois pour le client et le serveur d'origine.

Le Web est probablement le meilleur ambassadeur du style REST. En suivant ce style, les frameworks basés sur REST permettent un développement décentralisé et libèrent les

applications distribuées d'un contrôle centralisé qui tend à être imposé par les approches traditionnelles qui héritent de RPC [146]. Du fait de l'ubiquité du web, il existe un grand nombre de frameworks et de bibliothèques qui réifient entièrement ou en partie le style REST, notamment en fournissant une implémentation de http. On citera naturellement les premiers serveurs web [153, 50], les bibliothèques http [49] ou encore, côté client, les navigateurs Mosaïc [154], Firefox [105] ou encore Google Chrome [68]. Néanmoins, un certain nombre d'évolutions comme les CGI, les langages de scripts côté client (JavaScript), les web-services dits RESTful et dernièrement les mashups ont introduit du dynamisme dans le Web, ce qui à provoqué des dissonances dans l'architecture du web qui entrent en conflit avec certains des principes de REST [41].

Ces limitations ont donné lieu à plusieurs travaux, notamment AREST [84], qui propose d'étendre REST pour le support d'interactions asynchrones (basées sur des événements) ou plus récemment le style CREST [40] qui fait la constatation que l'architecture sous-jacente au web se déplace du besoin d'échange de contenu statique vers l'échange de *calcul actif*. CREST propose ainsi une architecture permettant des échanges suivant les principes de REST de programmes préalablement transformés sous forme de fonctions récursives par un processus de *lambda lifting* (technique empruntée aux langages fonctionnels [81]. En dehors du Web, REST a été choisi pour la création d'applications de gestion de capteurs dans le domaine de l'informatique ubiquitaire [101] et a notamment été réifié par un protocole de gestion de réseaux [152].

2.2.5 Web-Services

Durant la phase de croissance de CORBA (milieu des années 90), la progression de Java et l'émergence du web ont bousculé le paysage informatique. Si CORBA a su s'adapter à Java en proposant un *mapping* pour ce langage, rien n'a été fait pour réellement prendre avantage de l'essor du Web. C'est ainsi qu'ont émergé les Web-Services, idéalement perçus comme la conséquence de l'évolution du Web vers une plate-forme universelle simplifiant les interactions entre diverses applications marchandes, scientifiques et de nos jours sociales. Le but ultime des Web-services est de permettre l'automatisation d'interactions entre applications au dessus du web, mais aussi l'intégration d'applications s'exécutant sur des infrastructures réseaux propriétaires avec le web [33]. Ainsi, la cible principale des Web-Services n'est pas l'utilisateur final mais les développeurs qui souhaitent disposer des

fonctionnalités fournies par ces services dans leurs applications. Il existe aujourd'hui une myriade d'applications web fournissant des web-services : Flickr [156] qui fournit des services permettant *d'uploader* et d'annoter des photos, Amazon qui fournit un grand nombre de web-services qui constituent une plate-forme pour le *cloud computing* [6] ou encore les web-services *Google Map* [69] qui permettent d'obtenir des données géographiques.

Deux approches sont principalement utilisées pour mettre en œuvre des Web-Services. La première concerne les Web-Services basés sur SOAP et est fondée sur les principes d'une architecture orientée service (SOA). Ce type d'architecture a émergé comme une réponse aux problèmes d'échanges de services [115]. La seconde approche concerne les Web-services dit *RESTful* ou encore basée sur l'approche orientée ressources [122]. Ainsi ces web-services ne reposent pas sur SOA mais plutôt sur des concepts plus proches du style REST et sont ainsi centrés sur la notion de *ressource* plutôt que de *service*.

L'origine des Web-services remonte à la définition de la spécification XML-RPC [155] et à l'apparition, peu de temps après, du protocole *Simple Object Access Protocol* (SOAP) introduit par Microsoft. SOAP est un protocole léger permettant l'échange d'information structurée via le web [149]. Les Web-services basés sur SOAP suivent l'approche orienté service et s'articulent autour du concept de service comme brique fondamentale pour le développement d'applications. Papazoglou [114] définit les services de la manière suivante :

> "Services are self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications."

En théorie, un service a une description exposant ses capacités de manière agnostique, c'est-à-dire indépendante de son implémentation. Le service peut donc être utilisé afin de composer des applications faiblement couplées à un moindre coût. Les applications basées sur les services sont ainsi développées comme des ensembles indépendants de services interagissant entre eux aux travers d'interface bien définies. Ces applications s'agencent autour de trois acteurs : les fournisseurs de service qui offrent des services, les consommateurs ou clients de service qui requièrent et utilisent des services offert pas des fournisseurs et enfin le courtier de service ou *registry* contenant les descriptions des services publiés par les fournisseurs et que les consommateurs utilisent pour découvrirent les services (voir figure 2.6). L'élément central qui autorise ce mode d'interaction est la *description de service*,

les descriptions, aussi appelées spécifications, exposent les capacités, interfaces, comportements et qualité des services. Ainsi, la publication de la description des services disponibles dans le courtier donne les moyens nécessaires à la découverte, la sélection, la liaison et la composition des services.

Idéalement, les services doivent répondre aux trois principes suivants [114] :

1. Être neutres technologiquement. Cela implique que les standards liés à la liaison (protocole de communication), la description et la découverte de services doivent se conformer avec des standards largement acceptés.
2. Faiblement couplés. Les services ne doivent dépendre d'aucune connaissance particulière, de structure interne ou encore de convention, aussi bien du côté du client que du coté serveur. Cela signifie qu'il ne doit pas y avoir de couplage entre le client et le serveur autre que la description du service, c'est-à-dire l'interface.
3. Transparence de la localisation. Les registres qui contiennent les descriptions de service doivent être accessibles à tous clients qui doivent pouvoir sélectionner et appeler les services indépendamment de leur localisation.

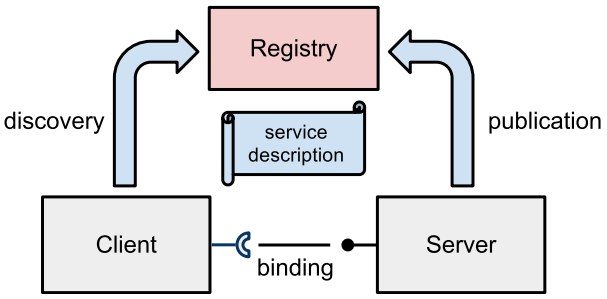


FIGURE 2.6 – Intéractions dans SOA.

C'est notamment pour garantir la neutralité technologique que SOAP n'est pas un protocole à proprement parlé mais plutôt un langage de description exprimé en XML. Un document SOAP peut ainsi être transféré par différents protocoles de transport tel que HTTP. En pratique, HTTP à été largement adopté comme couche transport pour SOAP pour que les implémentations puissent aisément passer au travers de pare-feux [120]. Cependant, SOAP peut aussi être utilisé au dessus d'autres protocoles de transport tels que TCP, SMTP ou encore JMS. Peu après l'introduction de SOAP comme langage de communication pour

les Web-services, le *Web Service Definition Language* (WSDL) a été introduit pour servir de langage de description de service. Comme SOAP, WSDL est une grammaire XML. Ainsi, pour envoyer un message à un Web-service particulier, une application peut simplement parcourir son WSDL et construire dynamiquement des messages SOAP. Le dernier acteur manquant pour permettre une interaction SOA est le courtier de service. Cette entité est connue sous le nom de registre UDDI (pour *Universal Description, Discovery and Integration*) dans le monde des web-services, mais ne dispose pas encore de standard officiel. De nombreux autres standards ont été formulés pour permettre la composition, et l'évolution de ces architectures orientées services [6].

Néanmoins, ces standards sont surtout des niches [7] et ne peuvent être largement déployés du fait qu'ils supportent mal le passage à l'échelle du web [40, 147]. Un des problèmes notoires est l'utilisation quasi systématique d'interaction SOAP employant la méthode http *POST* sans prendre en considération le fait que le résultat peut être mis en cache. Or, contrairement à la méthode HTTP *GET*, elle ne peut pas être idempotent. Ainsi, aucun résultat de ces interactions ne peut être mis en cache, du fait que l'on ne peut garantir qu'il reste identique dans une fenêtre de temps donnée. En pratique, le fait de devoir encapsuler toute l'enveloppe SOAP au dessus de HTTP viole la plupart des principes du style d'architecture du web, à savoir REST. C'est ainsi qu'ont émergé un nouveau type de web-service plus proche de REST connu sous le nom de *RESTful Services*.

Contrairement aux Web-services fondés sur SOAP, les Web-services dits *RESTful* s'articulent sur le concept de ressource plutôt que de service, et tendent ainsi à reprendre les principes du style REST. L'utilisation de services *RESTful*, plutôt que basés sur SOAP, a d'ailleurs apporté de réels gains, aussi bien en termes d'usage que de performance [8]. Ces gains s'expliquent notamment par le fait que les propriétés fondamentales derrière les Web-services *RESTful* s'accordent très bien avec les systèmes de cache générique de http/1.1 [151]. De plus, l'absence de contrat spécifique pour chaque service (WSDL), du fait de la contrainte d'avoir une interface uniforme (induite par REST), réduit le couplage

6. Pour une liste non exhaustive voir `http://en.wikipedia.org/wiki/List_of_Web_service_specifications`.

7. Notamment pour les Grilles informatique [134].

8. En 2006, 85% des utilisations des web-services d'Amazon ce faisait au travers de leurs API REST plutôt que SOAP `http://www.oreillynet.com/pub/wlg/3005`.

entre le client et le serveur et minimise ainsi sans effort les différences d'interface et de sémantique au sein de ressources hétérogènes [146].

En effet, SOAP induit un typage fort au travers du WSDL, ce qui nuit à la flexibilité des clients. Au contraire, une ressource REST n'est pas typée, et peut être représentée via tout type de format d'échange de données (XML, mais aussi JSON ou encore du simple texte). Les clients peuvent alors continuer à interpréter les messages, même si ces derniers on légèrement changés. Le typage fort n'est donc pas le meilleur choix pour les systèmes distribués faiblement couplés. D'ailleurs, certaines entreprises n'ont pas hésité a définitivement abandonner les API SOAP qu'ils fournissaient et a imposé l'utilisation d'une API REST à leur utilisateurs [142]. Néanmoins, les implémentations de services REST sont encore souvent limitées en comparaison aux Web-services SOAP proposé par les mêmes fournisseurs.

Les web-services ont traditionnellement été destinés aux développements d'applications d'entreprises, en particulier comme élément essentiel des serveurs d'applications. Le standard J2EE, produit d'une initiative industrielle dirigée par Sun Microsystems et la communauté Java, a donné naissance à plusieurs serveurs d'applications tel que Glassfish, WebSphere, Jboss ou encore Jonas. Hors du monde J2EE, on peut citer la plate-forme Microsoft .NET qui n'est pas un standard mais un produit à part entière, ou plus exactement une suite logicielle à destination des entreprises, visant a faciliter la conception et la création de Web-services. Malgré le fait que ces serveurs d'applications partagent tous une même architecture et voire dans certain cas un même standard, leur interopérabilité ne reste qu'hypothétique et demande plus que l'alignement de descriptions XML. En réalité, si ces projets diffèrent, c'est essentiellement dû à leur *business model* [67].

Parallèlement à l'évolution des web-services et de SOA, un nouveau modèle d'architecture à émergé pour permettre la conception et l'implémentation d'interactions et de communications entre des applications interagissent toutes dans un écosystème SOA, essentiellement aux travers de Web-services. Il s'agit des *Entreprise Service Bus* (ESB). L'ESB est une plate-forme d'exécution centralisée ou la médiation entre différents Web-services est organisée aux travers d'un middleware à message.

Outre, les serveurs d'applications, les ESB et les applications Web, les Web-service et

SOA forment aussi la base d'un nombre important de travaux académiques, notamment pour le calcul sur grilles [134], les réseaux de capteurs [90], l'informatique ubiquitaire [125, 71, 78, 143], et plus récemment le *cloud computing* [109].

2.3 Conclusion

Dans ce chapitre nous avons décrit plusieurs styles architecturaux classiques qui visent à simplifier le développement d'applications distribuées faiblement couplées. Ces styles ont évolué de RPC vers CORBA et DCOM essentiellement pour des besoins de standardisations et en particulier pour permettre une certaine comptabilité entre les implémentations divergentes fournies par différentes entreprises mais aussi en écho au développement des langages orienté objets. L'avènement du Web a alors bouleversé les standards faiblement établis et a donné naissance au Web-services sensés faciliter l'intégration entre le Web et les systèmes d'entreprises. Malgré ces différences, ces approches partagent les mêmes concepts et les mêmes objectifs :

1. La transparence de la distribution, c'est-à-dire cacher la nature distribuée des applications aux développeurs.
2. Permettre aux composants formant l'application d'interagir entre eux aux travers d'interfaces *stables* [9] et ainsi que les modules n'aient pas de dépendances sur les structures internes des autres modules.
3. Faciliter la collaboration entre différentes applications, la réutilisation des composants/services et ainsi réduire les efforts de développement et de maintenance.
4. Masquer l'hétérogénéité à la fois des différentes plate-formes d'exécution mais aussi des protocoles et formats de communication qui forment le système sur lequel s'exécute l'application.

C'est ainsi qu'est apparu ces dernières années un grand nombre de framework et middleware s'appuyant sur ces styles architecturaux pour faciliter la conception et la création d'applications distribuées faiblement couplées. Parallèlement à ces approches qui partagent une même philosophie, le Web s'est imposé avec son propre style architectural, REST. REST a donné naissance à des Web-services hybrides qui ne suivent plus une architecture

9. Les interfaces ne sont pas nécessairement stables si le système gère le versionnement.

orientée service mais qui sont centrés sur le concept de ressources pour répondre aux problématiques de passage à l'échelle particulières au web. Aujourd'hui SOA et REST sont les deux styles architecturaux de référence pour le développement d'applications distribuées faiblement couplées.

Néanmoins, l'émergence de nouveaux domaines comme le *cloud computing* et l'informatique ubiquitaire et par conséquent, les nouveaux besoins qui en résultent, changent la donne. En effet, la nature élastique des ressources du Cloud, et la disponibilité éphémère des objets ubiquitaires ainsi que l'hétérogénéité des standards de communication posent de nouveaux challenges. Ainsi, les besoins d'adaptation et d'évolution durant l'exécution pour répondre à des changements fonctionnels mais aussi contextuels doivent aujourd'hui être pris en considération et donc pouvoir être adressés dés la conception de l'architecture pour pouvoir avoir une réalité durant l'exécution. Même si certains des styles présentés ici permettent déjà un certain niveau de dynamisme, notamment REST.

Dans le chapitre suivant, nous présentons un état de l'art sur les systèmes dynamiques et en particulier les architectures et techniques permettant à des applications basées sur des composants d'évoluer durant l'exécution. Le but étant d'explorer les différents styles et techniques pour la conception d'applications dynamiques qui ont inspiré et influencé la proposition et l'approche décrite dans cette thèse.

Chapitre 3

Systèmes logiciels dynamiques

"Change alone is perpetual, eternal, immortal." *Arthur Schopenhauer*

3.1 Introduction

3.1.1 Adaptation

L'adaptation est le processus permettant à un système vivant ou artificiel d'assurer ses fonctions de façon continue dans un environnement évolutif. Elle s'effectue lors d'un processus discret modifiant la structure ou le comportement d'un système en fonction de l'environnement dans lequel il est plongé. La plupart des systèmes ont besoin de s'adapter pour survivre car très peu évoluent dans des environnements parfaitement stables. Ainsi, la pérennité d'un système est souvent liée à sa capacité à s'adapter à des milieux instables [91].

Cette définition et les constatations associées s'appliquent également à la plupart des systèmes artificiels. C'est par exemple le cas pour les bâtiments dans lesquels nous vivons. Ces édifices sont sans cesse remodelés par leurs occupants pour mieux répondre à leurs besoins. Patricia Waddy, dans son étude des palais romains du 17ème siècle [150], insiste d'ailleurs sur le fait qu'un bâtiment vit dans le temps et que cette vie est intimement liée aux personnes qui l'occupent. Le fameux – bâtiment 20 – du MIT est un bel exemple de bâtiment qui a su être adapté aux besoins de ses usagers [102]. Ainsi, le travail des architectes n'est pas seulement de maîtriser l'espace mais également de considérer le temps. Ceci

est particulièrement important pour les bâtiments utilisés dans divers contextes comme les salles de spectacle par exemple. Dans ce cas, l'adaptabilité est à l'évidence une exigence de première importance. Ceci est également le cas pour certains types de logiciels, tels que les logiciels pervasifs, dont la nature même est de s'exécuter dans des contextes différents et difficilement prévisibles.

Le processus d'adaptation est généralement mis en œuvre alors que le système est en cours d'utilisation et, la plupart du temps, ne doit entraîner aucune interruption de service. De la même manière qu'un bâtiment continue à être utilisé et modifié après son inauguration, les logiciels doivent être constamment adaptés tout au long de leur existence, au gré des besoins de leurs utilisateurs. De plus, le temps est aujourd'hui plus précieux que jamais et la maxime populaire associant temps et argent est on ne peut plus vrai pour les logiciels [1].

3.1.2 Adaptation logicielle

Mais que signifient les concepts de structure et de comportement pour un système logiciel ? La notion de structure logicielle, tout d'abord, a donné lieu à maintes définitions sans permettre l'obtention d'un consensus. Ceci est certainement dû au fait que la structure d'un logiciel peut être perçue et manipulée suivant différents niveaux d'abstraction. Au final, cependant, la structure d'un logiciel réside au niveau de son code binaire, c'est-à-dire au niveau du code exécuté par une machine ou un interpréteur. Adapter un logiciel revient donc à modifier son exécutable, d'une façon ou d'une autre.

Il en est de même pour la notion de comportement d'un système logiciel : il peut être défini à plusieurs niveaux d'abstraction mais se concrétise finalement au niveau du code exécutable. Il faut noter d'ailleurs que de nombreux auteurs depuis Parnas [116] considèrent le comportement comme une structure particulière [14].

Que dire maintenant de la notion d'environnement logiciel ? Elle doit assurément être prise dans son acceptation la plus large. A notre sens, l'environnement logiciel inclut tous

1. L'interruption d'une application se traduisant comme une baisse de sa disponibilité est très coûteuse. Pour les fournisseurs d'accès internet, les fournisseurs de services bancaires ou pour les sites de e-commerces, chaque minute perdue représente des sommes colossales.

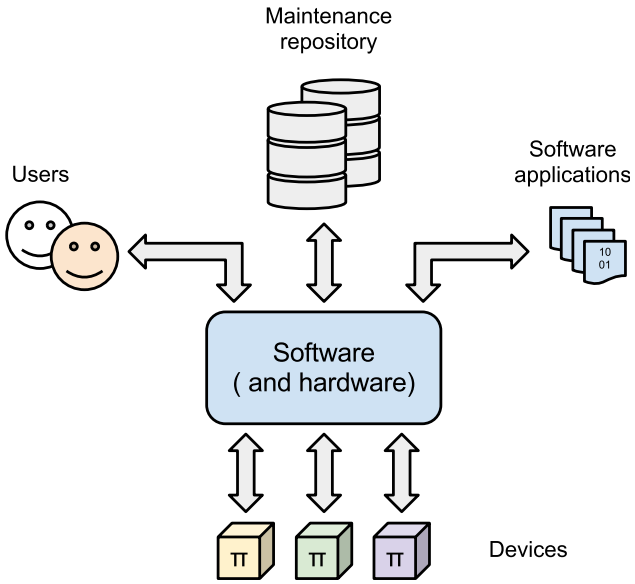


FIGURE 3.1 – Environnement logiciel.

les acteurs, c'est-à-dire tous les partis capables d'interagir avec le logiciel considéré. Il peut correspondre à des entités logiciel ou de maintenance comme des équipements communicants, des applications ou des facilités de gestion par exemple. Il comprend également le hardware sur lequel repose l'exécution du logiciel. Ceci est illustré par la figure 3.1.

Finalement, l'environnement inclut tous les humains utilisant le système d'une façon ou d'une autre. Il faut noter ici que les besoins et désirs des personnes évoluent au cours du temps, ce qui est parfaitement acceptable. Ainsi, les logiciels doivent être régulièrement mis à jour de façon à rester cohérents avec les attentes des utilisateurs.

Comme indiqué précédemment, les adaptations peuvent s'effectuer alors qu'un système est en cours d'utilisation. Elles sont déclenchées à la suite de phénomènes liés à l'exécution qui peuvent être perçus par le logiciel lui-même, par des utilitaires externes ou par l'administrateur.

En logiciel, il existe de nombreuses adaptations possibles. Elles incluent essentiellement les types d'actions suivants :

- Correction d'un bug. Le but ici est de changer la structure interne ou le comportement d'un système sans modifier son périmètre fonctionnel. Cette adaptation est déclenchée pour corriger un fonctionnement anormal ou sub-optimal tout en conservant les services originaux du logiciel.
- Adaptation à l'environnement. Le but maintenant est de prendre en compte une évolution de l'environnement. Cela peut, par exemple, correspondre à l'arrivée ou à la disparition d'un acteur ou à la modification d'interfaces de certains acteurs. Ici également, le but n'est pas de modifier le périmètre fonctionnel du logiciel mais bien de prendre en compte des changements externes.
- Évolution fonctionnelle. L'objectif dans cette situation est très différent. Il s'agit en effet de modifier les fonctionnalités du logiciel : des fonctions peuvent ainsi être ajoutées, retirées, ou modifiées. De telles adaptations sont déclenchées de façon à mieux satisfaire les utilisateurs ou pour profiter de nouvelles conditions d'exécution.
- Évolution non fonctionnelle. Le but dans ce dernier cas est de modifier les propriétés liées à la réalisation des fonctions fournies sans pour autant modifier le périmètre fonctionnel du logiciel. Par exemple, des adaptations peuvent permettre d'améliorer les performances ou certains éléments de sécurité.

La communauté du Génie Logiciel reconnaît aujourd'hui que l'adaptabilité est une caractéristique essentielle des systèmes logiciels. Cela est dû au caractère changeant des exigences et des conditions d'exécution des logiciels mais aussi à la difficulté de construire des systèmes répondant aux besoins et ne nécessitant pas de maintenance corrective.

Pour autant, l'adaptation logicielle demeure particulièrement complexe et constitue toujours un défi majeur. Bien que le sujet ait été étudié depuis des dizaines d'années, l'adaptation repose encore trop souvent sur des solutions ad-hoc. En effet, la plupart des systèmes ne sont pas faits pour être adaptés facilement. Ils sont simplement conçus et codés de façon à répondre aux exigences du moment, avec très peu de projection dans l'avenir. L'évolution dans le temps n'est tout simplement pas considéré en général.

Ainsi, adapter un système peut se traduire en un processus excessivement complexe demandant une grande expertise, des efforts considérables et perturbé par des effets de bord difficilement contrôlables.

3.1.3 Adaptation dynamique

Une adaptation peut être statique ou dynamique. Une adaptation statique demande l'arrêt du logiciel et, donc, son redémarrage une fois les modifications effectuées. Les adaptations dynamiques, quant elles, s'effectuent sans interruption de service.

Les adaptations statiques sont plus faciles à mettre en œuvre. Elles posent cependant des problèmes complexes à résoudre puisque, de toute façon, le logiciel a déjà été utilisé un certain temps. Un problème majeur, notamment, est que l'état du logiciel est souvent perdu entre deux activations, même si certains éléments peuvent être conservés de diverses façons (base de données, utilitaires externes, etc.). En fait, la simple définition de la notion d'état peut ne pas être compatible entre deux activations : une modification logicielle peut en effet modifier la définition même de l'état d'un système. Un autre problème est lié aux interactions existantes entre le logiciel est son environnement. Parfois, des transactions complexes peuvent être nécessaires pour maintenir un certain niveau de cohérence. Et même si de telles transactions sont mises en place, on peut obtenir des situations où des logiciels externes se retrouvent dans des états d'attente (ou de blocage) entraînant des coûts importants. Ainsi pour des raisons techniques et économiques, de nombreux systèmes demandent des adaptations dynamiques, c'est-à-dire sans interruption de service.

Les adaptations dynamiques soulèvent quant à elles des problèmes particulièrement ardus. En particulier, il faut assurer la correction fonctionnelle du logiciel à tout instant, maintenir ses propriétés non fonctionnelles, conserver son état et les flux de contrôle et préserver les interactions avec les logiciels externes pour éviter des interruption de services des autres logiciels (parfois plus importants que le logiciel considéré !).

La correction fonctionnelle, tout d'abord, est une propriété à assurer de façon non négociable. Pendant et après son adaptation, un système logiciel doit satisfaire les exigences initialement spécifiées ainsi que les exigences –du moment–. Il faut ainsi être capable de définir et de d'appliquer à la fois des tests de non régression et des tests liés aux évolutions. Bien évidemment, les stratégies de tests sont rendues complexes par le fait que le système logiciel ne peut se permettre aucune interruption. Les vérifications doivent ainsi être effectuées pendant l'exécution et, en général, ne peuvent être mises en place comme des tests traditionnels.

Ensuite, l'application de mise à jour dynamique ne doit pas altérer la qualité du logiciel considéré. Des pertes de données, des "plantages" (*crash*) ou des baisses significatives de performance ne sont pas acceptables. Ceci demande un niveau de contrôle important sur les mises à jour : les états du logiciel doivent être identifiés et conservés, les flots de contrôle doivent être maîtrisés de façon à éviter toute perte de traitement ou des états incohérents, et les propriétés non fonctionnelles comme la performance doivent être surveillées.

En général, il n'est pas possible de mettre à jour du code actif. Un code actif correspond à un code en cours d'exécution ou référencé dans la pile d'exécution. Bien évidemment, un système logiciel peut attendre que le code à mettre à jour devienne inactif, mais cela n'est malheureusement pas toujours possible. En effet, certaines parties de code peuvent être constamment référencées dans la pile et, ainsi, il n'existe pas de moment propice à leur mise à jour. Une technique usuelle est d'implanter un protocole de quiescence. Le principe de cette technique est d'intercepter les appels au code à mettre à jour et d'attendre que ce code devienne inactif. Quand cela est le cas, le code visé est mis à jour et les appels bloqués sont relancés. Le problème majeur lié à cette approche est que, dans certains cas, des dépendances de code peuvent amener à des situations de blocage.

Une manière relativement sûre de procéder à des mises à jour est d'opérer de façon non destructive. Cela signifie que certains éléments structurant du logiciel avant sa mise à jour sont conservés et peuvent donc être récupérés. Par exemple, certains langages de programmation permettent la cohabitation de versions différentes d'une même structure et fournissent une solution élégante à ce problème. Les mises à jour destructives sont plus faciles à réaliser mais sont également plus hasardeuses. Prenons l'exemple de la mise à jour de services Web pour illustrer ce problème. Une solution non destructive, communément utilisée, est de faire cohabiter un certain temps la nouvelle et l'ancienne signature d'un service web. Ainsi, les clients peuvent passer progressivement à la nouvelle interface. Si l'on adopte une approche destructive, l'ancienne interface est immédiatement supprimée. Cela est plus facile à gérer et à administrer mais au risque de voir apparaître des problèmes sérieux au niveau des clients.

Une approche très populaire pour mettre en place des adaptations dynamiques est d'utiliser la redondance, matérielle ou logicielle. Le principe est d'activer une nouvelle version

d'un logiciel sur une nouvelle machine ou une nouvelle couche middleware (JVM par exemple) tandis que l'ancienne version du logiciel continue de s'exécuter sur les ressources initiales [2]. Un régulateur de charge (*loadbalancer*) est alors utilisé pour rediriger les requêtes au logiciel vers la nouvelle version. Cette approche est efficace mais ...pas aussi simple qu'il n'y parait. D'abord, elle ne résout pas le problème du transfert d'état. Dans certains cas, des alignements complexes sont nécessaires pour passer de l'état de la version initiale vers l'état de la nouvelle version (une des solutions est l'utilisation de *sticky sessions* pour assurer que le régulateur de charge dirige chaque client vers le serveur correspondant à leur version courante). Néanmoins, définir le moment du transfert définitif est souvent problématique. En effet, un niveau de contrôle très précis peut être nécessaire quand le nouveau et l'ancien logiciel utilisent des ressources communes. Ainsi, chaque modification des API, des interfaces, ou encore du schéma de données qui engendre un stress sur un point particulier du système peuvent résulter dans l'apparition de situations incohérentes entre les versions co-existantes et nécessite donc la mise en place d'approches dédiés à résoudre cette classe de problème.

En conclusion, il est important de comprendre qu'il n'existe pas aujourd'hui de méthode générique garantissant des mises à jour dynamiques. La plupart des techniques utilisées aujourd'hui sont spécifiques à des domaines ou à des applications particulières. Les mettre en place reste un défi pour les programmeurs. Heureusement, on peut rencontrer des situations favorables : par exemple, lorsque les logiciels n'ont pas d'état ou lorsque le couplage avec des systèmes externes est très faible.

3.2 Adaptations de faible granularité

3.2.1 Système d'exploitation

La problématique de l'adaptation dynamique a été étudiée depuis longtemps dans le domaine des systèmes d'exploitation (OS pour *Operating Systems*). La raison d'être d'un système d'exploitation est d'abstraire les ressources d'une machine et de fournir un ensemble

2. Différentes techniques basées sur cette approche permettent d'automatiser le processus de déploiement logiciel, on pense notamment au *BlueGreen Deployment* et *Canary Deployment* tel qu'illustré par Jef Humble et David Farley [77]

de services communs pour faciliter le développement et l'exécution d'applications logicielles. De nombreux travaux de recherche ont été menés de façon à permettre la gestion dynamique des ressources et des services. C'est une exigence importante de ce domaine qui vise à améliorer la stabilité et la disponibilité des applications. A l'évidence, personne ne veut redémarrer sa station de travail et stopper les applications en cours d'exécution à chaque fois qu'un dispositif USB est introduit.

La plupart des systèmes d'exploitation sont ainsi capables de gérer l'arrivée et le départ de ressources à la volée, c'est-à-dire sans interruption de service. Des techniques variées ont été développées à cet effet et disséminées dans de nombreux domaines d'application. Par exemple, de nombreuses technologies *Plug&Play* issues des systèmes d'exploitation sont aujourd'hui utilisées dans les réseaux pervasifs.

La plupart des systèmes d'exploitation sont également capables de gérer le déploiement dynamique de services techniques. La notion de déploiement, ici, doit être entendue dans son acceptation la plus large. Cela signifie que les systèmes d'exploitation fournissent des facilités permettant l'installation, l'activation, ou encore la désactivation de services techniques. Par exemple, il est possible d'installer une nouvelle commande en Linux en copiant simplement l'exécutable (fichier binaire ou script) dans le répertoire `/bin`. Le service est lancé en tapant son nom et se trouve disponible dynamiquement dans l'environnement d'exécution.

Les systèmes d'exploitation peuvent donc constituer une infrastructure supportant la construction d'applications dynamiques. De nouvelles ressources et du code nouveau peuvent être ajoutés, et retirés, de façon relativement simple. Il faut néanmoins comprendre que le nouveau code n'est pas intégré de façon fine avec le code existant. De fait, il est packagé et déployé comme un service indépendant dans le système de fichier de l'OS. Le code existant peut alors appeler ce nouveau code de façon dynamique, sans interruption.

En dépit de l'indéniable possibilité d'ajouter du code dynamiquement, construire des applications dynamiques directement au dessus d'un OS demeure complexe et souvent fondé sur des mécanismes *ad-hoc*. L'exemple précédent de la commande Linux est basé sur l'introspection d'un répertoire spécifié dans une variable globale. En outre, il est extrêmement difficile d'établir une infrastructure pour l'interception et la redirection de messages

échangés entre deux structures internes à une application. Il est donc difficile, par exemple, de mettre en place des mécanismes de quiescence.

3.2.2 Langages de programmation

La problématique de l'adaptation dynamique a été également largement étudiée par la communauté travaillant sur les langages de programmation. Dans cette section, nous discutons des différentes facilités fournies par les environnements de programmation pour exprimer et effectuer des adaptations durant l'exécution. On s'intéresse en particulier à la possibilité de charger et de décharger des modules, c'est-à-dire des structures programmatiques, à l'exécution. Nous examinerons successivement les langages C, Java, Ocaml et Erlang qui présentent tous des caractéristiques particulières pour l'adaptation. Nous traiterons également des machines virtuelles qui ajoutent certaines capacités intéressantes. Enfin, nous conclurons cette partie en examinant les travaux qui manipulent explicitement le code pour apporter des modifications de façon dynamique, connues sous le nom de *Dynamic Software Updates*.

Édition de liens dynamique

La plupart des programmes sont en effet construits à l'aide de la notion de *modules*. Même si la définition des *modules* varie sensiblement d'un langage de programmation à un autre, on peut néanmoins définir un *module* comme un ensemble de *définitions* qui font correspondre des symboles à des fragments de code. De manière générale, un module est défini dans un seul fichier source. Chaque module source peut alors être compilé puis assemblé afin de produire un code exécutable. Cet assemblage de plusieurs modules est réalisé par un programme appelé *éditeur de liens* [121]. L'éditeur de liens combine le code des modules et résout toutes les références vers des symboles externes en les liants avec des définitions appropriées provenant de modules tiers.

Initialement, l'édition de liens était toujours effectuée de manière statique et avait pour seul but de construire des programmes constitués de plusieurs modules. L'ensemble des références devait être résolu avant de pouvoir exécuter le programme. Néanmoins la plupart des environnements de programmation actuels supportent l'édition de liens dynamique. De tels éditeurs de liens peuvent être appelés par un programme durant l'exécution. Ainsi

le programme est capable de s'étendre lui-même grâce à l'ajout de nouveaux modules. Les éditeurs de liens dynamiques autorisent donc un programme à s'adapter durant l'exécution. Généralement, l'édition de liens dynamique est réalisée soit par *indirection*, soit par *réécriture du code*. La méthode par indirection consiste à compiler les références externes de manière a ce qu'elles soient accessibles aux travers d'une table d'indirection propre au module. L'édition des liens revenant alors à remplir la table. La méthode par réécriture du code consiste à récrire le code de manière à ce que les références pointent directement sur les définitions externes adéquates. L'édition de liens dynamique dans Erlang ou le standard ELF (pour C) utilise l'indirection tandis que la gestion du chargement des modules dans le Kernel linux utilise l'approche par réécriture.

En C

Le chargement dynamique en C diverge selon les systèmes d'exploitation. Les systèmes d'exploitation similaires à Unix supportent le chargement dynamique de modules grâce à la bibliothèque **ld** [123]. Dans le monde C, on parle plutôt de bibliothèques (*libraries*) que de modules ; les modules destinés à être liés à l'exécution sont appelés *Dynamic-linklibraries* (`.dll`) dans windows et *sharedlibraries* (`.so`) dans linux.

L'exemple suivant nous montre comment charger une bibliothèque grâce à la fonction `dlopen()`, comment obtenir l'adresse d'un symbole défini dans la bibliothèque partagée grâce à la fonction `dlsym()` et enfin comment décharger la bibliothèque partagée grâce à la fonction `dlclose()`.

```
1  void     *handle;
2  int      *iptr, (*fptr)(int);
3
4  /* chargement de la librairie souhaite */
5  handle = dlopen("/usr/home/me/libtoto.so", RTLD_LOCAL | RTLD_LAZY);
6
7  /* obtention de l'adresse de la fonction my_function et de la donnee my_data */
8  *(void **)(&fptr) = dlsym(handle, "my_function");
9  iptr = (int *)dlsym(handle, "my_data");
10
11 /* on appelle my_function avec my_data comme parrametre */
12 (*fptr)(*iptr);
13
14 dclose(handle);
```

Listing 3.1 – Chargement dynamique en C.

Le chargement et le déchargement dynamique de module restent des opérations périlleuses en C. En effet aucune vérification n'est effectuée pour s'assurer que le code contenu dans une bibliothèque est de type *safe* ou même bien formé. L'absence de notion de *scope* dans les modules C (ex. .so, .dll) ne permet pas de définir explicitement ce qui doit être ou non partagé dans un module, ce qui introduit indubitablement des problèmes de sécurité. Finalement, aucun contrôle au déchargement du module n'est assuré, ce qui résulte souvent en l'apparition de *dangling pointers* [16]. Ainsi le programme va fatalement échouer s'il continue à manipuler des objets introduits par un module alors que celui si a été déchargé.

En Java

Le code source java est toujours transformé en un code intermédiaire appelé *bytecode* afin d'être exécuté par une machine virtuelle. Il n'existe donc pas d'édition de liens statique en Java ; toute édition de lien en Java est effectuée par la machine virtuelle au travers d'une entité appelée *ClassLoader* [133]. Le *ClassLoader* utilise la délégation pour rapatrier les classes et les ressources nécessaires à l'exécution d'un programme. De façon plus précise, le *ClassLoader* délègue la recherche d'une classe à son parent avant d'essayer de charger lui-même ladite classe.

Contrairement à C, les modules java (les *classfiles*) sont ajoutés à un programme seulement après que des vérifications aient été effectuées. Ces vérifications garantissent certaines propriétés telles que la sûreté du typage.

Le code suivant montre comment une classe java peut être chargé dynamiquement :

```
1 ClassLoader loader;
2 [...]
3 Class type = loader.loadClass(name);
4 Object obj = type.newInstance();
```

Listing 3.2 – Chargement dynamique en Java.

Malheureusement, il n'est pas possible de décharger une classe particulière à partir du *ClassLoader* en java. Le seul moyen est que le *ClassLoader* lui-même soit ramassé par

le *GarbageCollector*. Ce comportement particulier incite à gérer plusieurs *ClassLoaders*, ce qui complexifie le programme, notamment du fait que des classes similaires peuvent être chargées par différents *ClassLoaders*, mais sont incompatibles entre-elles. Certaines constructions du langage ont des comportements différents comme les membres statiques. De plus il n'est pas possible de remplacer une classe déjà ajoutée. Ainsi la nouvelle classe va être ajoutée aux bibliothèques disponibles et le *ClassLoader* chargera alors l'une ou l'autre version lors de l'exécution, ce qui résulte immanquablement en des comportements obscurs.

En Ocaml

OCaml, anciennement connu sous le nom d'Objective Caml est l'implémentation la plus avancée du langage de programmation Caml. Caml est un langage fonctionnel étendu avec certaines fonctionnalités permettant la programmation impérative. Objective Caml étend les possibilités du langage en permettant la programmation orientée objet et la programmation modulaire. Pour toutes ces raisons, OCaml entre dans la catégorie des langages multi-paradigmes [3].

Ocaml dispose d'un module appelé **dynlink** [94] qui permet l'édition de liens dynamique. La bibliothèque *dynlink* supporte le chargement dynamique de module (i.e. *bytecode object files*, .cmo et .cma) dans la machine virtuelle OCaml. Néanmoins, *dynlink* ne fournit pas de facilité particulière pour accéder aux symboles définis par le module. Ainsi le module est responsable d'enregistrer ces points d'entrée dans le programme principal en modifiant la table des fonctions.

Tout module OCaml est compilé. Ceci permet de garantir la sûreté du module. De plus, une vérification est faite au moment de l'édition des liens afin d'assurer que l'interface fournie par le module est cohérente avec le contexte du programme appelant en cours d'exécution. Toujours suivant la même logique de sûreté, le déchargement de module n'est pas permis depuis *dynlink*. En effet, autoriser un tel déchargement résulterait presque fatalement en l'apparition de *dandling pointers* (sans l'ajout de mécanismes additionnels).

3. `http://en.wikipedia.org/wiki/Ocaml`

En Erlang

Erlang est un langage de programmation créé par Ericsson et disponible en Open Source. Ce langage supporte plusieurs paradigmes : concurrent, temps réel, distribué. Son cœur séquentiel est un langage fonctionnel à évaluation stricte, affectation unique, au typage dynamique fort. Il possède des fonctionnalités de tolérance aux pannes et de mise à jour du code à chaud, permettant le développement d'applications à très haute disponibilité [4].

Erlang permet aux développeurs de charger du code dynamiquement et de gérer le remplacement de code durant l'exécution [7]. En effet, Erlang permet aux travers d'appels explicites (`module:func`) de lier dynamiquement `module:func` au code en cours d'exécution. En d'autres termes, à chaque appel explicite d'une fonction, c'est-à-dire en utilisant le nom du module complet devant celui de la fonction, le système va réévaluer la fonction en utilisant la dernière version du module.

Les processus existants qui exécutent la version précédente d'un module continuent d'utiliser cette ancienne version sans aucun problème. Cependant, uniquement deux versions d'un même code peuvent être disponibles simultanément. Ainsi la version ancienne du code courant doit être supprimée si elle existe avant qu'une nouvelle version ne puisse être chargée. A cet effet, Erlang dispose de bibliothèques permettant la détection de code ancien toujours en cours utilisation. Par défaut, tout processus qui exécute du code dans l'ancienne version est tué et le module fraîchement compilé devient la version courante. On comprend qu'il est primordial de bien connaître ces règles par défaut pour ne pas rencontrer de graves problèmes lors de l'exécution (et surtout lors de l'administration).

Erlang est ainsi un langage dynamiquement typé. Par nature même, donc, Erlang ne permet pas de faire des vérifications au moment du chargement des modules. Un tel comportement peut masquer d'éventuelles incohérences de typage jusqu'au moment ou les instructions correspondantes sont exécutées (ce qui rend bien évidemment la gestion de tels problèmes particulièrement complexe et incertaine).

4. `http://en.wikipedia.org/wiki/Erlang_(programming_language)`

Machines Virtuelles

Bien qu'étant des langages fondamentalement différents, Erlang et Java ont un point commun bien particulier. Ils sont tous deux basés sur une machine virtuelle. Une machine virtuelle permet d'isoler les applications de certaines spécificités des ordinateurs telles que le système d'exploitation ou encore l'architecture matérielle des processeurs. Cette couche d'abstraction permet aux développeurs de créer des applications pouvant s'exécuter directement sur différents types d'ordinateurs puisque la majeure partie des efforts de portage est assurée par la machine virtuelle elle-même.

De nos jours, les langages basés sur des machines virtuelles sont extrêmement populaires. Nous avons déjà mentionné le succès de Java et d'Erlang. Microsoft .Net est un autre exemple célèbre de machine virtuelle largement utilisée aujourd'hui. En outre, on peut noter que chaque navigateur web (Internet Explorer, Firefox, Chrome, Safari) embarque sa propre machine virtuelle, ce qui lui permet d'exécuter du code JavaScript. La souplesse et la dynamicité apportées par le langage JavaScript sont à la base des applications *Web* modernes [112]. Le code client de ces applications internet est ainsi chargé puis dynamiquement interprété par la machine virtuelle embarquée dans les navigateurs. Cela permet une adaptation retardée aux spécificités de la machine hôte.

Comme déjà indiqué, les machines virtuelles permettent le chargement dynamique de code. Mais ce n'est pas le seul mécanisme favorisant l'adaptation dynamique des applications. En effet, de nombreuses machines virtuelles se caractérisent par des capacités de *réflexion* via l'implémentation plus ou moins complète d'un *meta-object protocol* [86] ou MOP.

Un MOP est un framework orienté objet qui permet de connaître et de modifier le comportement d'un système orienté objet. De manière générale, un MOP fournit des opérations de manipulation des objets en cours d'exécution telles que l'envoi de message, l'accès et l'affectation de champs, la déclaration de méthodes, etc. Ces opérations sont en fait définies au niveau des représentations des objets, les méta-objets, d'où le nom *meta-object protocol*. L'adaptation dynamique est plus précisément rendue possible par un MOP aux travers de trois mécanismes : l'introspection, l'auto modification et l'intercession.

D'après Kiczales et ses collègues, un MOP supporte l'introspection s'il permet un accès réflectif en lecture seule à la structure d'un objet. Il supporte l'auto-modification si la modification de la structure de l'objet est possible et enfin il supporte l'intercession si les programmeurs peuvent redéfinir la sémantique des opérations d'un objet particulier. En pratique, l'introspection est supportée par la plupart des langages, l'auto-modification est beaucoup moins répandue tandis que l'intercession est très rarement supportée. Le *Common Object Lisp System* ([21]) est un exemple de langage incluant un MOP supportant ces trois mécanismes. JavaScript supporte intrinsèquement l'introspection et l'auto-modification mais ne fournit aucune API permettant l'intercession. Néanmoins, l'intercession peut être réalisée grâce à l'utilisation de proxy et d'un méta-modèle associé, comme proposé dans [144] pour JavaScript.

L'adaptation dynamique reste donc possible dans une certaine mesure, même sans le support des propriétés d'intercession et d'auto-modification. Ainsi, différentes machines virtuelles fournissent des mécanismes qui permettent d'observer le code s'exécutant (introspection) et de dynamiquement créer des proxies conçus pour intercepter certaines opérations [5]. Ces proxies permettent donc d'ajouter dynamiquement et de manière transparente du code (via la délégation).

Bien que les machines virtuelles soient communément utilisées pour exécuter des applications dynamiques, les mécanismes utilisés restent complexes à manipuler. Des connaissances avancées sont indispensables pour réaliser des adaptations via le chargement dynamique de module ou via des API de réflexion.

3.2.3 Approches de type *Dynamic Software Update*

Les langages de programmation fournissent ainsi des mécanismes permettant l'évolution dynamique de code. Ces mécanismes sont malheureusement complexes à mettre en œuvre et demandent aux développeurs de gérer explicitement les adaptations dans leurs applications.

C'est ainsi qu'ont émergé plusieurs systèmes s'appuyant notamment sur l'édition de

5. La machine virtuelle Java permet la création de tels proxies depuis la version 1.3 de Java au travers de l'API *Dynamic Proxy*.

liens dynamiques pour permettre l'adaptation des applications durant l'exécution. L'adaptation ici n'est permise que par le chargement dynamique d'un module qui s'aligne parfaitement sur une interface prédéfinie et attendue. Ces types de modules sont couramment appelés des *plugins*. Ainsi, pour pouvoir profiter d'un système à plugins, le programmeur doit d'abord identifier l'ensemble des éléments qui sont sujets au changement dans son programme et, dans un deuxième temps, créer une interface commune à ces modules pour qu'ils puissent être utilisés dans le programme. Les systèmes à plugins sont aujourd'hui mis en œuvre dans de nombreuses applications, allant des systèmes d'exploitation jusqu'aux navigateurs web.

Néanmoins, ces frameworks imposent une contrainte forte, à savoir la définition des plugins. Ainsi, si l'on veut pouvoir mettre à jour la totalité d'une application, il faut pouvoir la créer entièrement sous la forme de plugins (modules ou encore composants). Bien que l'idée soit séduisante, il est extrêmement complexe de prendre une application existante et de la décomposer sous forme de modules [74]. En effet, il est alors nécessaire d'ajouter une somme importante de code de liaison, dit *glue code*, pour pouvoir gérer l'ajout et le retrait de plugins à l'exécution. De façon à remédier à cette limitation, et ainsi pour permettre la mise à jour dynamique de tout type d'applications, l'approche appelée *Dynamic Software Update* (DSU) est apparue.

Les approches DSU s'appuient sur des compilateurs particuliers permettant à des applications d'être mises à jour dynamiquement. L'idée développée par cette approche est d'automatiser la modularisation de l'application, permettant ainsi aux développeurs de créer ces applications de manière traditionnelle. Le projet Ginseng [107, 106] est une implémentation de DSU pour le langage C qui permet de mettre à jour des logiciels durant l'exécution. Chaque mise à jour revient à appliquer un « patch ». Le patch est généré en quasi-totalité. Une fois compilé, il est appliqué à chaud grâce au framework qui garantit la sûreté de typage et l'intégrité des données mises à jour. Le projet Jvolve [135] repose sur une machine virtuelle Java qui implémente les techniques de type DSU. Jvolve supporte la mise à jour du code des méthodes ainsi que la mise à jour des classes (l'ajout, le retrait et la modification de type des champs et des méthodes). Bien que ces travaux soit principalement académiques, ils ont été éprouvés et se sont avérés très efficaces sur un grand nombre d'applications largement diffusées telles que des serveurs d'applications [107] ou encore

des systèmes d'exploitations [15].

3.3 Adaptations architecturales

3.3.1 Composants logiciels

L'introduction des composants logiciels dans les années 90 a fourni un nouveau niveau d'abstraction pour les programmeurs. Les approches de développement traditionnelles, orientée objet en particulier, présentent en effet des lacunes importantes : granularité trop fine, peu ou pas de facilités pour les aspects non fonctionnels, pas de gestion des dépendances de code, peu d'aide pour le déploiement, etc. Les approches à composants reposent sur des structures appelées composants qui peuvent être assemblées pour former des applications. Un composant est une unité de composition qui peut être déployée et exécutée de façon indépendante sur une machine [136]. Un composant est aussi caractérisé par des méta-données qui peuvent être utilisées lors de l'instanciation des composants ou lors de leur composition.

De nombreux modèles à composants ont été proposés. Beaucoup sont basés sur la notion d'interfaces fournies et d'interfaces requises et reposent sur différentes méta-données de description. Il n'y a pourtant pas de consensus sur un modèle donné. Les différents modèles définis visent en effet des domaines différents et des propriétés différentes. Cela résulte bien sûr sur structures différentes, des méta-données de description différentes, des mécanismes de composition différents.

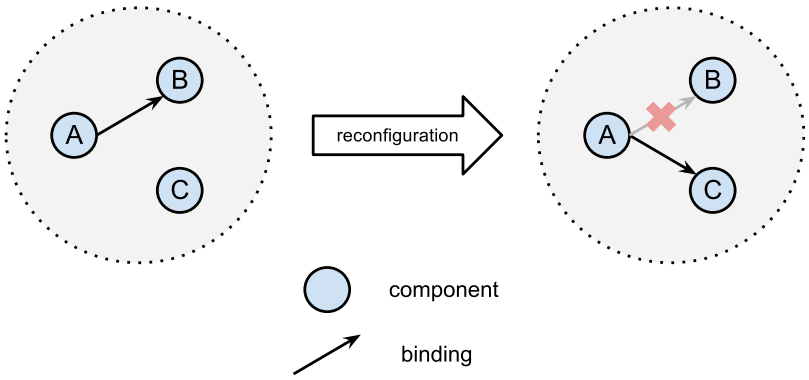


FIGURE 3.2 – Reconfiguration architecturale.

La notion d'adaptation a été largement étudiée par la communauté des composants logiciels. En fait, deux formes d'adaptation peuvent être implantées. Tout d'abord, les composants eux-mêmes peuvent être définis de façon à être adaptables. Cela repose sur des techniques de niveau langage telles que présentées précédemment. Par exemple, un composant peut avoir des capacités d'introspection et des capacités d'évolution à l'exécution.

La seconde forme d'adaptation est appelée reconfiguration architecturale. Ici, l'architecture globale d'une architecture peut être modifiée. Cela peut se concrétiser de diverses façons : modification d'interfaces, remplacement de composants, modification de liens, ajout ou retrait de liens, etc. Ceci est illustré par la figure 3.2.

3.3.2 Composants logiciels et dynamisme

De nombreux modèles à composant ont pour objectif de supporter le dynamisme. Cependant, tous ces modèles ne supportent pas forcément les mêmes types de dynamisme ou ne l'automatisent pas de la même manière.

C2

C2 [98] est un style d'architecture combinant des composants et des communications par messages visant la création d'applications extensible et flexible. C2 est développé et maintenu au sein de l'Université de Californie Irvine (UCI). Depuis la création de C2, plusieurs langages de description d'architecture (ADL) se sont succédés pour permettre aux architectes logiciels de décrire une application C2 sous la forme d'une composition de composant : C2SADL [99], xArch et xADL 2.0 [34]. En plus de ces ADLs, UCI fournit plusieurs framework qui permettent la création d'applications C2 (en Java et en C++).

Dans C2, les composants communiquent entre eux exclusivement via des messages asynchrones. De plus, C2 définit une architecture client-serveur en couche avec des règles de visibilité bien particulière. En effet, les messages de notification vont seulement vers le bas, et les messages de requête de service exclusivement vers le haut. C'est-à-dire qu'un composant ne peut utiliser que des services fournis par les composants d'une couche supérieure. Ainsi, un composant est faiblement couplé aux composants de la couche supérieure

(il peut ignorer les requêtes de service) et non couplé aux composant des couches inférieures (aucune connaissances des messages de notification), ce qui permet d'améliorer le contrôle du système sans perdre les avantages apportés par l'intégration basé sur les évènements.

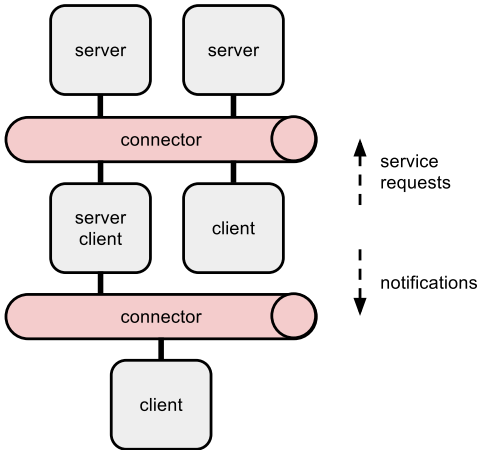


FIGURE 3.3 – Exemple d'architecture avec C2.

Cette architecture particulière promue par C2, couplée à un ADL, rend possible la reconfiguration dynamique de l'application. La spécification Instance d'xADL permet entre autre de modifier la structure d'une application durant l'exécution, en termes de composant, d'interface, de connecteur et de liaison d'instance. xADL fournit aussi la possibilité de décrire des *sub-architectures*, une sous architecture étant une instance de composant ou de connecteur contenant une architecture interne elle-même décrite en xADL.

OpenCom

L'approche OpenCom combine un modèle à composant et un Framework d'exécution réflexif, c'est-à-dire étant lui-même construit avec des composants OpenCom [32]. OpenCom à été créé et développé à l'université de Lancaster, il est aujourd'hui utilisé comme socle de base dans divers projets de développement et académiques, en particulier pour la conception de middlewares adaptifs ([20]).

Les composants OpenCom sont liés grâce aux interfaces qu'ils fournissent et qu'ils requièrent. OpenCom introduit une notion de capsule. Les capsules sont des composites dans lesquels les composants sont chargés, instanciés et composés. Une capsule fournit une API

de contrôle qui permet d'agir sur les composants durant l'exécution. Elle permet entre autre de charger, décharger et instancier des composants, mais aussi de les lier via leurs interfaces. Chaque liaison entre l'interface des composants est représentée par un composant, composants implicitement créé par le framework. Un composant peu être lié à plusieurs composants en tant que fournisseur par interface fournis mais il ne peut être lié qu'à un seul composant par interface requise en tant que consommateur.

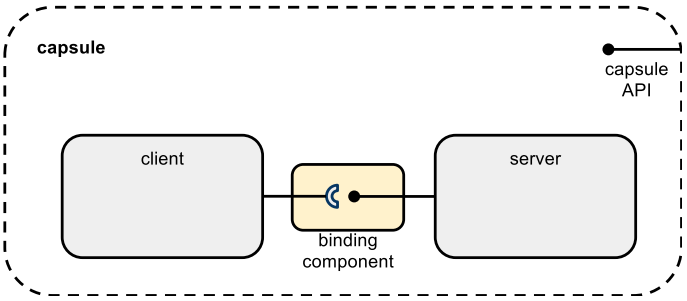


FIGURE 3.4 – Composants OpenCom.

OpenCom définit une plateforme d'extension qui permet par exemple de créer une *capsule* composé de composite s'exécutant sur des environnements d'exécution différents, par exemple des composants C++ et Java. Ces composites particuliers sont appelés des *caplets*. Un intérêt des *caplets* est qu'ils permettent d'isoler des composants dans des processus séparés. Pour simplifier l'adaptation dynamique des applications OpenCom fournit trois méta-model réflectifs réifiés sous forme de composant : le méta-model des interfaces, qui permet de découvrir et d'invoquer de nouvelles interfaces durant l'exécution ; le méta-model d'architecture qui représente la topologie de l'ensemble des composants présents dans la capsule ; le méta-model d'interception qui permet aux développeurs de créer dynamiquement des intercepteurs entre les liaisons des composants.

Fractal

Fractal est un modèle à composant hiérarchique et réflectif [24, 25]. Fractal est un projet open-source hébergé via le consortium OW2, il a été développé conjointement par l'INRIA et France Télécom R&D.

Un composant fractal est constitué de deux parties : une *membrane* et un *contenu*. La

membrane est un contrôleur qui fournit des interfaces permettant l'introspection du composant ainsi que la gestion d'éventuelles préoccupations non fonctionnelles (reconfiguration, sécurité etc.). Le contenu d'un composant réfère à l'implémentation des fonctionnalités. Ainsi, Fractal permet un modèle de développement permettant une réelle séparation des préoccupations.

De plus Fractal distingue deux types de composant, les composants dit *primitif* et les *composites*. Un composant *primitif* est un composant non décomposable servant généralement d'unité de développement mais pouvant aussi être utilisé comme une entité encapsulant du code natif [88] ; l'interface du primitif est visible et peut de ce fait être intégré au sein d'une même application avec d'autres composants. Le type `composite` permet une composition hiérarchique. En effet, un *composite* peut encapsuler des composants primitifs et composite. Dans Fractal, tout type de composant fournit trois type d'interfaces : les *serveurs* qui définissent les services fournis par le composant, les *clients* qui définissent les services requis par le composant et enfin les interfaces de la membrane qui permettent de contrôler les aspects non fonctionnels du composant.

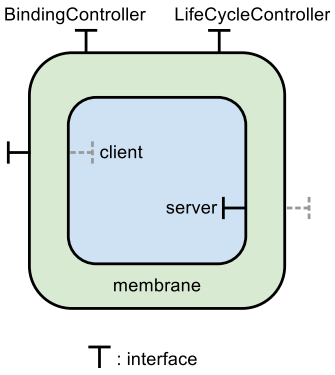


FIGURE 3.5 – Un composant Fractal.

Deux des interfaces de contrôle fournies par la membrane d'un composant permettent la reconfiguration et la composition dynamique d'une application Fractal. En effet, grâce aux interfaces d'introspection et de reconfiguration des liaisons fournies par la membrane du composant il est possible de surveiller et contrôler l'exécution d'un composant mais aussi de reconfigurer ses liaisons durant l'exécution. C'est en s'appuyant sur ces principes et sur le langage de description de Fractal qu'a été proposé FScript, un langage qui permet

et facilite la reconfiguration dynamique des d'architectures Fractal, et donc la recomposition dynamique de telles applications.

Il existe plusieurs framework qui réifient le modèle proposé par Fractal, chaque implémentation se distingue par le langage de développement (Smaltalk, Java, C et C++), mais aussi par les aspects non fonctionnels qu'elles intègrent dont notamment la recomposition dynamique de l'application. Les frameworks les plus populaires sont Julia [6] (pour Java) et Cecilia [7] (pour C).

3.3.3 Composants orientés services

Les composants orientés services introduits par Cervantès and Hall [29] combinent les concepts des architectures orientées services (voir 2.2.5) dans un modèle à composant. En combinant ces deux paradigmes, il devient possible de supporter explicitement la disponibilité dynamique dans un modèle à composant. Les modèles à composant orientés services permettent donc de créer des systèmes modulaires capables de s'adapter durant l'exécution. Les changements à la source de l'adaptation étant modélisés par un changement de disponibilité, c'est-à-dire l'ajout, le retrait ou encore la modification d'un service fourni par un composant. Un modèle à composant orienté service est basé sur les six principes suivants [29] :

1. Un service est une fonctionnalité fournie. Un service caractérise un ensemble d'opérations réutilisables.
2. Un service est défini par un contrat. Le contrat spécifie les caractéristiques du service nécessaires à la composition, l'interaction et la découverte. Cela comprend la syntaxe, le comportement et la sémantique. Pour permettre la composition structurelle, le contrat doit explicitement déclarer les dépendances du service sur d'autres services.
3. Les composants implantent un contrat. Ce faisant, un composant fournit le service spécifié et satisfaisait les contraintes exprimées. En plus des dépendances de services décrites dans le contrat, un composant peut aussi déclarer des dépendances de services additionnelles liées à une implémentation spécifique. Les services sont l'unique moyen d'interaction entre les instances de composants.

6. `http://fractal.ow2.org/julia`
7. `http://fractal.ow2.org/cecilia-site/2.1.0`

4. Le patron d'interaction orienté service est utilisé pour résoudre les dépendances de service. Chaque service fourni par une instance de composant est publié dans un registre. Le registre de services est utilisé pour découvrir les services durant l'exécution et ainsi pouvoir résoudre les dépendances de services dynamiquement.
5. Les compositions sont décrites dans les termes d'un contrat. Une composition est donc un ensemble de contrats qui est utilisé pour sélectionner les composants concrets à instancier. Comparée aux approches à composants classiques, la définition de liaisons explicites n'est pas nécessaire puisque celles-ci peuvent être inférées à l'exécution grâce aux dépendances de services spécifiées par les contrats. La composition est concrétisée durant l'exécution. Les composants qui fournissent les contrats sont découverts et instanciés dans cette composition.
6. Les contrats sont à la base de la substituabilité. Dans une composition, tout composant qui implémente un contrat donné peu être substitué par un autre composant qui implémente le même contrat.

Les modèles à composant orienté services sont importants dans le cadre de cette thèse puisqu'une partie de la proposition à venir repose sur ces principes mais aussi sur des frameworks réifiant ces principes. En particulier OSGi et iPOJO forment le socle technique du framework d'exécution présenté dans la section 4.3. C'est pourquoi nous fournissons ici les détails nécessaires à leur compréhension.

OSGi

OSGi [5] est une technologie très importante dans le cadre de cette thèse puisque une partie de la proposition à venir repose sur cette technologie. Nous fournissons ici les détails nécessaires à sa compréhension. OSGi est à la fois une plate-forme à composants et une plate-forme à services pour le langage Java et suit les principes des composants orientés services énoncés ci-avant, à la nuance près que la gestion des dépendances de services est laissée à la charge des programmeurs. OSGi a pour but de faciliter la modularisation des applications Java ainsi que leur interopérabilité.

La plate-forme OSGi repose principalement sur trois couches en interaction :

– La couche module définit un modèle de modularisation pour Java ;

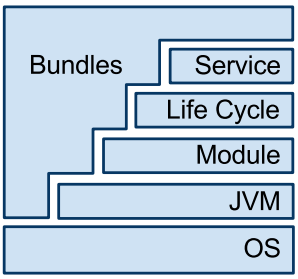


FIGURE 3.6 – Les différentes couches d'OSGi.

- La couche cycle de vie est basée sur la couche module. Elle définit l'API d'administration des bundles ainsi que les différents états dans lesquels peut être un bundle ;
- La couche service définit un modèle de programmation ainsi que les mécanismes pour la conception, le développement et l'exécution d'applications basées sur l'approche à service.

Premièrement, la couche module permet la division des applications Java en modules. L'unité de modularisation OSGi est appelé un bundle. Les bundles adressent quelques unes des faiblesses du modèle de déploiement Java ; un bundle étend les Java Archives Files (JAR) avec un fichier manifest particulier. Le manifest est utilisé pour spécifier les informations statiques liées au bundle, telles que les packages Java partagés ou requis par le bundle. La couche module se sert de ces information pour installer correctement et puis activer les bundles au sein du framework. Dans OSGi, les bundles sont les seuls entités qui permettent le déploiement d'applications basées sur Java.

Comme indiqué, OSGi spécialise la notion d'archive Java pour son contexte d'exécution. De ce fait il réutilise le fichier de métadonnées (manifest) défini par Java pour y inscrire ses propres métadonnées. OSGi définit ainsi un certain nombre de métadonnées (cf. section 3.2.1 de [5]).

La couche de cycle de vie définit un modèle d'exécution pour les bundles. Le modèle spécifie comment les bundles sont démarrés, arrêtés, installés, mis à jour et désinstallés. Fondamentalement, cette couche fournit une API qui permet de contrôler le cycle de vie des bundles. L'API couvre donc l'installation, le démarrage, l'arrêt, la mise à jour, la désinstallation ainsi que le suivi des bundles. La figure 3.7, illustre les différents états que peut avoir un bundle tout au long de son exécution, ainsi que les transitions possibles entre ces

différents états résultant d'une modification du cycle de vie.

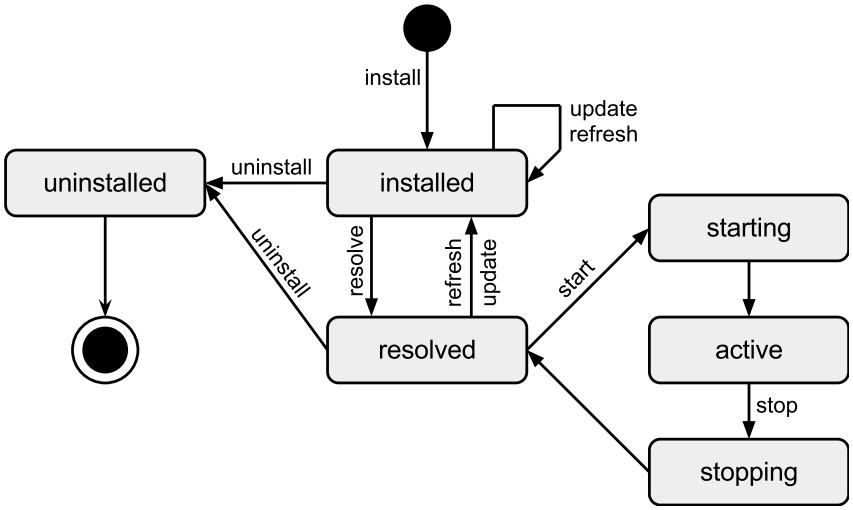


FIGURE 3.7 – Diagramme d'état d'un bundle.

Un bundle est dans l'état *installed* une fois qu'il a été correctement installé. Il passe dans l'état *resolved* si l'ensemble des classes Java requises par le bundle sont disponibles. Le bundle peut alors démarrer et passer dans l'état intermédiaire *starting*. Dans cet état, la méthode callback `start` du `BundleActivator` est appelée et en cours d'exécution. Une fois que le bundle à correctement démarré et terminé l'exécution du callback `start`, il passe dans l'état *active*. Quand le bundle est arrêté, il entre dans l'état *stopping*, la méthode `stop` du `BundleActivator` a été appelé et est toujours en cours d'exécution. Finalement, une fois le bundle correctement arrêté puis désinstallé, il finit dans l'état *uninstalled*.

A noter que le `BundleActivator` d'un bundle est identifié dans son *manifest*. La classe du bundle désigné comme `BundleActivator` doit implémenter une interface éponyme. Cette interface déclare les méthodes `start` et `stop` du bundle qui sont utilisées par le programmeur du bundle pour enregistrer des *listeners* et démarrer tous les *threads* nécessaires. La méthode stop a donc pour responsabilité de nettoyer et terminer tous *threads* encore en cours d'exécution.

Combinées les couches modules et cycles de vie permettent le développement d'application dynamique basé sur l'ensemble des bundles déployés. Un type d'architecture communément utilisé pour supporter ce dynamisme est appelé *extender pattern*. Il est à la base de l'architecture de l'IDE Eclipse. Cependant, l'implémentation de ce motif de conception repose sur des mécanismes *ad-hoc* (fichier particuliers, en-têtes définis dans le manifest), utilisent des mécanismes de bas-niveaux tel que les classloaders des différents bundles, et le dynamisme ne dépend que de l'ensemble des bundles déployés. En général le dynamisme exposé par ces applications est limité et ne permet pas la réutilisation de composant.

Afin d'éviter les limitations de l'extender pattern, OSGi propose une couche service permettant le développement et l'exécution d'applications orienté service. Cette couche permet l'implémentation d'applications exposant un dynamisme bien plus important que précédemment. Elle permet la création de composants qui dépendent de services plutôt que d'autre composants spécifiques. Les services permettent donc de réduire le couplage entre les composants. Cette spécificité rend alors possible un vrai modèle à composants, concis et basé sur la programmation par interface. Les développeurs des bundles peuvent ainsi lier les services en utilisant seulement leur interface de spécification.

En pratique, les services OSGi sont des objets Java classiques qui sont enregistrés avec une ou plusieurs interfaces Java aux travers d'un registre de service. Les composants peuvent non-seulement publiés, recherchés et utilisés des services, mais également être notifié de l'arrivée et du départ de services.

Si la couche service d'OSGi permet d'avoir un harmonieux couplage fonctionnel entre les bundles, en pratique, les bundles sont fortement couplés par leur *packages*. Ce couplage s'avère être une véritable épée de Damoclès au long terme car il peut provoquer l'apparition de problème complexes tels des *dandlings references* [8] et des erreurs d'édition de liens durant l'exécution, en particulier au moment ou les bundles sont installés, mis à jours ou encore désinstallés [55].

Une méthode communément utilisée par les développeurs de la communauté OSGi

8. On pense ici aux problèmes des *stale references* dans OSGi. Ce type de problème arrive quand on détruit un service, mais que ses consommateurs continuent à garder une référence vers lui. Le problème plus général est parfois appelé *dandling references* (notamment en C++).

pour traiter ce type de problème consiste à extraire les packages contenant les interfaces dans des bundles spécifiques contenant seulement les packages regroupant les interfaces des services d'une application. De cette manière, les bundles client et serveur ne sont pas directement couplés entre eux via leurs packages d'interface puisque ces dernières sont fournies par un bundle indépendant. Ces bundles peuvent dont être mis à jour dynamiquement, indépendamment l'un de l'autre, tant qu'il n'y a pas de changement dans les interfaces. Néanmoins, une telle manipulation n'est pas toujours possible et, en pratique, la plupart des bundles ne font même pas usage de la couche service.

De plus, les aspects dynamiques complexes liés à la publication, la découverte et la gestion de la disponibilité des services restent entièrement à la charge du programmeur, ce qui est inévitablement une source importante d'erreurs.

iPOJO

iPOJO [45] est un framework à composants orientés services qui facilite le développement d'applications aux dessus d'OSGi. Comme mentionné ci-dessus, une des principales préoccupations en regard du modèle de développement proposé par OSGi est que la couche service laisse la gestion des dépendances de service entièrement à la charge des développeurs de composants. Cette tâche manuelle complexe est prise en charge par iPOJO qui utilise l'inversion de contrôle pour envelopper un POJO dans un containeur. POJO est un acronyme pour Plain Old Java Object, un POJO est donc un objet java classique. Les composants iPOJO sont ainsi programmés comme de simple objet Java.

C'est le containeur qui va alors gérer l'ensemble des interactions orientées service, c'est-à-dire la publication d'un service, l'instanciation d'un service ou encore la sélection et la découverte de services. Ce modèle permet de réduire la plupart des corvées laissées à la charge du programmeur pour gérer la plupart des tâches liées à la couche service [89]. Programmer un composant iPOJO revient donc à créer une simple classe Java. Les services publiés et requis sont décrit soit dans un descripteur XML, soit avec des annotations.

iPOJO détecte les bundles définissants des composant *iPOJO* et va gérer les services exposés et injecté les services requis. iPOJO gère également la configuration des composants,

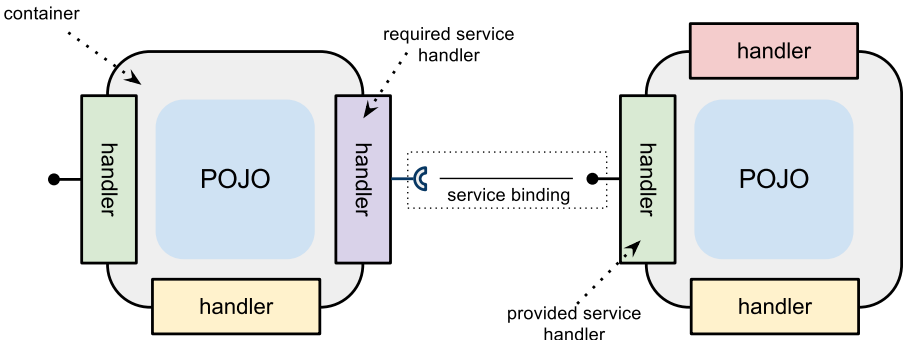


FIGURE 3.8 – Interaction entre deux instances de composants iPOJO via un service.

le cycle de vie et leur introspections. De surcroit, iPOJO fournit des mécanismes d'extension, les *handlers*. Comme indiqué par la figure 3.8, les handlers peuvent être ajoutés au containeur pour apporter différents aspects non-fonctionnels tels que la sécurité, la persistance ou encore l'ordonnancement. Chaque handler gère une propriété non-fonctionnelle particulière.

Il existe trois façons différentes de créer un composant iPOJO, via l'utilisation d'annotations Java, via un langage de description d'architecture, et aux travers d'une API. Le programmeur utilise ces outils pour exprimer les dépendances de services du composant, les services fournis par le composant, les callbacks liés au cycle de vie, mais aussi tout autre aspect lié aux éventuels handlers ajoutés au composant.

Le listing suivant montre comment réaliser un simple composant iPOJO qui fournit un service implémentant l'interface `HelloService`.

```
1  @Component(name="acme.hello.component")
2  @Instanciate(name="helloService") //default instance
3  @Provides //Provide HelloService
4  public class MyComponent implements HelloService{
5
6      @Requires(optional=true) //require a Log Service
7      private LogService logger;
8
9      public String hello(String name){
10         return "Hello "+name+" !";
11     }
12
```

```
13      @Validate //on validation callback
14      private void start(){
15          logger.log(INFO,"HelloService started.");
16      }
17
18      @InValidate //on invalidation callback
19      private void stop(){
20          logger.log(INFO,"HelloService stopped");
21      }
22  }
```

Listing 3.3 – Composant iPOJO fournissant un service `HelloService`.

Le même résultat peut être obtenu en créant le fichier *metadata* suivant à la place des annotations :

```
1  <ipojo>
2      <component classname= "acme.MyComponent" name="acme.hello.component">
3          <provides/>
4          <requires field="logger" optional="true"/>
5          <callback transition="validate" method="start"/>
6          <callback transition="invalidate" method="stop"/>
7      </component>
8      <instance component="acme.hello.component" name="helloService"/>
9  </ipojo>
```

Listing 3.4 – Déclaration d'un composant iPOJO en XML.

Les annotations ou le fichier *metadata* sont analysés à la compilation pour manipuler la classe du POJO et ajouter la spécification du composant iPOJO au fichier *manifest* du bundle. La spécification du composant contient les services requis (`@Requires`), les services fournit (`@Provides`) ainsi que des callback optionnels liés au cycle de vie (`@Validate`, `@InValidate`). Ces informations sont ensuite interprétées durant l'exécution, l'instance du composant est alors créé, les dépendances de services sont dynamiquement injectés et les callbacks appelés en fonction du cycle de vie de l'instance. Le cycle de vie des instances iPOJO est extrêmement simple : une instance est soit valide, soit invalide. Une fois l'instance créée, elle est valide si et seulement si l'ensemble des handlers de son containeur sont valides. En particulier, si le composant requiert un service S de manière non optionnelle, le *handler* de dépendance de services sera valide si et seulement si un service S est présent dans le registre de service. En conséquence, l'instance passera de l'état invalide à l'état valide. De plus, si des callbacks liés au cycle de vie on été spécifiés, ils seront alors appelés lorsque

que l'instance devient valide (*validate callback*) et similairement lorsqu'elle devient invalide (*invalidate callback*).

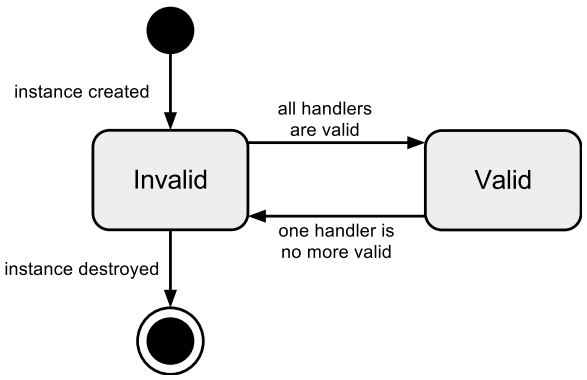


FIGURE 3.9 – Diagramme d'état d'une instance d'un composant iPOJO.

Une des particularités d'iPOJO est qu'il permet la composition hiérarchique, dans ce type de composition les services fournis par les instances de composant peuvent être cantonné dans un composite. Les composites iPOJO permettent ainsi d'isoler les services d'une application du reste des composants présents sur la plateforme.

Similairement au composant iPOJO les composites peuvent être créés de manière déclarative dans un fichier de description. Conformément au sixième principe des modèles à composants orientés services, une composition hiérarchique iPOJO peut aussi être définie en termes de contrat (i.e de services) plutôt que de type de composant.

Le code ci-après présente un exemple de création de composite :

```
1  <ipojo>
2      <!-- Declare a composition -->
3      <composite name= "compositionLambda" >
4          <!-- Instanciate a component implementing the contract acme.HelloService -->
5          <subservice action="instanciate" specification="acme.HelloService"/>
6
7          <!-- Create an instance of the component acme.componentLambda -->
8          <instance component="acme.componentLambda" />
9      </composite>
10     <!-- Create an instance of compositionLamba -->
```

```
11      <instance component="compositionLambda" />
12  </ipojo>
```

Listing 3.5 – Création d'un composite iPOJO.

En conclusion, iPOJO complète OSGi pour accomplir un véritable modèle à composant orienté services. A ce titre, iPOJO à été utilisé dans de nombreux projets qui nécessitait un modèle de programmation facilitant la création d'applications dynamique aussi bien industriel qu'académiques [42]. Les principales limitations d'iPOJO sont héritées des limitations de la plateforme OSGi, elle-même en partie hérité de Java et en particulier des systèmes de class-loaders. Ainsi, il convient en général de suivre un certain nombre de règles pour pouvoir créer un logiciel modulaire qui profitera pleinement des avantages fournit par OSGi et iPOJO [9]. La deuxième limitation, particulièrement importante dans le cadre de cette thèse, est l'aspect centralisé de la plateforme.

3.4 Conclusion

Dans ce chapitre nous avons décrit comment les logiciels peuvent être adaptés durant l'exécution. Nous faisons la distinction entre deux types d'adaptations, les adaptions dites de faible granularité et les adaptations architecturales. Nous avons décrit comment les systèmes d'exploitation, les langages de programmation et les outils ont su évolué pour pouvoir répondre à ce besoin grandissant, l'adaptation des applications durant leurs exécutions. Dans un deuxièmes temps, parallèlement à l'essor du génie logiciel, et aux concepts d'architecture des logiciels, divers approches ont été proposées pour adapter les applications de manière structurelles, notamment grâces aux model à composants orienté services. C'est d'ailleurs en s'appuyant sur des mécanismes d'adaptations de faible granularité que cette adaptation architecturale devient possible. Ainsi, le chargement et déchargement dynamique des modules est indispensable à l'évolution architectural dynamique. De plus, comme illustré par les approches DSU, si ces mécanismes existent, ils sont malheureusement insuffisant, en particulier il n'offre que peu (ou pas) de garantie et nécessite la mise en œuvre d'approche permettant la migration d'état et une meilleur robustesse.

Dans le cadre de nos travaux, nous nous intéressons particulièrement à l'adaptation architecturale. Nous étudions particulièrement comment la distribution d'une application

9. http://www.osgi.org/blog/2011/05/what-you-should-know-about-class.html

peut évoluer durant son exécution tout en réduisant les pertes de disponibilité et sans pour autant impacter la partie métier. Pour ce faire, notre approche repose essentiellement sur les principes introduits par les modèles à composants orientés service. Nous nous concentrons sur des changements dynamiques bien particulier, à savoir :

- le changement de la topologie durant l'exécution,
- l'introduction de nouvelles ressources dans l'environnement,
- l'ajout, le retrait, et la mise à jour de nouveaux protocoles de communication,
- l'ajout, le retrait et la modification d'éléments devant être distribués.

Dans le chapitre suivant nous proposons une solution architecturale et un framework d'exécution permettant d'adresser ces besoins en adaptation pour les applications distribuées faiblement couplées (voir chapitre précédent). Nous revenons dans un premier temps sur les exigences qu'impliquent de telles applications et les stratégies employées pour les traiter. Nous décrivons ensuite notre approche et sa mise en place dans les domaines de l'informatique ubiquitaire et des applications web.

Chapitre 4

Proposition

"Rose is a rose is a rose is a rose" *Gertrude Stein*

4.1 Introduction

Dans la première partie de cette thèse, nous avons présenté un état de l'art sur les problématiques de la distribution et du dynamisme. Dans un premier temps, nous avons défini la notion d'architecture et examiné les différentes technologies de distribution couramment utilisées aujourd'hui, à savoir RPC, Corba, REST et les Web Services. Nous avons mis en avant l'hétérogénéité de ces différentes solutions qui interopèrent difficilement. Nous avons également constaté qu'un des enjeux majeurs de nos jours est le traitement de nouveaux domaines tels que le *Cloud Computing* et l'informatique ubiquitaire qui apportent de nouvelles exigences fortes. Ces nouveaux besoins sont mal négociés aujourd'hui, notamment en termes d'élasticité et de dynamisme.

Ensuite, nous avons examiné la notion d'adaptation logicielle à différents niveaux d'abstraction. Plus précisément, nous avons étudié les techniques les plus utilisées aujourd'hui. Nous avons notamment montré qu'une adaptation peut s'effectuer au niveau du système d'exploitation, au niveau des langages de programmation et au niveau architectural. C'est cette dernière approche qui sera privilégiée dans cette thèse.

Nous avons également montré que de nouveaux domaines apportent des exigences très fortes en termes d'hétérogénéité et de dynamisme. Il est nécessaire dans de nombreux cas

de figure de prendre en compte des ressources volatiles et très hétérogènes au niveau des protocoles de communication. C'est par exemple le cas du Cloud Computing et de l'informatique pervasive où les ressources sont élastiques, volatiles, hétérogènes. Aujourd'hui, le dynamisme et l'hétérogénéité sont mal supportés. De façon plus précise, nous avons fait un ensemble de constatations dans notre état de l'art. Les rares solutions actuelles permettant de marier distribution, hétérogénéité et dynamisme sont :

- **Complexes**. Le code permettant de traiter la communication dans les architectures distribuées, hétérogènes et dynamiques est en général d'une très haute technicité. Il demande de gérer différents protocoles de communication mais aussi de gérer la volatilité des ressources. On pense notamment à la complexité d'utilisation et de configuration associé aux frameworks permettant de créer des web-services.
- **Illusoires** lorsque les problématiques de distribution sont totalement masquées. Ces approches vont souvent trop loin dans la transparence et ne permettent pas de développer des applications conscientes de la distribution et des problèmes particuliers qui y sont associés.
- **Très fortement couplées**. Le code lié à la communication (à l'intégration en général) et le code « métier » sont en général fortement enchevêtrés. Il n'y a pas de réelle séparation des préoccupations dans les architectures actuelles, ce qui soulève clairement des problèmes de correction, de lisibilité, de validation, d'évolution etc.
- **Fermées**. De façon à répondre à la complexité des enjeux, les solutions actuelles sont souvent propriétaires. Elles reposent sur des technologies fermées et sur quelques rares standards comme les services Web (qui, du reste, ne le sont même pas toujours).
- **Peu évolutives**. Toutes les considérations précédentes nous amènent vers des systèmes très difficiles à maintenir et à faire évoluer. Par exemple, dans le monde des serveurs d'applications, l'introduction de nouvelles librairies supportant de nouveaux protocoles de communication nécessite une refonte de l'application. Dans le domaine de l'informatique ubiquitaire, pour les mêmes raisons, les systèmes installés n'évoluent pas ou très peu.

La seconde partie de cette thèse présente notre proposition qui introduit un patron, au sens *pattern*, et un framework logiciel qui répond à la problématique d'adaptation dynamique dans les systèmes distribués hétérogènes. L'approche présentée repose sur les

grands principes du génie logiciel, à savoir la modularité et la séparation des préoccupations. Elle permet de résoudre les problématiques d'intégration entre systèmes hétérogènes, distribués et dynamiques.

Dans le premier chapitre de cette seconde partie, nous rappelons le contexte de notre travail et présentons une vision globale de notre approche. D'abord, nous discutons la problématique et les besoins dans les nouveaux domaines d'application. Ensuite, nous présentons des constats qui guident notre approche ainsi que les objectifs à aborder. Enfin, nous donnons une présentation – en largeur – de notre proposition.

4.2 Problématique

L'objectif de notre travail est de proposer une solution facilitant la conception et l'exécution d'applications distribuées composées d'entités hétérogènes et volatiles capables d'évoluer durant l'exécution. Il s'agit en particulier de permettre aux applications de s'adapter en minimisant l'impact sur la disponibilité mais aussi de répondre à des changements dynamiques (flexibilité des ressources, perte de communication, qualité de service ...) . Ainsi, notre solution s'adresse en particulier au domaine de l'informatique pervasive, des applications web et du *Cloud*.

De façon plus détaillée, l'objectif de notre travail est de définir et d'implanter un environnement logiciel répondant aux exigences suivantes : facilité d'utilisation, extensibilité et résilience. Ces critères sont inspirés des grands principes du Génie Logiciel et nous semblent fondamentaux dans le domaine de l'informatique distribuée. Comme indiqué par la table ci-dessous, nous avons également associé à ces critères un ensemble de défis plus précis.

L'état de l'art a montré qu'il n'existe pas aujourd'hui de consensus sur un style d'architecture favorisant la création d'applications composées d'entités distantes dynamiques et hétérogènes. Les propositions actuelles se concentrent généralement sur un des critères ci-dessus mais aucune ne traite ces critères conjointement. Dans un premier temps, nous détaillons les critères que nous avons retenus puis discutons de notre proposition.

Critère	Objectifs associés
Facilité d'utilisation	Transparence (maitrisée) de la distribution Séparation des préoccupations Automatisation
Dynamisme	Passage à l'échelle *anarchique* Flexibilité / élasticité des applications
Résilience	Simplicité Diversité

TABLE 4.1 – Critères et défis associés.

4.2.1 Facilité d'utilisation

Le tout premier critère est la facilité d'utilisation. L'expérience montre que l'adoption d'une solution logicielle par un informaticien est directement liée à sa facilité d'utilisation [35], qui plus est lorsque le domaine ciblé est complexe. La facilité d'utilisation se traduit au travers des mécanismes mis en œuvre pour diminuer la complexité des tâches de développement. Dans le cadre de notre problématique de recherche, nous avons identifié trois défis qui doivent être adressés : la transparence maîtrisé de la distribution, la séparation des préoccupations et enfin l'outillage.

Transparence de la distribution

Dans le cadre des applications distribuées il est fortement désirable que le programmeur ait accès à une *vue centralisé* du système, c'est-à-dire que le logiciel supportant le développement soit capable de rendre transparente la nature distribuée du système [118, 138, 141]. Ainsi, un programmeur agit comme s'il s'agissait d'un système centralisé, ce qui simplifie considérablement son travail. Néanmoins, il serait illusoire et contre-productif de traiter un système distribué exclusivement comme un système centralisé en particulier pour des aspects tels que les performances et la tolérance aux fautes. On pense notamment aux huit hypothèses généralement faites par les architectes de systèmes distribués et qui s'avèrent être complètement fausses sur le long terme [36, 82].

Séparation des préoccupations

La séparation des préoccupations est une notion déjà ancienne en informatique, mise en évidence en 1974 par Dijkstra [39]. Le principe est de séparer différents aspects d'un

programme, appelés préoccupations, lors de la conception, voire même de l'exécution. La séparation des préoccupations peut se concrétiser grâce à la programmation orientée aspect [85] (AOP en anglais) ou simplement en utilisant la modularité. Dans notre cas, il est intéressant de considérer la distribution comme un aspect particulier et de la traiter de façon indépendante par rapport au code métier. Cela permet en effet de réduire le couplage entre différents éléments d'une application distribuée et favorise la lisibilité, la validation et l'évolution.

Automatisation

L'outillage est un aspect fondamental pour l'adoption de nouvelles technologies. Il permet de simplifier le travail du développeur en effectuant certaines tâches de manière automatique. Le succès des environnements de développement tels qu'Eclipse [1] par exemple, et les nombreux plugins qui les accompagnent montrent à quel point un outillage bien pensé facilite la conception d'applications logicielles. En ce qui concerne les applications distribuées, il peut s'agir, par exemple, d'automatiser la génération des connecteurs ou encore des descriptions de connecteurs, aussi bien de manière statique que dynamique.

4.2.2 Dynamisme

Les domaines que nous abordons dans cette thèse sont caractérisés par un fort dynamisme, c'est-à-dire qu'ils doivent faire face à de nombreuses évolutions durant l'exécution. Dans le cadre de notre problématique de recherche, nous avons identifié deux défis : le passage à l'échelle et la flexibilité de la solution logicielle elle même.

Passage à l'échelle anarchique

Une propriété remarquable d'Internet est la notion de passage à l'échelle *anarchique* [47]. La plupart des logiciels sont ainsi conçus avec l'hypothèse implicite que le périmètre du logiciel est sous contrôle et qu'il n'est pas impacté par des évolutions externes. Cette hypothèse est doublement fausse à l'époque d'Internet et de la multiplication des réseaux privés. Sur ces réseaux, les ressources disponibles évoluent de façon incontrôlée et les entités hébergées peuvent profondément impacter le logiciel considéré. Ainsi, la notion de

1. `http://www.eclipse.org`

passage à l'échelle anarchique fait référence à la nécessité de concevoir des logiciels capables de fonctionner alors qu'ils sont sujets à des charges imprévues et discontinues, à des pertes de communications, à des possibilités nouvelles de communication, etc. Une telle propriété s'applique à tous les éléments architecturaux constituant un logiciel. Ainsi, on ne peut pas faire l'hypothèse que les clients maintiennent une connaissance de tous les serveurs. De la même manière, on ne peut pas s'attendre à que les serveurs gardent une connaissance des états entre différentes requêtes.

Flexibilité/Élasticité

L'élasticité, aussi appelée flexibilité, peut être définie comme la capacité d'ajouter ou de retirer des fonctionnalités à un système existant. L'élasticité durant l'exécution fait référence à cette même propriété mais durant l'exécution. Comme nous l'avons vu tout au long du chapitre 3, des modifications peuvent être apportées à différents niveaux d'abstraction. De nombreux langages de programmation permettent, dans une certaine mesure, d'ajouter ou de retirer des modules durant l'exécution (voir 3.2.2). Néanmoins cette capacité est souvent limitée et est la source de nombreux maux. D'autres approches se focalisent sur la phase de compilation ou au niveau des machines virtuelles pour appliquer dynamiquement des patchs à des programmes. Parallèlement, des travaux basés sur des modèles d'architecture autorisant des changements d'architectures durant l'exécution on fait leur apparition. Ces travaux cherchent souvent à réduire le couplage entre les composants, via l'utilisation de communications à base d'évènements par exemple. Ici, l'élasticité se traduit par l'ajout, le retrait, le remplacement ou la reconfiguration de composant ou de connecteur durant l'exécution.

4.2.3 Résilience

La tolérance aux fautes, c'est-à-dire la capacité à fonctionner en dehors des spécifications, est une propriété essentielle en logiciel. Bien sûr, il est rarement possible de garantir une couverture totale des fautes et erreurs qui peuvent intervenir. Une telle couverture est d'ailleurs illusoire quand on aborde les systèmes distribués qui, encore une fois, sont caractérisés par l'échelle des réseaux sur lesquels ils se situent. D'ailleurs, l'enjeu pour des applications qui évoluent sur ces infrastructures réseaux complexes et en constante évolution n'est pas de garantir une robustesse à toute épreuve mais plutôt de pouvoir fournir

des services auxquels on peut faire confiance [8], et ce malgré les changements apparaissant durant l'exécution [92]. Dans le cadre de notre problématique de recherche, nous avons sélectionné deux défis : la simplicité et la diversité.

Simplicité

La simplicité est la qualité de rester simple. Richard P. Gabriel argumente dans le désormais célèbre *Worse is Better* [53] que la qualité n'augmente pas nécessairement avec le nombre de fonctionnalités. Il défend notamment l'idée que la conception doit rester simple aussi bien pour les interfaces que pour les implantations. Plus généralement, le principe de conception KISS, pour *Keep It Simple Stupid* , énoncé pour la première fois par Kelly Johnson alors ingénieur chez Lockheed Skunk Works met l'accent sur la simplicité qui doit être un des objectifs principaux de toute conception. Il s'agit d'éviter toute complexité inutile qui rend les systèmes obscurs, difficiles à comprendre et qui augmente ainsi les risques d'introduire des erreurs au cours du temps. Dans notre cas, il est crucial que notre solution soit simple au niveau des interfaces de programmation fournies [73], de la quantité et pertinence des fonctionnalités et de la généralité des éléments architecturaux [65]. Ici la généralité implique que chaque élément architectural doit répondre à un problème général, ce qui augmente la réutilisation et diminue le nombre d'éléments.

Diversité

Les applications distribuées que nous souhaitons aborder évoluent dans des écosystèmes extrêmement complexes ou l'hétérogénéité et la diversité sont naturellement présentes. Cela signifie que les entités qui apparaissent ou disparaissent sur le réseau sont potentiellement très différentes. Souvent, elles utilisent des protocoles de communication différents et expriment la connaissance de façons diverses. Cela est particulièrement vrai sur Internet où les problèmes de divergences sémantiques sont particulièrement difficiles à résoudre. Cette hétérogénéité n'est pas totalement négative cependant. La diversité permet d'éviter que des vulnérabilités dans une des composantes de l'écosystème deviennent un point de défaillance unique. Il est donc essentiel de pouvoir s'accommoder de cette diversité. Une solution logicielle visant à supporter des architectures distribuées dans ces domaines doit donc pouvoir prendre en compte cette hétérogénéité, au même titre que le dynamisme des entités à considérer.

4.3 Proposition : RoSe

4.3.1 Vision globale

Dans le cadre de cette thèse nous avons conçu et développé un framework logiciel spécifique à la distribution, nommé RoSe. Ce framework permet d'importer et d'exporter des ressources de façon automatisée, non intrusive et de manière dynamique. Le framework RoSe repose sur les grands principes suivants :

- La séparation des préoccupations : RoSe s'occupe exclusivement de l'intégration et de la publication de ressources et laisse le code applicatif traiter les aspects *métier*. Contrairement à d'autres approches, RoSe n'intervient pas dans le code applicatif.
- L'extensibilité : RoSe a été conçu et implanté de façon à pouvoir être adapté. Il fournit entre autre des facilités (APIs) pour ajuster les technologies de communication utilisées.
- La dynamicité : toutes les extensions/modification prévues dans RoSe peuvent être réalisées à chaud. Ceci est dû en partie à l'utilisation de technologie des composants orientés service, à savoir iPOJO. Cette approche associe modularité et flexibilité à tous niveaux du framework.
- L'utilisation de standards : RoSe intègre de nombreux frameworks de communication issus des technologies Internet, comme CXF et Jersey par exemple.
- Les patrons de conception : RoSe met en œuvre différents patrons issus du domaine de la programmation orientée service et de la programmation distribuée, tels que l'inversion de contrôle par exemple.

Il en résulte un framework modulaire, homogène, extensible et capable de prendre en compte la dynamicité de son environnement. Il permet également la modification à chaud de nombreux éléments internes pour permettre une plus grande souplesse. La version de référence de RoSe est implantée en Java/iPOJO et est disponible sous la forge OW2. Il existe également un prototype en JavaScript utilisé avec succès pour la mise en place de clients riches sur Internet [11].

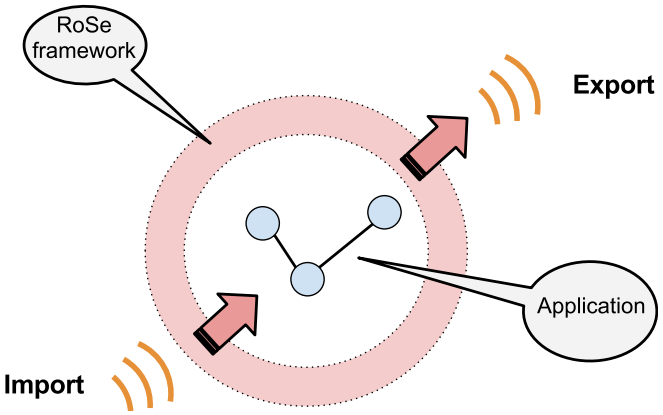


FIGURE 4.1 – Le framework RoSe.

4.3.2 L'import de ressources

RoSe prend en charge l'import de ressources hétérogènes et volatiles au sein d'une application. RoSe est ainsi capable de scruter l'environnement d'une application suivant différents protocoles et selon un filtre de recherche défini par le programmeur. Une ressource est réifiée sous la forme d'un proxy qui est vu comme un service local. Ce proxy peut être augmenté de métadonnées lors de sa création. RoSe suit le cycle de vie des ressources : il ajoute ou retire les proxies en fonction de leur disponibilité. RoSe est conçu pour être modifiable de façon dynamique :

- Les spécifications des ressources recherchées et les contraintes associées (filtre, scope, protocoles utilisés) sont reconfigurables à chaud.
- Des drivers de découverte peuvent être ajoutés et supprimés à tout moment.

L'un des grands principes de RoSe est de ne pas interférer avec les codes applicatifs. Ainsi, RoSe donne les moyens d'insérer des ressources externes sans couplage fort avec le code applicatif. De façon plus précise, l'approche de RoSe lorsqu'il découvre une nouvelle ressource est d'insérer un proxy au niveau applicatif. Ce proxy devient disponible au sein de l'application considérée et c'est elle qui choisit de l'intégrer ou pas. Comme tout proxy, il sert à invoquer directement la ressource découverte sans plus passer par le framework qui l'a créé.

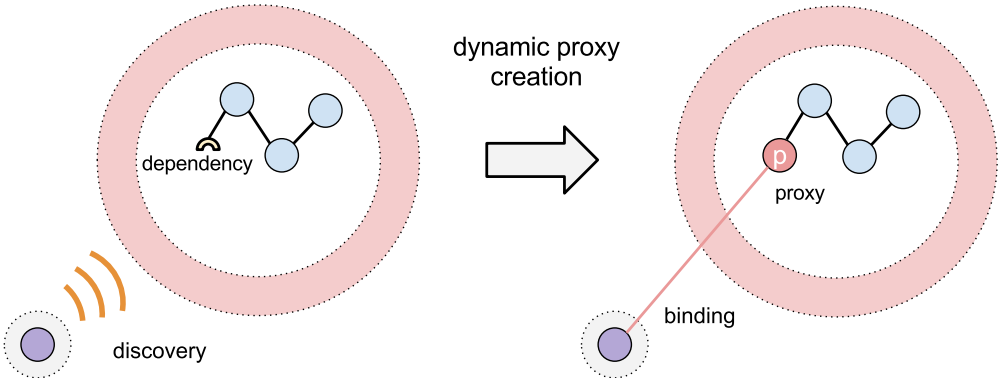


FIGURE 4.2 – Import de ressource dans RoSe.

Les proxies insérés par RoSe peuvent être générés, comme dans le cas des services Web par exemple. La plupart du temps, pourtant, les proxies sont préparés à l'avance et instanciés lors de la découverte des ressources associées. En ce sens, RoSe s'inscrit plus naturellement dans les approches – domaines – où les ressources potentielles sont définies statiquement. Pour une même ressource, néanmoins, plusieurs proxies avec des qualités de service différentes peuvent être préparés au sein d'un domaine. La gestion des proxies à l'exécution est entièrement dynamique : la modification des drivers disponibles, des protocoles utilisés, des filtres entraînent une nouvelle génération des proxies.

4.3.3 L'export de ressources

RoSe permet également d'exporter des ressources de façon dynamique, selon des formats hétérogènes et avec une description contenant des métadonnées. Par exemple, les ressources publiées peuvent être prendre la forme d'une représentation REST, RPC ou d'un service Web. RoSe permet des expositions simultanées de la même ressource suivant des technologies différentes et garantit la pertinence de l'information présentée. Cela signifie que l'information publiée par RoSe correspond bien à celle qui est manipulée par le code. RoSe est conçu pour être configurable et extensible :

- Les ressources à exporter, leur format et les protocoles utilisés sont reconfigurables à chaud.
- Les serveurs de communication peuvent être ajoutés et supprimés également à tout moment.

L'approche de RoSe est de ne pas lier le code applicatif et les publications. Le programmeur doit simplement spécifier qu'une ressource doit être publiée suivant certaines conditions. RoSe prend alors en charge cette exposition. Précisément, le framework crée un *endpoint*. Ce dernier est un élément concret d'exposition, un objet virtuel permettant l'accès de la ressource à distance (c'est-à-dire via un service Web ou une représentation REST par exemple).

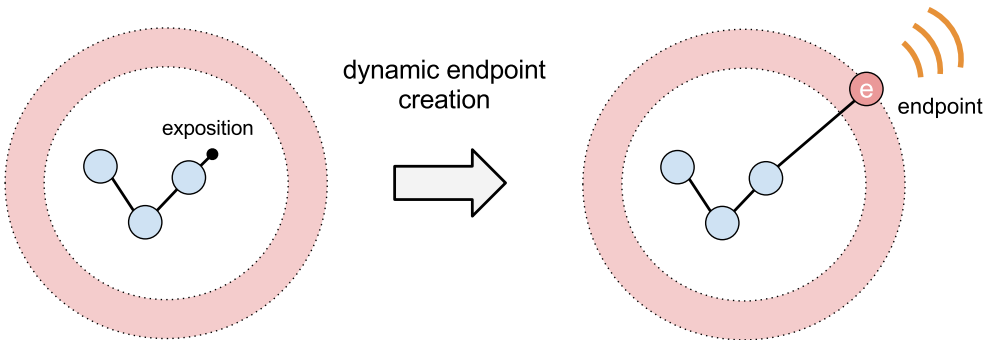


FIGURE 4.3 – Export de ressource dans RoSe.

Les endpoints peuvent être entièrement générés par RoSe, notamment pour les services Web, JSON-RPC ou encore XML-RPC. Les programmeurs peuvent ainsi choisir les frameworks assurant les générations. Ils peuvent également annoter leur code en suivant des standards Java tel que Jax-RS et Jax-WS, ce qui permet d'affiner la génération des endpoints. Sans l'usage des annotations, les endpoints sont générés de façon *brute* en exposant toutes les méthodes publics d'une classe.

Les endpoints sont générés uniquement lorsque les ressources à exposer apparaissent et ils sont détruits dès lors que les ressources disparaissent. Également, la modification des serveurs disponibles, des frameworks utilisés, des configurations entraînent une nouvelle génération des endpoints. Cela signifie par exemple que l'on peut passer dynamiquement d'une exposition en service Web vers une exposition REST (ou encore les deux).

4.3.4 Applications distribuées

Les capacités d'import et d'export de RoSe permettent de construire des applications distribuées où les communications sont en grande partie automatisées. Dans cette optique,

tous les composants distribués d'une application intègre le framework RoSe. L'idée est alors de générer les proxies et les *endpoints* nécessaires à la communication dynamique entre les composants distribués. La génération étant homogène pour tous composants, la correction et d'éventuelles qualités non fonctionnelles sont ainsi assurées. Cela libère en outre complètement le programmeur de la gestion de l'aspect *distribution* de son application répartie.

Utiliser RoSe sur différentes machines d'exécution permet ainsi la création automatique des proxies et des enpoints. Cette distribution 'automatique' s'appuie sur des protocoles de type RPC, services Web mais aussi REST pour les communications entre endpoint et proxies et sur des annuaires distribués tels que zookeeper, dns-sd, pubsubhubbub pour la découverte des descriptions échangées entre passerelles. Ainsi, pour chaque endpoint créé, RoSe va automatiquement publier une description du endpoint contenant les informations nécessaires à la génération d'un proxy. Les machines intéressées vont donc être notifiées grâce à cette description de la disponibilité du endpoint et donc pouvoir générer un proxy pour cet endpoint. En effet, dans ce cas particulier où RoSe est à la fois du côté client et serveur, il est aisé de générer des endpoints et proxy associé en s'appuyant sur des standards tel JSON-RPC ou JAX-WS.

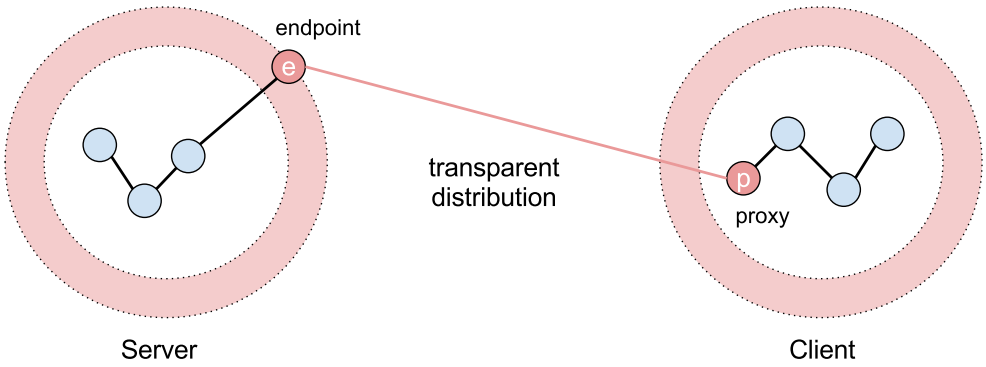


FIGURE 4.4 – Automatisation de la distribution.

Grâce à RoSe, l'administrateur pourra spécifier de manière déclarative et durant l'exécution quels éléments de l'application doivent être distribués et avec quel protocole. Il pourra également ajouter d'éventuelles propriétés non fonctionnelles aux descriptions des endpoints qui sont publiées. Similairement, RoSe permet de spécifier quels sont les services distants qui doivent être réifiés sur la machine. Ainsi, une fois cette configuration donnée, la distribution entre différentes machines d'exécution hétérogènes peut être gérée de

manière transparente par RoSe. Il existe certaines limitations néanmoins. Premièrement, chaque machine doit intégrer le framework RoSe [2]. Ensuite, les différents frameworks RoSe doivent êtes liés à un registre distribué commun et finalement les machines doivent intégrer des protocoles communs deux à deux.

Néanmoins, lorsqu'une application doit être distribuée, il est recommandé de développer les ressources devant être distribuées en étant pleinement conscient de cette exigence. C'est pourquoi nous nous appuyons notamment sur les annotations pour permettre aux développeurs d'affiner la génération des endpoints et des proxies. Bien sûr, cela diminue la transparence mais cela permet de mettre en évidence certaines contraintes liées à la distribution. L'avantage de cette approche mixte est qu'elle permet aux développeurs de traiter spécifiquement les aspects liés à la distribution (sérialisation, passage par référence, latence etc..) tout en profitant de la flexibilité et de la modularité introduite par le modèle a composant orienté service sous-jacent. Cela, sans être couplé, ni à avoir à configurer un framework de communication ou une librairie particulière.

Utiliser RoSe pour la conception et l'exécution d'applications distribuées apporte une très grande flexibilité. Un composant de l'application (offrant des services) peut être exposé suivant différents protocoles simultanément et d'autres composants de l'application peuvent choisir et faire évoluer la meilleure façon de communiquer avec ce composant serveur. Les adaptations peuvent se faire à chaud et de façon totalement transparente. Par exemple, la communication entre deux composants peut évoluer à chaud en utilisant un nouveau protocole, plus en phase avec les besoins du moment.

Cette propriété est aujourd'hui particulièrement intéressante dans le domaine du Cloud Computing ou les ressources disponibles sur le Cloud évoluent fréquemment, aussi bien en termes de capacité que de protocoles utilisés. RoSe permet ainsi à une application de communiquer avec une partie Cloud tout en s'abstrayant de la partie purement liée à la communication et au dynamisme des moyens mis en œuvre.

2. Pour le moment ils existent une version de RoSe pour les plateformes OSGi et pour les machines virtuelles JavaScript.

RoSe permet aussi de gérer de façon transparente le déplacement de ressources sur différentes machines. Par exemple, si une ressource utilisée par une application est déplacée, RoSe détruira le proxy utilisé pour la communication et créera un nouveau proxy lorsque la ressource déplacée sera publiée.

4.3.5 Langage de spécification

RoSe propose une approche déclarative permettant de spécifier d'une part les ressources devant être importées et d'autre part les ressources devant être publiées automatiquement. RoSe permet d'exprimer un ensemble de contraintes sur les ressources visées, les protocoles à mettre en place, les framework de communication à utiliser, etc. La grammaire est présentée ci-dessous.

Une spécification est stockée dans un fichier sur la machine d'exécution. Il peut y avoir plusieurs spécifications pour une même application ; cela signifie que plusieurs éléments logiciels peuvent être exportés et importés suivant des modalités différentes. RoSe prévoit ainsi le fait qu'il peut y avoir plusieurs administrateurs ou tout simplement plusieurs applications indépendantes qui nécessitent d'être distribuées ou de communiquer avec des services distants. Par ailleurs, tous les éléments spécifiés grâce au langage de RoSe sont modifiables à l'exécution, ce qui permet d'adapter les traitements au contexte d'exécution, qu'il soit interne ou externe. Pour cela, il suffit de changer le ou les fichiers de spécification.

Machine

Machine

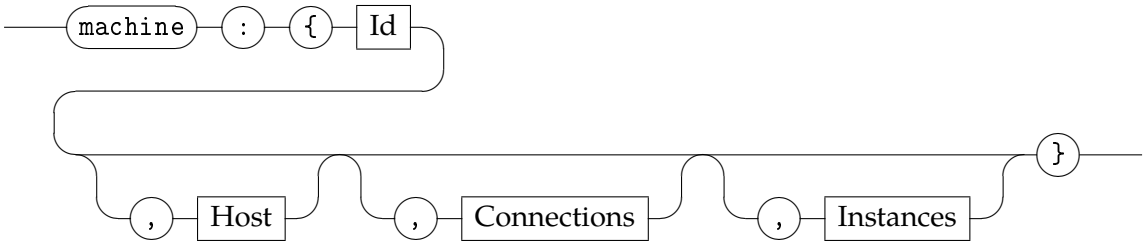
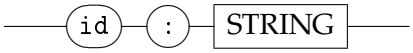


FIGURE 4.5 – Diagramme syntaxique d'une machine.

Dans un premier lieu, il convient de déclarer une machine RoSe. Une machine est spécifiée par un Id (`Id`), et trois configurations optionnelles : un nom d'host (`Host`), une liste de

connexions (`Connections`) et une liste de déclarations d'instance de composant (`Instances`).

Id



Host

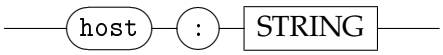


FIGURE 4.6 – Id et Host.

L'Id d'une machine est son identifiant, il s'agit d'une chaine de caractères. L'Id sert donc à identifier la machine sur l'ensemble des réseaux auxquels elle est reliée. Deux machines présentes sur un même réseau doivent avoir un Id différent. Le Host d'une machine est un second identifiant, il s'agit de l'identifiant de la machine sur le réseau (son nom de domaine par exemple).

Connections

Connections

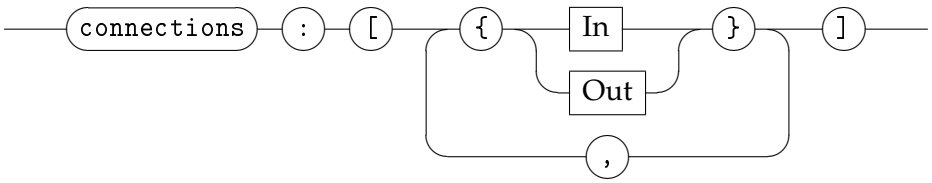


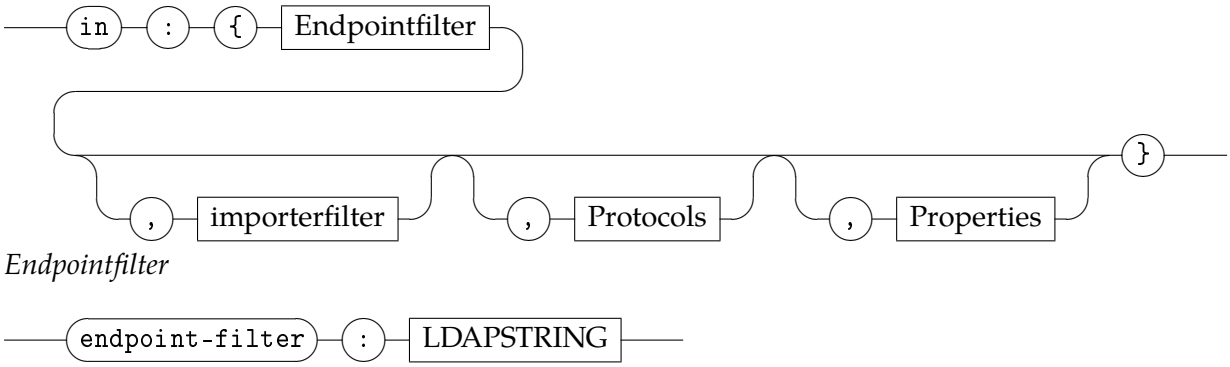
FIGURE 4.7 – Diagramme des connections.

Les connections sont une liste de configuration d'import de ressources distantes (`In`) et d'export de ressources locales (`Out`). Ainsi, chaque ressource distante correspondant à une configuration In sera réifiée sous la forme d'un proxy dans l'environnement d'exécution local. Réciproquement, chaque ressource locale correspondant à une configuration Out sera exposée au travers d'un Endpoint.

Pour spécifier une connexion In, il faut spécifier un filtre LDAP[3] (`EndpointFilter`).

3. LDAP pour *Lightweight Directory Access Protocol*, est un protocole permettant l'interrogation et la modification de service d'annuaire. Nous nous intéressons particulièrement à la syntaxe des filtres LDAP qui permettent d'exprimer des contraintes de premier ordre sur un tableau associatif de propriétés.

In

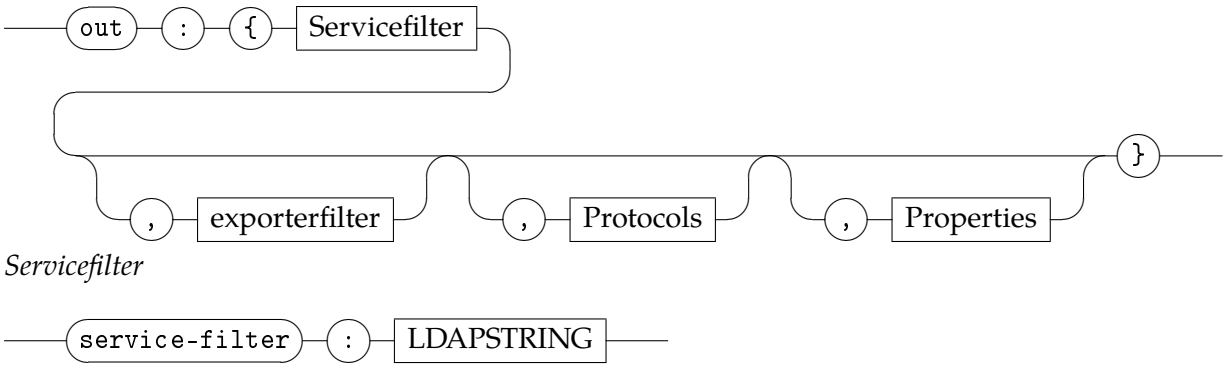


Endpointfilter

FIGURE 4.8 – Connection In et EndpointFilter.

Ce filtre est utilisé pour effectuer la sélection des ressources distantes à importer. En effet, chaque ressource distance dont la description correspond à ce filtre sera réifiée. La seconde propriété à spécifier est aussi un filtre LDAP, mais sur les descriptions des *importer* à utiliser pour créer les proxies. Pour des raisons pratiques, on peut aussi spécifier la liste de protocoles à utiliser (`Protocols`). Chaque composant capable de créer un proxy peut aussi être identifié par son protocole. Finalement, l'administrateur peut spécifier des propriétés (clef-valeur) qui seront rajoutées aux propriétés du service fourni par le proxy. RoSe interprète donc chaque connexion In et s'en sert pour traquer les ressources distances qui doivent être réifiées selon cette configuration.

Out



Servicefilter

FIGURE 4.9 – Connection Out et ServiceFilter.

Parallèlement aux connexions In, les connexions Out permettent de spécifier une configuration d'export de ressources locales. Ainsi, tous les services locaux dont les propriétés *match* le filtre (`ServiceFilter`) seront exposés aux travers d'Enpoints créés par les *exporter* qui *match* le filtre (`ExporterFilter`) et qui supportent les protocoles présents dans la liste (`Protocols`). Les propriétés spécifiées (`Properties`) sont ajoutées à la description des Endpoints créés.

Protocols

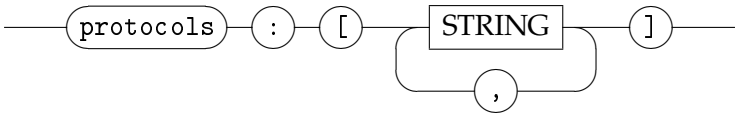
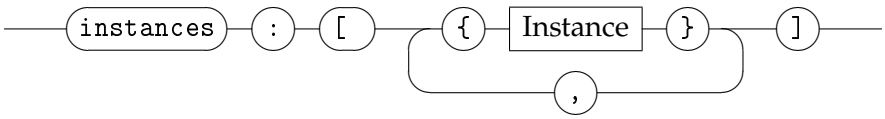


FIGURE 4.10 – Syntaxe de la liste des protocols.

Instances de composants

Instances



Instance

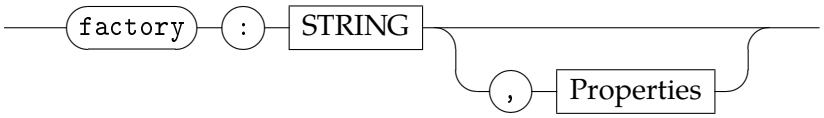


FIGURE 4.11 – Syntaxe de déclarations d'instances.

Finalement, il est possible de créer des instances de composant via la configuration RoSe. On peut par exemple créer les instances des composants du framework, tels les Exporter et les Importer ou encore des instances de composants fournissant des ressources destinées à être exposées.

Le fichier de configuration est interprété durant l'exécution et résulte en l'instanciation de composants du Framework et la création au travers de l'API RoSe de managers contrôlant l'import et l'export de services en accord avec les configurations In et Out. Chaque modification du fichier résulte en la suppression, la reconfiguration ou la création de ces entités logicielles. Nous décrivons, dans le chapitre suivant, les éléments qui constituent le Framework RoSe, leurs interactions ainsi que l'API java. La grammaire EBNF du langage est fournie dans l'annexe A.

L'exemple suivant correspond à une configuration minimale de la machine :

```
1  {
2     machine : {
3      id : "server1",
4      host : "localhost",
5      connections : [
6          { out : {
7                  // tous les services qui implantent l'interface HelloWorld
8                  service_filter : "(objectClass=rose.example.jaxrs.contract.HelloWorld)" ,
9                  // ajout de metadata dans la description d'export
10                 properties : {"tag" : ["example","helloworld"] }
11             }
12     }]
13 }
```

Listing 4.1 – Exemple d'une configuration RoSe.

4.3.6 Principes

La conception et l'implantation de notre framework ont essentiellement été guidées par les principes issus du Génie Logiciel suivants : la modularité, la séparation des préoccupations, l'inversion de contrôle et la décentralisation. Nous examinons ici ces aspects fondamentaux pour RoSe.

Modularité

Concernant la modularité, notre approche repose sur les composants orientés service [29], ce qui permet de conjuguer modularité et flexibilité au sein d'un même modèle à composants. Pour rappel, la composition suivant les composants orientés service est fondée sur

la notion de contrat. Toutes les dépendances sont automatiquement résolues à l'exécution, ce qui permet des compositions et recompositions continues. Un composant peut ainsi être dynamiquement substitué à un autre s'il implante le même contrat. Un modèle à composant orienté service permet aux développeurs de constituer des applications modulaires grâce à la notion de composants. Il permet également de mettre en place un couplage faible par la séparation des interfaces et implémentations.

Les modèles à composants orientés services actuels, tels que iPOJO, ne sont néanmoins pas distribués. Nous avons ainsi été amenés à étendre le modèle proposé par [29] pour traiter les problèmes propres aux systèmes distribués. Selon notre approche, chaque machine comprend un modèle à composant orienté service disposant d'un registre local où sont publiés les services locaux. Une machine peut donc être vue comme un composite, c'est-à-dire un ensemble de services.

Séparation des préoccupations

La séparation des préoccupations est au cœur de RoSe. Au niveau de son utilisation, nous avons vu que les configurations liées à la distribution sont faites de manière déclarative et sans impacter le code métier. Au niveau de la conception interne, RoSe isole clairement les différents aspects liés à la distribution. Ainsi, les structures de découverte, de configuration et de communication sont clairement séparées au sein du framework. Dès lors, un programmeur n'a pas à se soucier des aspects liés à la distribution et au dynamisme structurel au sein de son code métier. Les registres réseaux liés aux découvertes sont également des composants indépendants et permettent de traiter le dynamisme, c'est-à-dire l'apparition et la disparition de machines (ou plus généralement de services) au cours de l'exécution. La notion de connecteurs permet de traiter les problèmes spécifiques à la communication tels que la sérialisation ou la latence. Les connecteurs peuvent se contenter d'implanter des standards pour garantir une transparence de la communication ou les spécialiser afin d'apporter de la qualité de service supplémentaire.

Inversion de contrôle

Les techniques d'inversion de contrôle permettent de clairement séparer l'utilisation de services et leur configuration. En particulier, l'injection de dépendance évite aux programmeurs de spécifier dans le code métier les dépendances dynamiques d'un composant. Ici, un composant doit simplement déclarer les services requis et le framework d'injection de dépendance prend en charge leur fourniture à l'exécution. Martin Fowler [4] présente l'injection de dépendance de la façon suivante : si un objet de type A dépend d'un objet de type B, alors A doit le déclarer à un système d'inversion de contrôle, et non pas appeler lui même un constructeur de B. Le système d'Inversion de Contrôle peut alors injecter une instance de B dans l'instance de A qui en a besoin, et ce durant l'exécution. L'utilisation de framework d'inversion de contrôle, en particulier d'injection, est populaire dans le monde Java (Spring [5], Google Guice [6], iPOJO [42]).

L'inversion de contrôle apporte beaucoup de flexibilité, notamment par rapport aux variations dans l'environnement d'exécution [13]. C'est pourquoi nous avons décidé de spécifier de manière déclarative les liaisons entre machines et d'utiliser le principe d'inversion de contrôle pour instancier les connecteurs dynamiquement, selon la configuration et l'environnement d'exécution. L'assemblage des composants est traité comme un aspect et donc séparé du code métier. On peut ainsi aisément substituer différentes configuration pour différents assemblages.

Utilisation du patron *factory*

Pour s'adapter aux différentes formes d'évolutions et à l'hétérogénéité, nous nous appuyons sur les travaux proposés par [51, 54, 71]. Ici, interopérabilité et adaptation sont rendues possibles grâce à la génération ou l'instanciation de proxies à l'exécution. Dans notre architecture, nous parlons de connecteurs plutôt que de proxies. Nous distinguons deux familles de connecteurs qui doivent être instanciés pour permettre une liaison entre des machines : les connecteurs côté client (appelé proxies) et les connecteurs côté serveur (appelé endpoints). Afin que les connecteurs puissent être aisément réutilisés et instanciés dynamiquement via des mécanismes d'inversion de contrôle, chaque connecteur dispose

4. http://martinfowler.com/articles/injection.html
5. http://www.springsource.org/
6. http://code.google.com/p/google-guice/

d'une *factory* [56]. Le framework utilise cette factory pour créer une instance du connecteur avec une configuration spécifique, propre à chaque protocole et potentiellement au contexte d'exécution. Un point important est que l'ajout ou le retrait de ces factories de connecteurs doit être possible à tout moment durant l'exécution et cela sans impacter directement les applications de manière à permettre l'évolution du framework et des applications.

De façon originale, nous n'imposons pas une architecture centralisée mais bien une architecture décentralisée, où chaque machine doit pouvoir instancier les connecteurs selon ses besoins. De la même manière nous encourageons l'apparition de machines faisant office de passerelle entre différents types de réseaux. Cette architecture décentralisée correspond mieux à la problématique de passage à l'échelle anarchique propre au web et aux environnements ubiquitaires.

4.4 Exemples d'applications

4.4.1 Introduction

En premier lieu, le framework RoSe a été appliqué à l'informatique pervasive et plus particulièrement aux applications de domotique ou de gestion intelligente des bâtiments. Ces applications se caractérisent par de fortes contraintes de distribution et d'hétérogénéité des ressources. Le développement et l'exécution d'applications pervasives nécessitent de communiquer avec différents types de dispositifs (télévision, capteurs divers, compteur de gaz et d'électricité etc.) possédant leurs propres formats et protocoles de communication. Il n'existe pas dans ce domaine de standard universel permettant de s'abstraire de cette hétérogénéité. Par ailleurs, un certain nombre de ces dispositifs est volatile. Par exemple, les *smart phones*, tablettes ou assistants personnels sont des objets mobiles qui se déplacent et modifient ainsi leur disponibilité du point de vue des réseaux. Aussi, des problèmes de latence, de perte de réseaux ou simplement d'arrêt ou de mise en marche d'un dispositif peuvent causer apparition ou disparition non anticipées. Pour aborder les problèmes soulevés par les applications ubiquitaires, il est ainsi nécessaire de traiter les défis posés par les systèmes distribués et mobiles. Nous pensons à ce niveau qu'il n'est pas envisageable de laisser la gestion de l'intégralité de ces défis aux seuls programmeurs. Cela nuirait non seulement au développement mais aussi à la maintenance des applications puisque le code nécessaire à la gestion du dynamisme et des communications ce trouverait mêlé avec

le code applicatif.

Le framework RoSe a également été adopté pour faciliter le développement et l'exécution de serveurs d'applications modulaires. Les serveurs d'application ont fréquemment des besoins en termes d'import ou d'export pour intégrer des services. Comme dans le cas précédent, la gestion de la distribution (hétérogénéité, communication distance, latence) s'ajoute à la gestion de la dynamicité (évolution des modules, perte de service). Par ailleurs, la disponibilité des applications abritées par les serveurs d'applications est souvent critique et il est important que le plus grand nombre de modifications puissent se faire à chaud en minimisant ainsi le temps d'indisponibilité. En s'appuyant sur un modèle à composant orienté service tel que iPOJO, il est possible de mettre à jour certains composants de l'application à chaud. Toutefois, les approches existantes n'adressent pas les problématiques inhérentes à la distribution. De ce fait, le code applicatif est rapidement couplé avec celui lié à la gestion de la distribution, cassant ainsi les bénéfices apportés par un environnement d'exécution modulaire et dynamique tel que iPOJO. Ici, grâce à la séparation des préoccupations et à l'utilisation de composants orientés service, RoSe apporte à la fois modularité et évolution durant l'exécution.

Dans les sections suivantes, nous détaillons l'utilisation de RoSe dans ces deux domaines de façon à illustrer les parties import et export de notre framework.

4.4.2 Informatique pervasive

L'approche proposée par RoSe repose sur la réification des ressources découvertes sur le réseau sous forme de services. Cette approche permet aux développeurs de concevoir des applications manipulant des services distants de manière transparente mais consciente. De plus, la nature dynamique des services réifiés permet de traiter l'apparition et la disparition des services distants. La réification peut concerner tous types de ressources présentes sur le réseau, que cela soit des services UPnP ou des services Web. Elle se déroule en quatre étapes :

1. RoSe est notifié de la découverte d'une ressource distante via un protocole de publication (inclus dans RoSe) ou un annuaire distant (dns-sd, UPnP).
2. RoSe évalue si la ressource doit être importée et, le cas échéant, comment elle doit

l'être et si des métadonnées doivent être ajoutées (sur la qualité de service par exemple).

3. RoSe génère ou instancie un proxy sous forme de service représentant la ressource distante.
4. Lorsque la ressource distante n'est plus disponible ou plus nécessaire, le proxy est détruit.

La première étape repose sur une séparation claire entre les protocoles de communication et de découverte proposée par RoSe. Cela permet de factoriser les mécanismes de découverte mais aussi de gérer des dispositifs utilisant des protocoles différents pour la découverte et pour la communication, comme c'est souvent le cas pour les imprimantes par exemple. Cette séparation apporte également une grande flexibilité pour le programmeur. Il est ainsi possible de découvrir des services Web en utilisant un protocole *multicast* plutôt qu'un registre UDDI (souvent absent).

Pour la seconde étape, RoSe permet de sélectionner les ressources à réifier. Les environnements pervasifs peuvent inclure un très grand nombre de dispositifs et la sélection en amont est nécessaire pour assurer le passage à l'échelle. La sélection dans RoSe s'exprime de manière déclarative dans un fichier séparé. Elle spécifie des filtres à appliquer sur les propriétés des ressources à importer et d'éventuelles propriétés à ajouter aux proxies. Il est également possible d'affiner cette sélection de manière programmatique. Cela permet, par exemple, de gérer des cardinalités, de mettre en place des comportements plus complexes liés à la qualité de service ou encore pallier le manque d'informations transmises par certains protocoles de découverte. La sélection d'une ressource se concrétise par la création ou l'instanciation d'un proxy dont le but est de gérer la communication avec la ressource. Le proxy apparaît comme un service dans l'environnement applicatif. Il contient une propriété indiquant sa nature ainsi que d'éventuelles propriétés non fonctionnelles (son importance relative par exemple). Le processus de sélection de ressource et de création de proxy se déroule pendant l'exécution. Il peut être déclenché par plusieurs événements : l'apparition ou la disparition d'un dispositif, la réception d'une notification correspondant aux filtres préalablement spécifiés ou la disparition d'une fabrique de proxy (ce dernier cas signifie que de nouveaux protocoles de création doivent être utilisés). De plus, la configuration d'une application peut changer. Le framework RoSe est capable de déterminer ces modifications de spécification, de les interpréter et d'effectuer les reconfigurations dynamiques.

La troisième étape correspond à la création du proxy en tant que tel. Les proxies peuvent être générés s'il existe un *mapping* possible entre les interfaces manipulées par l'environnement applicatif local et celles du protocole distant. Cela peut être le cas avec des protocoles standardisés tels que UPnP ou de manière générale avec les services Web. Cependant, la plupart des dispositifs pervasifs demandent le développement de proxies spécifiques. Pour chacun de ces cas, RoSe dispose d'une fabrique (conforme au *factory design pattern*) permettant de créer et de détruire les proxies correspondants. De façon concrète, une fabrique est associée à tout dispositif pouvant être intégré.

La dernière étape est liée au dynamisme des environnements pervasifs. A tout moment, l'environnement applicatif peut recevoir une notification indiquant la disparition (ou l'indisponibilité) d'une ressource distante. Cela peut traduire l'arrêt du dispositif associé (panne, manque d'énergie), un problème réseau ou bien une demande explicite de changement de configuration applicative. Dans ce cas, le proxy associé est détruit. De façon similaire, si une fabrique est mise à jour, tous les proxies issus de cette fabrique sont regénérés.

4.4.3 Serveurs d'applications

Les serveurs d'applications modernes consistent en un ensemble de composants utilisables par les développeurs aux travers d'API bien définies. Idéalement ces API sont fournies aux travers de services, eux-mêmes exposés par les composants. Quelques plateformes modulaires comme OSGi et les composants orientés service tels que iPOJO permettent la conception de tels services découplant ainsi les fonctionnalités des implémentations. L'instance d'un service fournissant une API est publiée dans un broker, ce qui permet aux applications de s'adapter dynamiquement à la disponibilité du service. Ces propriétés sont extrêmement importantes pour les serveurs d'applications puisqu'elles fournissent un modèle de programmation supportant l'adaptation à l'exécution, ce qui est aujourd'hui une exigence majeure de ce domaine [93, 12].

Les serveurs d'applications doivent également interagir avec des clients distants et éventuellement d'autres serveurs distants. Ainsi, en plus de la gestion de l'évolution, les modèles de programmation doivent également traiter la distribution sans impacter le code métier des

applications. Malheureusement, les frameworks actuels ne permettent pas une claire séparation des aspects liés à la distribution et au dynamisme du code applicatif. Ainsi, un développeur voulant exposer un service local comme web-services ou encore comme une ressource REST doit lier son code à une librairie spécifique ainsi qu'à une configuration particulière, introduisant un couplage fort dans son application.

Nous l'avons vu, RoSe permet de découpler le code applicatif et le code lié à la distribution. L'approche mise en place est similaire à celle utilisée pour les environnements pervasifs. Elle se déroule donc en quatre étapes :

- RoSe évalue si un service local doit être exporté, détermine comment il doit l'être et fournit les métadonnées associées au endpoint à créer.
- RoSe génère ou instancie le endpoint permettant d'exposer le service local.
- RoSe notifie les registres distants de la disponibilité d'une ressource.
- Lorsque le service local n'est plus disponible ou si la configuration applicative a changé, le endpoint est détruit.

La première étape est similaire à l'étape initiale dans le cas de l'informatique pervasive, si ce n'est que, cette fois, il s'agit de sélectionner les services à exposer.

La seconde étape traite de la création des endpoints. Il est ici relativement aisé de générer un endpoint dynamiquement pour un service local en s'appuyant sur des librairies existantes telles que Apache CXF ou Jersey. On peut ainsi générer un service Web, un service JSON-RPC, ou encore une ressource REST dont le but est de rediriger les appels sur l'instance du service local. Cependant, bien que la génération « brute » soit possible, il est très intéressant de permettre aux développeurs d'annoter leur code de façon à affiner la génération. Cela permet d'apporter une sémantique cohérente au sein d'un système distribué puisque seules les informations spécifiées sont présentées et, de plus, dans un format approprié. RoSe s'appuie sur les standards Java JAX-WS ou JAX-RS pour spécifier la façon d'exposer une ressource, avec pleine conscience de la distribution. L'approche de Rose se basant sur l'ajout d'annotations sur le service, au niveau interface ou implantation, il est possible de développer des services qui sont aussi des ressources REST, obtenant ainsi les avantages d'un service web restful (voir 2.2.5). De plus, en se fondant uniquement sur ces annotations, les services à distribuer n'ont pas de dépendances vers les librairies liées aux protocoles de communication.

La troisième étape correspond à l'annonce des endpoints sur le réseau. Chaque endpoint a une description associée contenant ses métadonnées. Elles incluent son interface et des propriétés non fonctionnelles liées au service (version, identifiant, etc.) ainsi que des informations liées au protocole (url, nom du protocole) et au framework (id). RoSe permet de publier la description suivant divers protocole de découverte et donc de notifier des clients hétérogènes de la disponibilité du service.

La quatrième et dernière étape correspond à la destruction du endpoint. Cela est nécessaire lorsqu'un service n'est plus disponible ou que la configuration applicative a changé (un service ne doit plus être exposé). Lors de la destruction d'un endpoint, les clients sont notifiés.

4.5 Conclusion

Dans ce chapitre, nous avons introduit les éléments fondamentaux du framework RoSe qui constitue le cœur de notre proposition. RoSe permet d'automatiser l'intégration et la publication de ressources dynamiques et hétérogènes dans des environnements distribués.

RoSe permet une réelle séparation du traitement de la distribution. Les besoins en import et en export de ressources sont exprimés de façon déclarative dans un fichier séparé. Cela permet au programmeur de se focaliser sur le code métier, sans se soucier de la gestion de la distribution, du dynamisme et de l'hétérogénéité. La définition de filtres et de conditions diverses permet de passer à l'échelle et d'aborder des domaines tels que l'informatique pervasive. L'adaptabilité à chaud de RoSe permet de modifier les filtres, les protocoles, les frameworks utilisés et ainsi d'aborder des domaines tels que le cloud computing.

RoSe repose sur les composants orientés service, l'inversion de contrôle et les patrons de conception de type *fabrique*. Il est implanté aujourd'hui au dessus de OSGi/iPOJO et de JavaScript. Les applications visées doivent donc être conçues et implantées suivant ces langages. Par contre, les ressources à intégrer ou publier peuvent être dans des formats très divers.

Objectifs	**RoSe**
Transparence de la distribution	Import : les *proxies* apparaissent sous forme de services. Une propriété indique qu'ils sont distants. Il n'y a pas d'impact sur le code applicatif. Export : les *endpoints* exposent les services locaux et sont générés. Cela n'impacte pas le code applicatif.
Séparation des préoccupations	Langage déclaratif dédié pour l'import et l'export. L'aspect lié à la distribution est séparé du code applicatif.
Automatisation	Import : les *proxies* sont générés ou instanciés par RoSe. Export : les *endpoints* sont également générés. Les expositions peuvent être affinées par annotations.
Passage à l'échelle *anarchique*	Filtrage dynamique des services à intégrer et des services à publier.
Flexibilité/élasticité des applications	Import : gestion dynamique du cycle de vie des *proxies* en fonction des évolutions applicatives. Export : gestion dynamique du cycle de vie des *endpoints* en fonction des évolutions applicatives.
Simplicité	Import : langage déclaratif très focalisé (*domain-specific*) Export : utilisation d'annotations standard.
Diversité	Gestion multi-protocolaire et extensibilité.

TABLE 4.2 – Objectifs.

Chapitre 5

Implantation et Utilisation de RoSe

"It's not what you are, it's what you don't become that hurts." *Oscar Lavant*

5.1 Introduction

Le but du cinquième chapitre est de présenter en détail le framework RoSe dans son état actuel. Il s'agit d'une part de spécifier l'architecture interne du framework et, d'autre part, de détailler l'implantation des composants majeurs. Ces composants sont développés au-dessus des technologies OSGi/iPOJO. A l'heure actuelle, RoSe peut être intégré avec du code applicatif développé en iPOJO (Java/OSGi) ou en H-ubu (JavaScript).

Le framework RoSe est pleinement opérationnel. Dans la seconde partie, nous montrons comment RoSe peut être installé et utilisé. En particulier, nous présentons les différentes APIs et langages déclaratifs proposés par RoSe.

5.2 Implantation du framework RoSe

5.2.1 Architecture globale

RoSe est un framework orienté objet focalisé sur la gestion de la distribution dans des environnements hétérogènes et dynamiques. L'objectif de RoSe est ainsi de faire le lien entre des applications et leur environnement informatique (local et distant) de façon transparente

et extensible.

Nous présentons ici l'implantation de référence de RoSe. Elle repose sur l'utilisation des technologies OSGi et iPOJO, présentées précédemment dans ce manuscrit, et se focalise sur les ressources exposées sous forme de services logiciels. Cette notion a également été discutée dans l'état de l'art de cette thèse.

RoSe se décompose en plusieurs éléments architecturaux qui collaborent pour permettre l'exposition de services locaux(export) et la réification de services distants(import) suivant différents protocoles et différentes conditions. Cette décomposition a été élaborée afin de permettre l'utilisation de RoSe dans de nombreux contextes et de faciliter le développement et l'administration d'applications RoSe.

Nous présentons ici l'architecture globale du framework RoSe. La figure 5.1 donne un premier aperçu des éléments architecturaux majeurs du framework RoSe et leur rôle respectif. Elle présente ainsi les composants majeurs, ainsi que les liens conceptuels les réunissant. Pour simplifier la lecture du diagramme, nous ne faisons pas apparaître les composants liés à la découverte et à la publication des services sur les réseaux. Ils sont présentés en détail par la suite.

Le framework RoSe s'articule autour des types de composants suivants :
- les composants représentant les services distants et locaux (`RemoteService` et `OSGiService`),
- les composants techniques de type connecteurs permettant la communication effective entre services distants (les `Proxies` et les `Endpoints`),
- les composants permettant de décrire les services, qu'ils soient distants ou locaux (`Description` et `Contract`),
- les composants de fabrique permettant la construction des composants de communication (`ImporterService`, `ExporterService`),
- un composant de médiation assurant la communication asynchrone entre les différents composants du framework (`RoseMachine`).

Décrivons maintenant ces différents composants de manière plus précise.

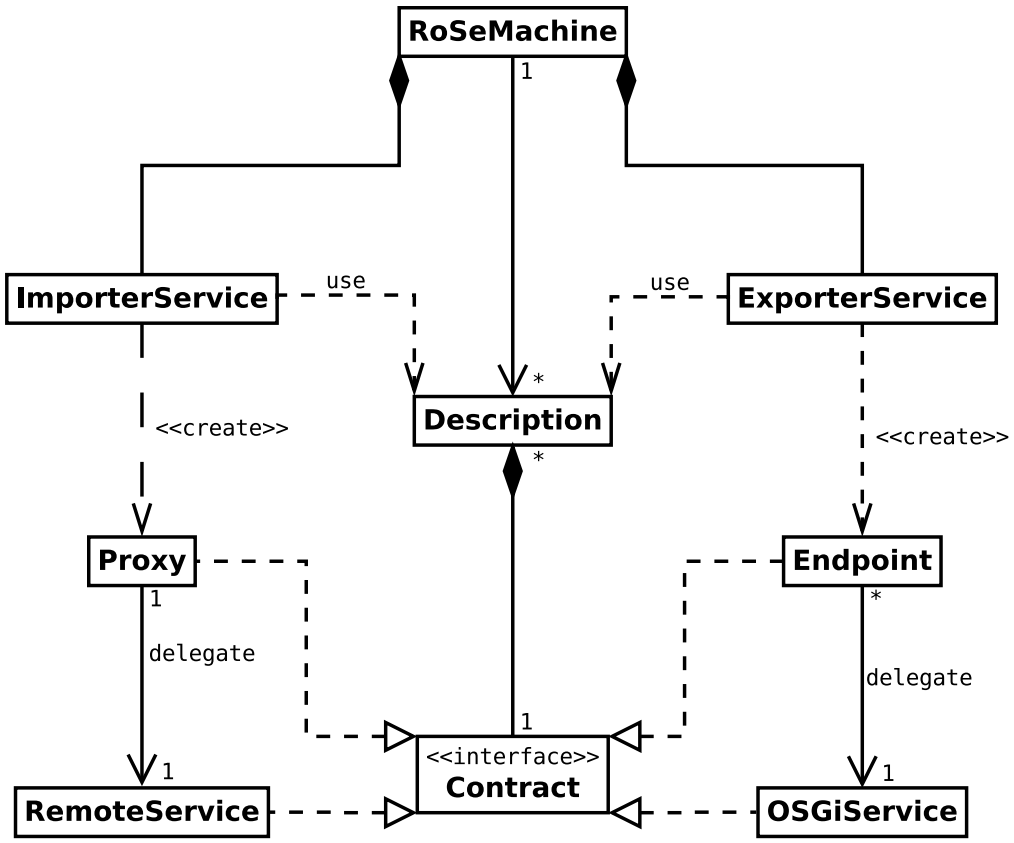


FIGURE 5.1 – Relations entre les composants du framework.

Les instances du composant OSGiService représentent les services disponibles localement sur une machine. Ces services peuvent être exportés et ainsi rendus accessibles à distance. Pour chaque service local exporté, une instance d'Endpoint est créée de façon à exposer le service et rediriger les appels distants vers le service local. Les instances du composant RemoteService représentent des services ne s'exécutant pas sur la même machine d'exécution. Les services distants sont réifiés comme des services locaux grâce à la création d'instances du composant Proxy qui délèguent les appels locaux sur les services réels (distants).

Chaque service géré par RoSe, importé et exporté, possède une description (*Description*) précisant le contrat qu'il implante, ainsi que ses propriétés fonctionnelles et non-fonctionnelles comme par exemple l'adresse du Endpoint et le protocole utilisé. Le contrat est un élément

fondamental dans l'approche à service et est défini par un élément architectural propre (`Contract`).

La notion de description de service est utilisée par les importateurs de services (`ImporterService`) pour générer ou instancier un Proxy correspondant au service distant. A l'inverse, les exportateurs de services (`ExporterService`) créent cette description à partir des propriétés du service local et du protocole utilisé par le Endpoint. Ces descriptions de services sont échangées entre machine via des composants de découverte détaillés dans la suite de ce chapitre.

Le composant central du framework RoSe est appelé `RoseMachine`. Le rôle de ce composant est de prendre en charge la gestion des services locaux (`OSGiService`) exportés et des services distants (`RemoteService`) importés. A tout moment, ce composant connaît l'ensemble des services distants et locaux disponibles et fournit les moyens techniques pour l'envoi de notifications entre composants. En ce sens, il correspond à un médiateur au sein du framework.

Dans ce chapitre, nous revenons sur chacun des composants du framework et détaillons leur rôle ainsi que leurs relations. Nous présentons également les principes de conception reposant sur des patrons architecturaux éprouvés.

5.2.2 La machine RoSe (RoSeMachine)

Le composant `RoSeMachine` tient une place centrale dans le framework RoSe. D'une part, il maintient l'ensemble des services disponibles au sein du framework et, d'autre part, il assure le rôle de médiateur pour tous les autres composants du framework. Chaque instance de RoSeMachine représente une *machine virtuelle* et contient les descriptions de tous les services distants et locaux gérés par le framework.

Ce composant repose sur des méthodes d'introspection qui permettent de connaitre l'ensemble des services importés et exportés à tout moment ; et il notifie les composants intéressés par des changements sur la disponibilité et les caractéristiques de ces services. En fait, tous les composants du framework RoSe peuvent envoyer et recevoir des notifications (événements) via la machine RoSe. Ainsi, la machine RoSe définit et implante les

interactions entre chaque composant du framework en redirigeant les notifications vers les composants appropriés. La machine RoSe suit le patron de conception connu sous le nom de *mediator* (voir figure 5.2). Ce patron favorise un couplage faible entre différents éléments en interaction en éliminant les référencements explicites et en permettant de reconfigurer leurs interactions de manière indépendante.

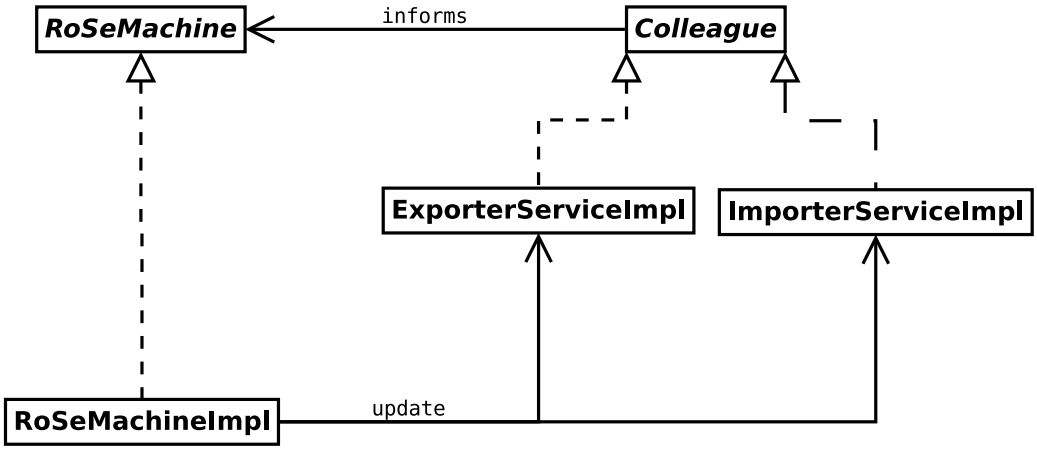


FIGURE 5.2 – Le patron Mediator dans RoSe.

La machine RoSe s'appuie sur le broker de service OSGi pour gérer les interactions et évite ainsi d'avoir à maintenir un registre de service privé. Notre implantation repose également sur le patron de conception *WhiteBoard* qui permet de s'appuyer sur le registre de service OSGi plutôt que d'avoir à maintenir son propre registre privé, comme cela est nécessaire avec le patron de conception dit *listener* (voir figure 5.3). En effet, le whiteboard pattern permet aux listeners de s'enregistrer eux mêmes en tant que service dans le framework OSGi plutôt que d'avoir à tracker les sources d'évènements et dans un second temps de s'enregistrer auprès d'elles. L'utilisation combinée du registre propre à OSGi et du patron de conception *WhiteBoard* permet à RoSe de supporter le dynamisme.

Plus précisément, les composants désirant connaître la disponibilité des services locaux et distants doivent publier un service implémentant le contrat `EndpointListener`. Ce

contrat définit deux fonctions principales, `endpointAdded` et `endpointRemoved` qui sont appelées respectivement à chaque fois qu'un Endpoint a été créé et à chaque fois qu'un Endpoint a été détruit. La description du endpoint est passée en paramètre. La machine RoSe peut ainsi notifier les composants intéressés par toute évolution topologique en les appelant au travers de ce service.

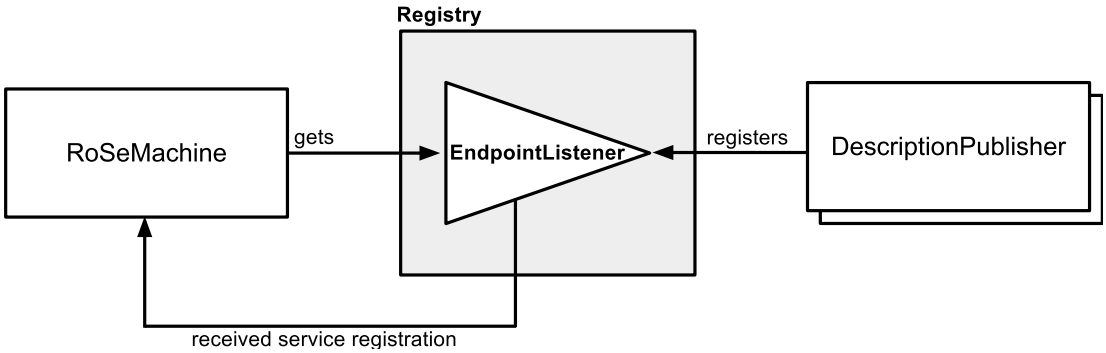


FIGURE 5.3 – Exemple d'utilisation du whiteboard pattern dans RoSe.

La figure 5.4 illustre les relations entre le composant `RoSeMachine` et les autres composants du framework. Cette figure fait notamment apparaître les éléments `ImporterService` et `ExporterService` qui sont les fabriques capables d'instancier des `Proxies` et des `Endpoints`. La `RoSeMachine` est également liée au composant de découverte nommé `DescriptionDiscoverer` qui fournit les descriptions des services distants découverts sur le réseau.

5.2.3 Les connecteurs (Proxy et Endpoint)

RoSe repose sur la notion de connecteur permettant la communication entre applications et services distribués. Les connecteurs donnent la possibilité à une machine d'exécution de communiquer avec d'autres machines utilisant, bien entendu, des espaces d'adresses différents. Les connecteurs permettent de clairement séparer le code de communication et le code applicatif ; ce qui amène une séparation effective des préoccupations.

Comme nous l'avons vu à maintes reprises, RoSe définit deux types de connecteurs distincts pour gérer la distribution :

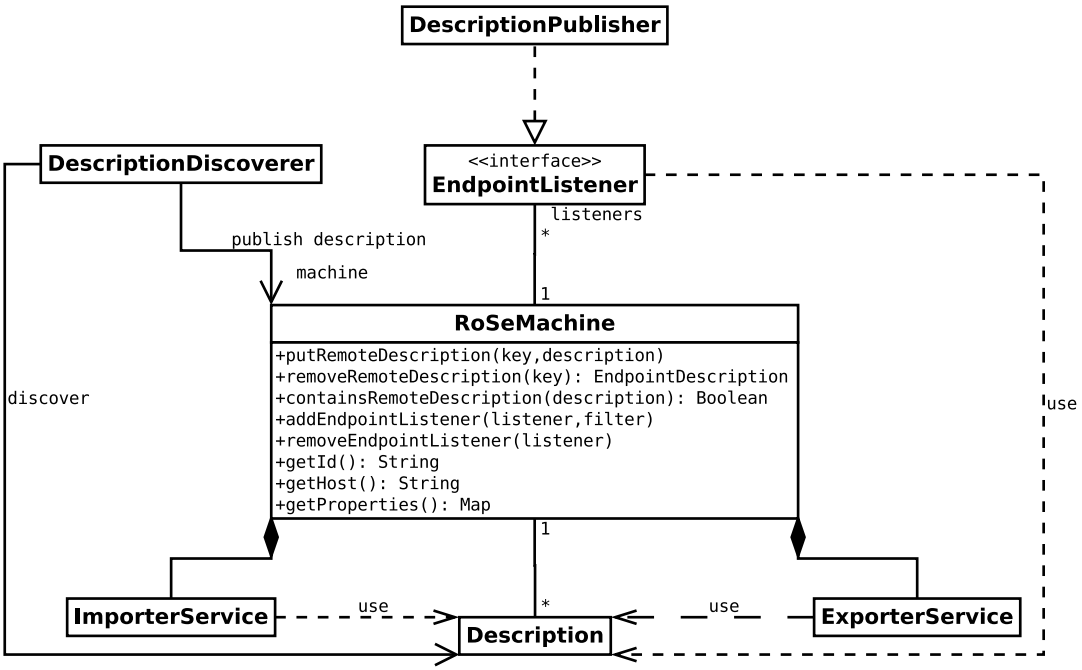


FIGURE 5.4 – Diagramme de classes RoseMachine.

- les connecteurs d'import appelés `Proxies` permettent d'accéder à des services distants hébergés par des machines différentes,
- les connecteurs d'export appelés `Endpoints` permettent à toutes autres machines d'accéder à des services s'exécutant dans l'espace d'adressage propre à la machine hébergeant le framework RoSe.

Les Proxies suivent le patron de conception portant le même nom. Un proxy est donc un composant (un composant iPOJO dans notre implantation) qui représente et fait le lien avec un objet s'exécutant dans un espace d'adressage différent. Dans notre cas, cet objet est exposé sous forme de service et peut s'exécuter sur toutes sortes de machines, comme par exemple un équipement ambiant ou un serveur Internet. Il peut également utiliser différentes technologies à services (et les protocoles de communication associés) telles que UPnP pour les équipements ambiants ou les services Web pour les services Internet.

Un proxy fournit ainsi un service local respectant le contrat du service distant en déléguant

chaque appel sur le service distant via le protocole associé. Le cycle de vie des proxies est entièrement géré par les importateurs de RoSe (`ServiceImporter`), chaque importateur étant spécialisé dans un protocole particulier. C'est donc uniquement au travers d'un importateur que les proxies peuvent être créés (par génération ou par instanciation) et détruits. Un service distant ne peut avoir qu'un seul proxy pour une technologie donnée dans une même machine.

Les Endpoints sont les connecteurs d'export. Ils permettent à des machines distantes d'accéder à un service local. Chaque endpoint peut donc apparaître comme un proxy vers un service local. Les endpoints sont créés et instanciés par les exportateurs de RoSe (`ServiceExporter`), chaque exportateur étant spécialisé dans un protocole particulier, tout comme les importateurs.

Un service local peut s'exposer suivant différents protocoles. En ce sens, un service local peut avoir plusieurs endpoints, soit un par protocole au moyen duquel il est exposé. Chaque endpoint est caractérisé par une description unique. Cette description permet aux machines distantes de réifier le service.

5.2.4 Les services (OSGiService et RemoteService)

L'objectif premier de RoSe est d'être associé à du code applicatif. Cela signifie qu'un développeur doit d'une part développer son application et d'autre part configurer le framework RoSe pour qu'il traite de la communication de manière indépendante et de la façon la plus autonome possible. Dans le cas d'imports, le développeur peut aussi être amené à coder des proxies, si ceux-ci ne peuvent pas être générés (cela dépend directement de la technologie de communication utilisée et du niveau de description des services).

Le code applicatif est développé en utilisant un modèle à composant orienté service. Aujourd'hui, il est possible d'utiliser le modèle iPOJO ou le modèle H-ubu pour le développement de la partie applicative. Nous avons démontré avec succès la réification de RoSe dans ces deux modèles. Lors de son travail, le concepteur/développeur peut spécifier des services locaux à exporter (`OSGiService`) ou des services distants à importer (`RemoteService`), s'ils sont présents sur le réseau au moment voulu.

Le framework RoSe n'impose aucune contrainte sur le développement des services locaux. En ce qui concerne l'implantation de référence sur OSGi/iPOJO, les services locaux sont de simples services OSGi. Dans le cas de l'implantation en JavaScript, les services locaux sont des objets H-ubu. Ici, les services sont donc des objets JavaScript publiés par un composant H-ubu. RoSe se base sur la définition des services locaux (iPOJO ou H-ubu) et de leurs propriétés pour générer les Endpoints demandés via les ExporterService. Néanmoins, la génération des Endpoints dépend fortement du protocole de communication utilisé et peut s'avérer problématique, notamment au niveau de la sérialisation des invocations. C'est pourquoi le développement de services ayant vocation à être distribués doit se faire en conscience et en portant une attention toute particulière aux problèmes liés à la distribution. L'utilisation de type *sérialisables* fait partie de ces contraintes. Nous proposons, l'utilisation des annotations JAX-WS et JA-RS pour le développement de services exposés respectivement comme web service et comme web service restful. Dans le cadre des serveurs d'applications, nous recommandons la création d'une couche d'interactions constituée de services spécialement destinés à être distribués et déléguant les appels sur les services métiers.

Pour ce qui est des services distants, RoSe s'appuie avant tout sur leur description. Un service distant s'exécute sur une machine d'exécution séparée et doit être exposé de façon explicite par l'administrateur concerné afin d'être utilisé a distance. Un service distant peut aussi bien être fourni par un périphérique connecté sur le réseau, comme une imprimante par exemple, que par une autre machine virtuelle. Il n'appartient pas forcément au même domaine d'administration que RoSe. Cela signifie notamment que l'on ne peut pas *forcer* un service distant à s'exposer. Chaque service distant doit posséder une description contenant le contrat rempli par le service et d'éventuelles propriétés supplémentaires. De façon générale, la description doit contenir toutes les informations nécessaires à la création d'un proxy du coté RoSe. En pratique, la seule entité qui permette d'identifier un service distant avant qu'il ne soit réifié au travers d'un proxy est sa description. De ce fait, les composants de découverte ne découvrent pas des services distants mais plutôt leur description. C'est donc cette description, unique pour chaque service distant, qui matérialise la disponibilité.

5.2.5 Les fabriques (ImporterService et ExporterService)

Comme décrit précédemment, les proxies et les endpoints sont créés et détruits par des composants particuliers, respectivement les importateurs (`ImporterService`) et les exportateurs (`ExporterService`). Ces services suivent le patron de conception connu sous le nom de *fabrique*. La mise en œuvre de ce patron pouvant être complexe, RoSe fournit des classes abstraites permettant de ne fournir que les aspects dépendants du protocole lors de l'implantation d'une fabrique. Ces composants sont souvent du code intégrant une librairie dans RoSe.

Comme indiqué par la figure 5.5, les importateurs utilisent les descriptions des services distants pour créer les proxies correspondants. Chaque importateur est spécialisé dans un protocole particulier. Les informations nécessaires pour la création du proxy sont contenues dans la description du service ainsi que dans des propriétés (optionnelles) qui peuvent être fournies par le framework.

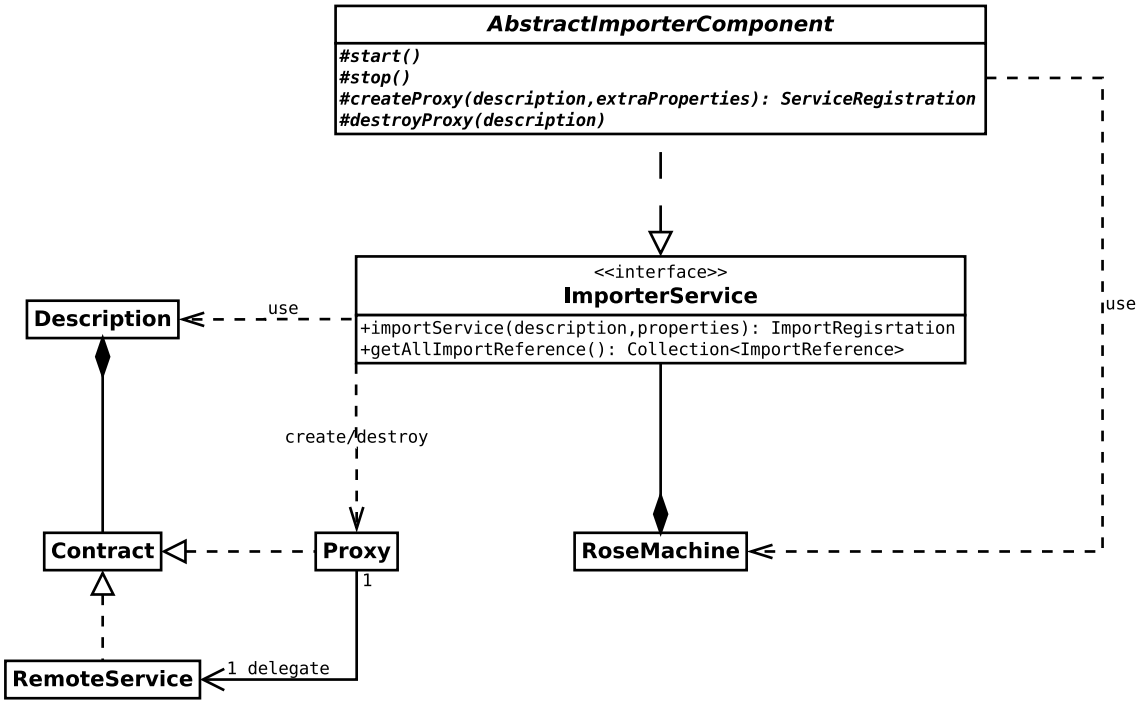


FIGURE 5.5 – Diagramme de classes ImporterService.

Un composant abstrait (`AbstractImporterComponent`) permet de faciliter la création des importateurs. En effet, à chaque demande d'import une ImportRegistration est créée. C'est au travers de cet enregistrement que le proxy peut être détruit. Quand tous les enregistrements sont *fermés*, alors le proxy est détruit. Ainsi, si un service est importé plusieurs fois (par différentes configurations), le proxy ne sera détruit que lorsque toutes ces demandes d'importation seront terminées. C'est notamment cet aspect que gère le AbstractImporterComponent. En pratique, les importateurs sont toujours utilisés par des composants du framework RoSe, et les développeurs n'utilisent jamais ces services directement (ils ne font *que* de la configuration).

Les exportateurs sont symétriques aux importateurs à la différence qu'ils utilisent les propriétés du service local et le service lui-même (l'objet) pour créer l'endpoint correspondant au service. De la même manière que les importateurs, chaque exportateur est spécialisé pour un protocole donné (tel que jsonrpc, jax-ws et jax-rs par exemples).

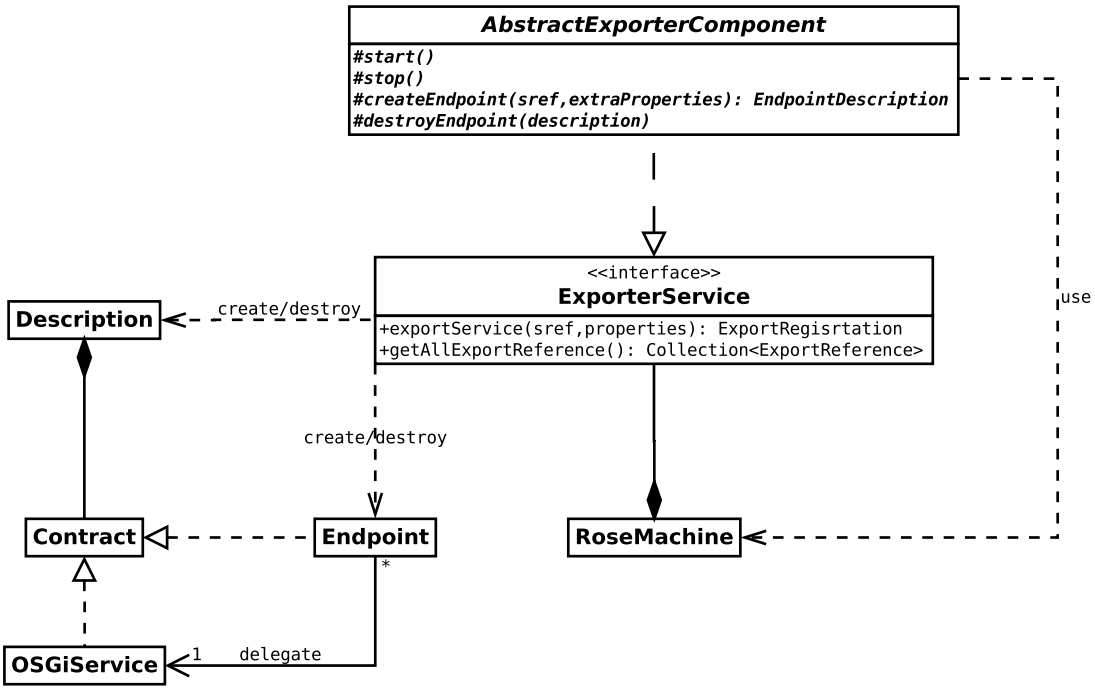


FIGURE 5.6 – Diagramme de classes ExporterService.

Un composant abstrait (`AbstractExporterComponent`) permet de faciliter la création des

exportateurs. Chaque endpoint créé possède une description associée et une instance d'-ExportRegistration qui fonctionne sur le même modèle que l'enregistrement d'importation précédemment décrit. Comme pour les importateurs, chaque exportateur possède une propriété qui permet d'identifier le protocole auquel il est lié.

5.2.6 Métadonnées

La notion de métadonnées joue un rôle primordial dans le framework RoSe. En effet, quel que soit le modèle à composant utilisé pour le code applicatif, un service peut être caractérisé par des métadonnées. Cela permet d'effectuer des sélections de service basées sur des propriétés et pas seulement sur le contrat.

Ainsi, chaque service exposé par notre canevas possède des propriétés associées qui peuvent être valuées par différents composants. Par exemple, les importateurs et exportateurs peuvent affecter une propriété qui permet d'identifier le protocole utilisé par un service. La machine RoSe, quant à elle, apporte l'identifiant et le nom de la machine (hostname) sur laquelle s'exécute un service. Un proxy, qui rappelons-le, apparaît comme un service, possède une propriété supplémentaire qui permet d'identifier le fait que ce service correspond à un service distant ainsi que des informations liées à son protocole.

Les descriptions de services distants sont des métadonnées clés de l'architecture de RoSe. Une description correspond aux métadonnées associées à un service distant. Ces métadonnées contiennent les informations nécessaires à la création de proxy. Elles contiennent l'adresse du endpoint, le contrat fournit par le service distant, éventuellement d'autres informations liées au protocole, les identifiants de la machine distante ainsi que tout autre propriété que les développeurs auront estimé nécessaire d'ajouter (qualité de service, étiquette, etc...). Le tableau 5.1 récapitule ces propriétés. Les descriptions sont échangées aux travers de divers protocoles de découverte et représentent ainsi la disponibilité des services distants. Ce dernier aspect permet de modéliser le dynamisme.

Puisque les descriptions sont amenées à être échangées et sérialisées, elles sont du type Java *immutables*. Cette contrainte impose que les métadonnées associées à un service soient statiques. La modification des métadonnées revient à notifier la disparition de la description et la réapparition d'une nouvelle description contentant les nouvelles métadonnées.

Ceci permet donc de publier les descriptions aux travers de protocoles aussi divers que des multicasts, des registres distribués ou encore des mémoires partagées.

Propriété	Description	Optionnel
endpoint.id	Identifiant de l'endpoint	
machine.id	Identifiant du framework	
service.contrat	Nom du contrat	
service.contrat.version	Version du contrat	√
service.id	Identifiant du service associé	√
endpoint.url	Adresse de l'endpoint	√
endpoint.config	Nom du protocole	

TABLE 5.1 – Propriétés d'une description.

5.2.7 Découverte (DescriptionDiscoverers et DescriptionPublishers)

Les composants de découverte permettent à une machine RoSe de découvrir les services distants disponibles aux travers des réseaux et de publier les endpoints créés sur la machine.Plus précisément, ces composants ont la responsabilité de découvrir les descriptions des services distants et de publier les descriptions des services locaux exportés. Tout comme pour les importateurs et les exportateurs, chaque composant de découverte est spécialisé sur un protocole particulier. Ainsi, pour des applications ubiquitaires, on s'appuiera par exemple sur des découvertes basées sur des *multicasts* alors qu'on préférera l'utilisation de registre pour les serveurs d'applications (tel que JNDI). Comme indiqué par la figure 5.4, on distingue deux types de composants majeurs : les DescriptionDiscoverers et DescriptionPublishers.

Les découvreurs de description sont les composants chargés de notifier le framework RoSe de la disponibilité d'un service distant via la publication de sa description dans un registre de RoSe prévu à cet effet. Ensuite, si le service découvert correspond aux besoins de RoSe, alors un proxy est généré ou instancié. Les besoins de RoSe sont spécifiés par les développeurs grâce notamment à des filtres LDAP qui permettent d'exprimer les types de services devant être intégrés au code applicatif. Bien que la sélection soit faite en aval de ce composant, il peut être intéressant de filtrer les descriptions à partir des composants de découvertes pour éviter des phénomènes d'inondation (*flood*) due à la présence en grande

quantité de services distants sur le ou les réseaux connectés à la machine.

Lorsqu'un service disparaît, les découvreurs de description doivent retirer sa description du registre RoSe prévu à cet effet. Si le service a été réifié sous la forme d'un proxy, ce proxy doit également être détruit.

Les publieurs de description sont notifiés par le framework RoSe à chaque fois qu'un endpoint est créé ou détruit. Les publieurs reçoivent ces notifications aux travers d'un service qu'ils fournissent (voir figure 5.4), l'EndpointListener, en suivant les principes du patron de conception WhiteBoard. Une fois la description reçue, les publieurs la sérialisent et la publient sur leur protocole (tel que dns-sd, zookeeper etc...). La sérialisation est bien entendu spécifique au protocole. Une propriété peut être ajoutée au service EndpointListener afin de filtrer les descriptions reçues.

5.2.8 Connexions (Connections)

Les connexions sont des entités particulièrement importantes dans notre approche. Elles contiennent la logique de la distribution. Cela signifie que les connexions décident de l'import et de l'export des services en fonction du contexte courant. Pour cela, les connexions traquent et analysent les descriptions de services distants ainsi que celles des services locaux qui doivent être exportées. Une singularité de notre canevas est que ces évaluations se font à l'exécution et doivent pouvoir prendre en compte les changements de disponibilité des services mais aussi les changements de configuration de ces services. Ces connexions peuvent être instanciées et configurées de manière déclarative avec un langage de configuration (interprété à l'exécution) ou bien de manière programmatique grâce à une API dédiée. Il existe deux types de connections, les connexions entrantes (`InConnection`) et les connexions sortantes (`OutConnection`). Le comportement par défaut de ces entités peut être spécialisé grâce au concept de Customizer qui peut être étendu par les développeurs.

Les connexions entrantes (InConnection)

Une connexion entrante est chargée d'analyser les descriptions de services distants reçues par la `RoSeMachine`. Une fois les descriptions analysées, la connexion décide ou non de

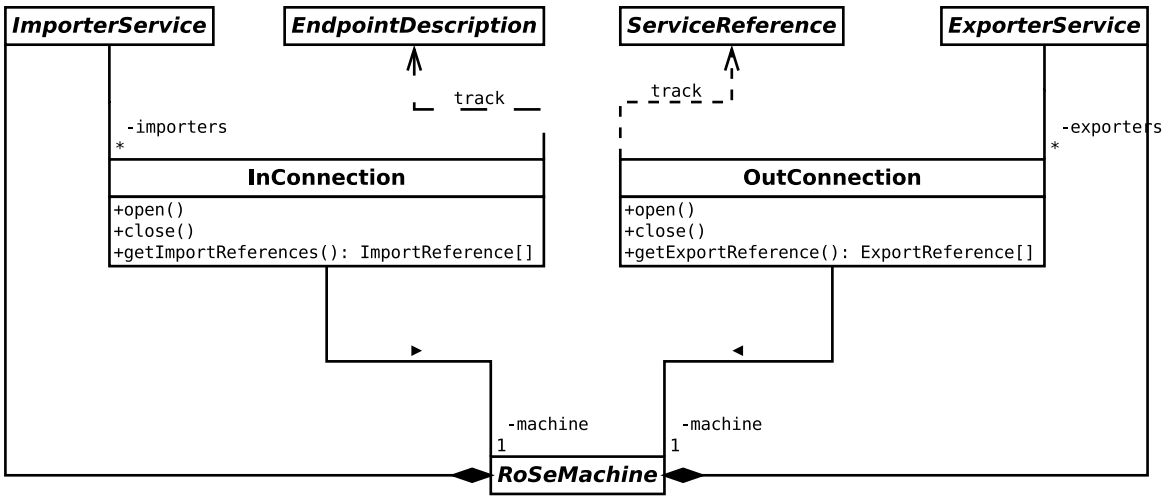


FIGURE 5.7 – Diagramme de classes InConnection et OutConnection.

réifier les proxies en se reposant sur les importateurs de services. Finalement, elle est en charge de détruire le proxy s'il n'est plus nécessaire d'importer le service distant ou encore si un élément indispensable au proxy n'est plus disponible. Ce composant contient donc la logique d'importation.

Chaque instance de InConnection correspond à une configuration d'importation particulière. Ainsi, un utilisateur du canevas RoSe voulant réifier des services distants doit seulement créer une instance de connexion entrante avec la configuration appropriée. Cette instance requiert les propriétés suivantes :

Le diagramme de séquence ci-après (figure 5.8) illustre le processus d'importation de service de description (`desc`), depuis sa découverte par un découvreur de description jusqu'à la réification du proxy correspondant. Ce processus est réitéré à chaque fois qu'un nouveau service distant est découvert pour chaque connexion entrante instanciée. Les instances de connexions entrantes suivent le cycle de vie suivant (voir figure 5.9) :

- **Init** : la connexion a été correctement configurée.
- **Open** : la connexion a été ouverte et recherche le ou les importateurs requis.
- **Ready** : dans cet état la connexion est correctement liée à un importateur de service, grâce auquel elle pourra réifier le service distant.

Propriété	Description	Optionnel
description.filter	Toutes les descriptions qui passent ce filtre (ldap) sont réifiées sous forme de proxy.	
importer.filter	Le premier importer qui passe ce filtre et réussit à créer un proxy est utilisé pour importer le service.	√
protocol	Plutôt que d'utiliser un filtre pour les importer on peut directement spécifié le nom du protocole.	√
properties	Un dictionnaire de propriété qui peut être passé à l'importer et répercuté sur les propriétés du service fourni par le proxy.	√
customizer	Le customizer à appeler avant d'effectuer l'import.	√

TABLE 5.2 – Propriétés d'une `InConnection`.

– **Active** : une description qui correspond à la configuration est présente, par défaut la connexion délègue la création du proxy à l'importateur de service.

Si la connexion entrante est stoppée ou si l'importateur de service utilisé disparaît, alors tous les proxies créés sont détruits. A chaque démarrage (état open), la connexion vérifie si un importateur est présent et si des descriptions correspondant à sa configuration ont déjà été découvertes. Une configuration particulière permet d'ajouter un `Customizer` à la connexion. Ce customizer est appelé dans l'état actif avec la description et l'importateur utilisé. Le customizer prend la décision de réifier ou non le service distant.

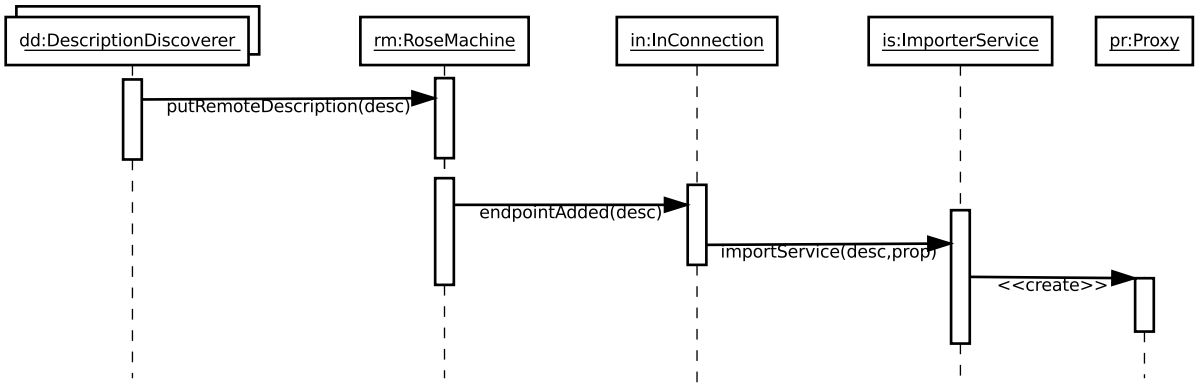


FIGURE 5.8 – Diagramme de séquence, réification d'un service distant.

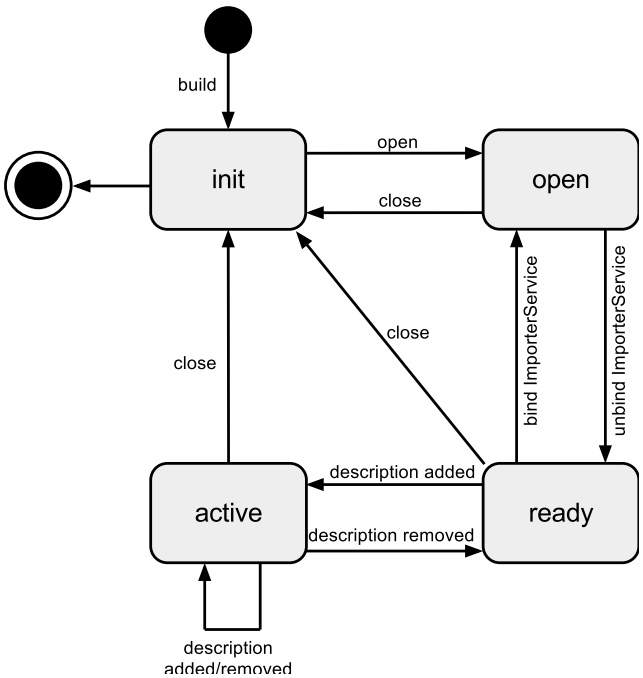


FIGURE 5.9 – Diagramme d'état InConnection.

Connexions sortantes (OutConnection)

Les connexions sortantes sont les entités chargées d'analyser les services locaux et, s'ils correspondent à leur configuration, de les exporter grâce aux exportateurs de service auxquels elles sont liées. Leur conception et comportement sont en tout point symétrique à celles des connexions entrantes. La différence principale réside dans le fait qu'un service local peut avoir plusieurs endpoints alors qu'un service distant n'a qu'un proxy unique (pour un protocole donné). Le diagramme de séquence suivant illustre le processus de création d'un endpoint depuis l'apparition du service jusqu'à sa création.

Chaque instance de connexion sortante intéressée par un service local crée un endpoint pour chaque exportateur de service auquel elle est liée. La description correspondante à l'endpoint est ensuite publiée au travers de la `RoSeMachine` qui se charge de notifier tous les publicateur de description de la disponibilité des nouveaux endpoints. Chaque instance d'une connexion sortante correspond à une configuration d'exportation de service. Ainsi,

un utilisateur de RoSe voulant exporter un ensemble de services présents dans la plateforme d'exécution doit seulement créer une instance de connexion sortante avec la configuration adéquate. Cette création nécessite les propriétés listées dans le tableau 5.3.

Propriété	Description	Optionnel
service.filter	Un ou plusieurs endpoint sont créés pour chaque service qui correspond à ce filtre (ldap).	
exporter.filter	Chaque exporter qui passe ce filtre et réussit est utilisé pour exporter le service.	√
protocol	Plutôt que d'utiliser un filtre pour les exporter on peut directement spécifié le nom du protocole.	√
properties	Un dictionnaire de propriété qui peut être passé à l'exporter et répercuté dans la Description du endpoint.	√
customizer	Le customizer à appeler avant d'effectuer l'export.	√

TABLE 5.3 – Propriétés d'une `OutConnection`.

Tous comme les connexions entrantes, chaque instance de connexion sortante suit un cycle de vie en 4 états (voir figure 5.9) :

- **Init** : la connexion a été correctement configurée.
- **Open** : la connexion sortante a été démarrée et recherche les services et exportateur de services.
- **Ready** : dans cet état, la connexion sortante est correctement liée à au moins un exportateur de service, grâce auquel elle exporte les services locaux.
- **Active** : un service qui correspond à la configuration est présent, par défaut, la connexion sortante crée un endpoint pour chaque exportateur lié.

5.2.9 Customizer

Il existe deux types de Customizer, les InCustomizer et les OutCustomizer qui servent respectivement à spécialiser les instances des InConnection et des OutConnection. Le comportement des deux types de customizer étant similaire nous nous contenterons de détailler celui des OutCustomizer. Si un customizer a été spécifié lors de la création d'une instance d'un OutConnection alors ce dernier va être appelé pour chaque service et ExpoterService correspondant à la configuration de la connexion. Ce sera donc l'OutCustomizer qui devra prendre la décision finale de créer ou non un Endpointpour le service via l'ExporterService sélectionné (voir figure 5.11). La méthode `export` du customiser est appelée.

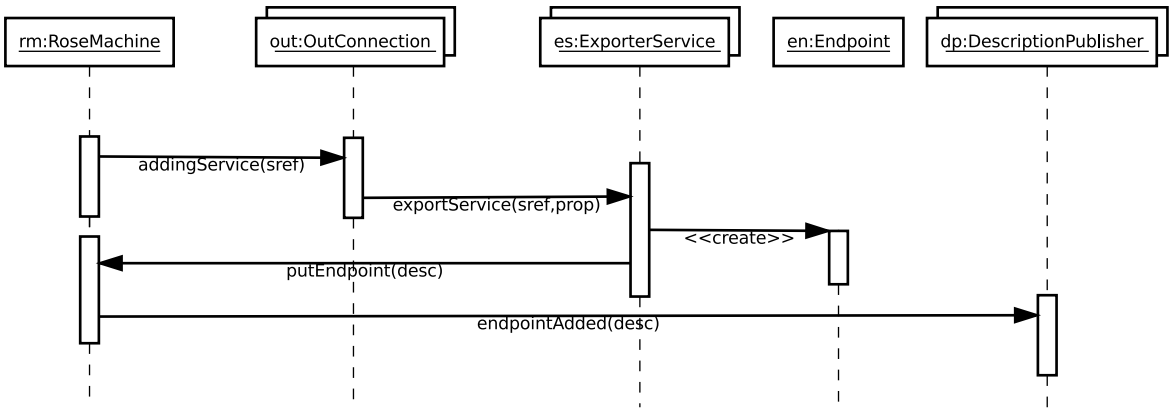


FIGURE 5.10 – Diagramme de séquence export d'un service local.

Si un objet est renvoyé alors l'OutConnection continue à tracker le service et appellera la méthode `unexport` avec ce même objet et le ExporterService en paramètre. Si le customiser renvoie `null`, alors l'OutConnection cesse de tracker le service. De cette manière, les customizer permettent une gestion plus fine de la distribution, en maintenant par exemple une cardinalité ou tout autre comportement que les filtres ne peuvent exprimer. De plus, les customizer peuvent faire appel à des composants tiers pour introduire des qualités de services ou effectuer une sélection plus fine.

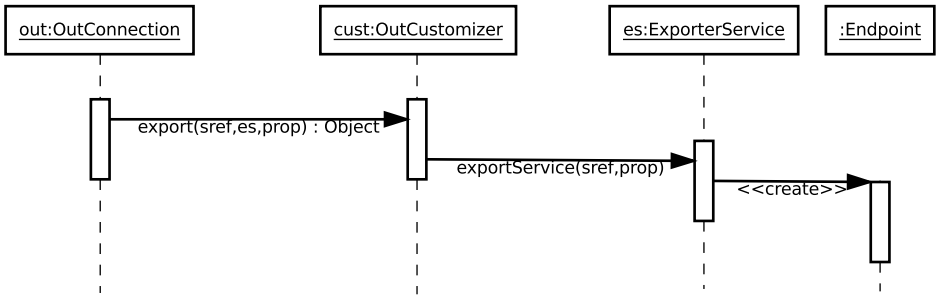


FIGURE 5.11 – Export de service avec un Customizer.

5.3 Utilisation du Framework RoSe

Dans cette section, nous expliquons comment utiliser le canevas RoSe implanté au dessus d'OSGi ainsi que l'ensemble des composants que nous mettons à disposition des concepteurs/développeurs. Nous détaillons en particulier l'API qui permet de configurer RoSe de

façon à gérer les importations de services. Nous montrons comment, en s'appuyant sur des annotations Java, nous permettons de développer facilement des services qui sont exportés comme des services web ou des ressources REST.

5.3.1 Installation

Comme décrit dans la section précédente, le framework RoSe se compose d'un ensemble de composants à déployer dans l'environnement d'exécution. Ainsi, installer la version OSGi de notre canevas revient à déployer un ensemble de bundles OSGi décrits ci-après.

Core bundle Le Core bundle contient l'*API* de RoSe, c'est-à-dire toutes les interfaces des services RoSe (ImporterService, ExporterService, EndpointListener...) ainsi que le composant RoSeMachine, les connexions (InConnection, OutConnection). Les classes abstraites destinées à faciliter le développement sont également inclues dans ce bundle (AbstractComponentImporter, AbstractComponentExporter). Tous les autres bundles du canevas dépendent des classes fournies par le Core bundle.

Importer bundles Chaque composant fournissant un importateur de service est empaqueté dans son propre bundle. Ainsi, il existe un ImporterBundle pour chaque importateur de service, correspondant à un protocole spécifique. De cette manière, nous pouvons spécialiser le canevas en choisissant l'ensemble des protocoles que nous voulons supporter. Ces bundles dépendent du CoreBundle et éventuellement de bundle tiers fournissant des bibliothèques liées au protocole ainsi que leurs dépendances. Ils peuvent aussi dépendre de service tel le `HttpService` ou encore des services de sérialisation.

Exporter bundles Chaque exportateur de service est empaqueté dans son propre bundle, de la même manière que les importateurs. Ces bundles dépendent du bundle core et de bundle tiers fournissant les bibliothèques liées au protocole de communication.

Configurator bundles Les configurator bundles contiennent les composants capables d'interpréter une configuration RoSe. Ils permettent donc de configurer le framework RoSe et donc la distribution de manière déclarative dans un langage spécifique. Il existe pour l'instant une seule configuration basée sur le langage JSON. Nous décrivons cette configuration dans la section suivante.

Discovery bundles Finalement, les Discoverybundles contiennent les publieurs de services et les découvreurs de descriptions. Similairement aux importateurs et exportateurs, chaque module est spécialisé pour un protocole donné, tel que la découverte par *multicast* ou encore l'utilisation de registre de nommage comme Apache Zookeeper.

Ainsi, la modularité de notre canevas permet de configurer et de spécialiser la distribution à la demande, via le déploiement à chaud des bundles requis. Nous pouvons donc aisément construire des profils de déploiement orienté domaine, par exemple pour la domotique ou encore pour un micro-serveur d'application web. De plus, de par la nature dynamique du canevas, l'ajout ou le retrait de ces composants peut être fait à chaud.

5.3.2 Utilisation

Utiliser RoSe consiste essentiellement à exprimer les besoins en matière de distribution d'une application grâce au langage fourni par le framework RoSe. En plus de ce langage déclaratif, les développeurs peuvent s'appuyer sur une version 'embarquée' de ce langage en Java. Ainsi, nous fournissons un *Embedded DSL* [52] qui permet au travers d'un chaînage de méthodes d'instancier une machine RoSe, et de lui associer des connexions ou d'éventuelles instances. Dans cette partie nous décrivons l'API RoSe et nous revenons sur le langage déclaratif.

Embedded DSL en Java

Le concept de *fluent interface* a été énoncé par Martin Fowler et décrit un type de DSL, aussi appelé *Embedded DSL*, ou l'on embarque les caractéristiques d'un domaine particulier dans un langage de programmation existant pour bénéficier de son implémentation et de ces outils [52]. Cette approche permet de fournir aux développeurs une API concise, exclusivement destinée à la distribution de leur application et ce dans un langage maitrisé et disposant d'outils éprouvés. Chaque élément du canevas peut donc être instancié et configuré aux travers de cet API.

Instanciation d'une machine

```
1 import static org.ow2.chameleon.rose.Machine.MachineBuilder.machine;
2 [...]
3
```

```
4 Machine machine = machine(context, "dummyId").host("localhost").create();
5 machine.start(); // start the machine
```

Listing 5.1 – Instanciation d'une machine.

Dans cet exemple nous créons une machine identifiée `dummyId`, avec le nom d'hôte `localhost`. L'argument context est le BundleContext d'OSGi auquel la machine va être liée. C'est cet objet qui permet l'accès aux services composant l'application.

Ajout d'une connexion entrante

```
1 [...]
2
3 /* Add an in connection  for this machine*/
4 machine.in("(service.quality > 5)") // track each endpoint with the
5                                     // service.quality property value > 5
6 .withImporter("(instance.name=RoSe_importer.cxf)") // specify which ImporterService to use
7 .protocols(["ws"]) // specify the protocols
8 .add(); // add the connection
```

Listing 5.2 – Ajout d'une connexion entrante.

Ici, nous ajoutons une connexion entrante `in` à la machine. Tous les services distants dont la description contient la propriété `service.quality` avec une valeur supérieure à 5 et qui sont accessibles au travers d'un Web-Service (`["ws"]`), seront importés avec l'importateur `cxf`.

Ajout d'une connexion sortante

```
1 [...]
2
3 /* Add an outconnection  for this machine*/
4 machine.out("(ObjectClass=foo.bar.HelloWorld)") // track each HelloWorld service
5 .withExporter("(instance.name=RoSe_exporter.jabsorb)" )// specify which exporter to use
6 .protocols(["jsonrpc"]) // specify the protocols
7 .add(); // add the connection
```

Listing 5.3 – Ajout d'une connexion sortante.

La déclaration ci-dessus spécifie que tous les services locaux implantant l'interface `Java foo.bar.HelloWorld` seront exposés comme endpoint JSON-RPC avec l'exportateur `jabsorb`.

Gestion indépendante des connexions Il est important de noter que les connections peuvent être ajoutées durant l'exécution même de la machine, c'est-à-dire après que la machine ait été démarrée (`machine.start()`). De plus, il est possible de gérer chaque connexion de manière individuelle, comme illustré dans l'exemple suivant :

```
1 [...]
2
3 OutConnection out = machine.out("(pasdebras=pasdechocolat)").add(); // add the connection
4 out.close()// close the connection
5 [...]
6 out.open() // open the connection
7 [...]
8 machine.remove(out)  // remove the connection from the machine
9                      // (the connection will be closed too)
```

Listing 5.4 – Gestion indépendante des connexions.

Instanciation des composants du Framework

```
1 [...]
2
3 /* Add an outconnection  for this machine*/
4 machine.instanciate("RoSe_exporter.jersey") // specify the exporter to be instantiated
5 .withProperty("jersey.servlet.name","/rest") // set an instance property
6 .withProperty("instance.name","RoSe_exporter.jersey-default")
7 .add(); // add the Exporter to the machine
```

Listing 5.5 – Instanciation des composants du Framework.

L'API permet aussi de créer des instances de composants du canevas RoSe. Dans cet exemple, nous créons une instance du composant `jersey` qui fournit un exportateur permettant d'exporter les services locaux sous forme de ressources REST.

Langage déclaratif

Si l'API permet un contrôle fin de la distribution durant l'exécution, un DSL découplé du langage de programmation permet plus de souplesse dans la reconfiguration de l'application. C'est pourquoi nous proposons un composant permettant la configuration déclarative de la distribution. Cette configuration est interprétée à l'exécution et est basée sur le langage JSON. Un point essentiel est qu'il peut y avoir plusieurs configurations et

fichiers de configuration pour un seul environnement d'exécution. L'exemple suivant est équivalent à celui présenté dans la section précédente.

```
1  {
2      "machine" : {
3      "id" : "dummyId",
4      "host" : "localhost",
5
6      "connection" : [
7  {   "in" : {
8                  "endpoint-filter" : "(toto=true)" ,
9                   "importer-filter" : "(instance.name=RoSe_importer.cxf)" ,
10                  "protocols" : ["ws"] }
11          },
12 {   "out" : {
13                 "service-filter" : "(ObjectClass=foo.bar.HelloWorld)" ,
14                  "exporter-filter" : "(instance.name=RoSe_exporter.jabsorb)" ,
15                  "protocols" : ["jsonrpc"] }
16 } ],
17
18       "instances" : [
19 {
20           "factory" : "RoSe_exporter.jersey",
21           "properties" : {"jersey.servlet.name" : "/rest",
22                           "instance.name" : "RoSe_exporter.jersey-default"}
23 }],
24    }
25 }
```

Listing 5.6 – Exemple de configuration via le langage déclaratif.

La grammaire du langage est décrite dans la partie 4.3.5.

5.3.3 Développement des Importer et Exporter

Si le langage déclaratif et l'API permettent de spécifier et de configurer la distribution durant l'exécution, il est indispensable de faciliter la conception et l'implantation des composants de communication tels que les exportateurs et importateurs.

Dans ce but, nous avons développé des classes génériques qui regroupent les interactions avec le canevas. Ainsi, pour développer un importateur, un développeur ne doit gérer

que la création et la destruction d'un proxy, idéalement en s'appuyant sur une librairie existante. L'exemple suivant montre l'effort nécessaire à la réalisation d'un importateur permettant la création de proxies pour des services Web grâce à l'utilisation de la librairie Apache CXF.

```
1  [...]
2  @Override
3  protected ServiceRegistration createProxy(EndpointDescription description,Map properties) {
4
5      ClientProxyFactoryBean factory;
6
7      //load the class from the description
8      final List<Class<?>>klass = loadClass(context, description);
9
10     //use annotations if present
11     if (klass.get(0).isAnnotationPresent(WebService.class)){
12         factory = newJaxWsProxyFactoryBean();
13     } else {
14         factory = newClientProxyFactoryBean();
15     }
16
17     //set the class corresponding to the web-service
18     factory.setServiceClass(klass.get(0));
19
20     //set the URL
21     factory.setAddress((String) description.getProperties().get(ENDPOINT_URL));
22
23     //create the proxy and register it
24     return registerProxy(context, factory.create(), description, properties);
25 }
26
27
28 // Nothing to release or destroy is the factory garbage collected.
29 @Override
30 protected void destroyProxy(EndpointDescription description,
31         ServiceRegistration registration) {
32     registration.unregister();//unregister the proxy
33 }
```

Listing 5.7 – Code métier d'un Importer basé sur Apache CXF.

Ainsi, idéalement, les développeurs doivent simplement implanter les méthodes `createProxy` et `destroyProxy`. De plus, nous fournissons la méthode `loadClass` qui permet de charger la classe du proxy depuis la description du service en utilisant les classloaders des bundles

présents sur la passerelle. Nous fournissons aussi une méthode qui facilite l'enregistrement de l'objet proxy comme service sur la passerelle (`registerProxy`). Néanmoins, la réalisation d'un importateur ou d'un exportateur n'est pas toujours si aisée. Premièrement, il est nécessaire que les librairies soient disponibles sous forme de bundle OSGi. De plus, la gestion des classes effectuée par ces librairies rentre souvent en conflit avec la gestion des classloaders dans OSGi. De manière générale, on arrive à contourner ces problèmes en manipulant les classloaders. Cependant, le conditionnement des librairies sous forme de bundle étant de plus en plus courante, ce type de problème tend à disparaitre.

5.3.4 Génération de Proxy et d'Endpoint

Dans cette section nous détaillons les différents protocoles de communication intégrés à RoSe qui permettent la génération automatique de proxys pour des services distants et de endpoints pour des services locaux. Plutôt que d'implanter des standards et des protocoles de communication, RoSe s'appuie sur des bibliothèques existantes. De cette manière, RoSe profite de la stabilité et des fonctionnalités proposées par ces bibliothèques tout en rendant transparent leur utilisation et configuration. Le tableau ci-dessous liste les différents protocoles intégrés à notre canevas permettant l'importation de services distants (création de proxy) et l'exportation de services locaux (création d'endpoint).

Protocole/Standard	Style	Librairie	Import	Export
JSON-RPC	RPC	Jabsorb.org	√	√
JAX-WS	WS	Apache CXF	√	√
JAX-RS	REST	Jersey		√
XML-RPC	RPC	Apache XML-RPC	√	√

TABLE 5.4 – Protocoles de communication intégrés à RoSe.

JSON-RPC

JSON-RPC est un protocole RPC conçu pour être simple et basé sur le format d'échange de données JSON. L'exportation et l'importation JSON-RPC de RoSe s'appuie sur la librairie `jabsorb.org` qui fournit une implantation Java du protocole. Grâce à cet exportateur, un service local peut ainsi devenir accessible au travers d'un endpoint JSON-RPC. Réciproquement on peut réifier un objet distant accessible via JSON-RPC sous la forme d'un service local grâce à l'importateur JSON-RPC. La sémantique Java et celle de JSON étant proches,

les informations contenues dans la description et le contrat sont suffisantes pour générer de manière automatique les proxies et endpoints.

L'exemple suivant montre l'implémentation d'un simple service avec iPOJO et comment on peut ensuite y accéder depuis un client JavaScript grâce à l'endpoint créé par RoSe.

```
1  @Component(name="acme.hello.component")
2  @Instanciate(name="helloService") //default instance
3  @Provides //Provide HelloService
4  public class MyComponent implements HelloService{
5
6      @Requires(optional=true) //require a Log Service
7      private LogService logger;
8
9      public String hello(String name){
10         return "Hello "+name+" !";
11     }
12
13     @Validate //on validation callback
14     private void start(){
15         logger.log(INFO,"HelloService started.");
16     }
17
18     @InValidate //on invalidation callback
19     private void stop(){
20         logger.log(INFO,"HelloService stopped");
21     }
22 }
```

Listing 5.8 – Exemple d'un HelloService implémenté avec iPOJO.

```
1  ..
2  var jsonrpc = new JSONRpcClient("/JSON-RPC");
3  var greeting = jsonrpc.helloService.hello("Dave");
4  $('#greeting').text(greeting);
5  ..
```

Listing 5.9 – Proxy JavaScript du endpoint créé par RoSe.

XML-RPC

Tout comme JSON-RPC, XML-RPC est un protocole qui implémente un protocole RPC basé sur le langage de balise XML. Le fonctionnement est en tout point similaire à JSON-RPC. L'implantation des exportateurs et importateurs XML-RPC repose sur la librairie Apache

XML-RPC.

JAX-WS

JAX-WS (*Java API for XML Web Services*) est une API Java permettant la création de services Web. RoSe s'appuie sur cette API et en particulier sur le framework Apache CXF pour permettre l'import de services Web et l'export de services locaux sous forme de services Web. Se reposer sur JAX-WS permet l'utilisation d'annotations sur le contrat ou sur la classe d'implantation du service. Ceci entraine une génération plus fine du service Web. On pourra, par exemple, faire en sorte que le service Web généré corresponde à un WSDL bien spécifique. De la même manière, en générant une interface annotée à partir d'un WSDL existant, on facilite la création du proxy. Les annotations sont optionnelles, mais deviennent indispensables si l'on veut intégrer l'application avec des applications existantes reposant sur des services Web. L'avantage de reposer sur les annotations CXF réside aussi dans le fait que les services utilisant ces annotations dépendent de la spécification JAX-WS et non pas d'une implantation particulière telle que CXF ou encore GlassFish Metro. L'exemple suivant montre l'utilisation d'annotation JAX-WS sur le contrat d'un service local.

```
1  import javax.jws.WebMethod;
2  import javax.jws.WebService;
3
4  @WebService
5  public interface HelloWorld {
6
7      /**
8       * @return <code>"Hello "+name+ "!"</code>
9       **/
10     @WebMethod(operationName="hello")
11     String hello(String name);
12
13 }
```

Listing 5.10 – Contrat annoté avec les annotations jax-ws.

JAX-RS

Tout comme JAX-WS, JAX-RS est une API Java. Elle permet la création de service Web RESTFul, c'est-à-dire se conformant aux principes du style REST. Contrairement aux services Web classiques et aux autres standards réifiant RPC, il est ici indispensable d'utiliser les annotations pour pouvoir exposer un service comme une ressource. En effet, contrairement aux services, les ressources sont basées sur une interface uniforme et ne disposent pas de contrat spécifique. Ainsi, il est indispensable que le développeur annote son POJO avec les annotations JAX-RS pour permettre un mapping entre le service et la ressource. RoSe s'appuie sur la bibliothèque Jersey pour exposer le service OSGi annoté comme une ressource REST. RoSe maintient alors le cycle de vie de la ressource synchronisée avec celui du service. L'exemple suivant montre un composant iPOJO annoté avec les annotations JAX-RS, le service fourni par le composant peut donc ainsi être exposé par RoSe sous forme d'une ressource grâce à l'ExporterService basé sur Jersey.

```
1  @Path(value="/helloworld/{name}")
2  public class HelloImpl implements HelloWorld {
3
4      @GET
5      @Produces("text/plain")
6      public String hello(@PathParam("name") String name) {
7          return "Hello "+name+" !";
8      }
9
10     @GET
11     @Produces("application/json")
12     public String helloJson(@PathParam("name") String name){
13         return "{hello : \""+hello(name)+"\"}";
14     }
15 }
```

Listing 5.11 – Composant annoté avec les annotations jax-rs.

5.3.5 Découverte

RoSe supporte divers protocoles pour la publication et la découverte de services distants. Comme pour les protocoles de communications, ces derniers ont été implantés en s'appuyant sur des bibliothèques existantes. Chaque composant de découverte est un pont entre la machine locale et les dispositifs physiques ou les machines distantes connectées

sur le réseau. En s'appuyant sur des protocoles s'appuyant sur des multicasts réseaux tel qu'UPnP ou dns-sd, jusqu'à des registres supportant de la synchronisation et de la redondance (tel que zookeeper), RoSe permet le développement d'applications distribuées faiblement couplées couvrant un domaine allant de l'informatique ubiquitaire jusqu'aux applications Web et largement distribuées.

Protocole/Standard	**Domaine**	**Publication**	**Découverte**
UPnP	Pervasive	√	√
DPWS	Pervasive	√	√
Dns-sd	Pervasive	√	√
Modbus	Pervasive, industriel		√
Comet-d	Web	√	√
Zookeeper	Web, industriel, cloud computing	√	√
Pubsubhubbub (R2D2)	Web, cloud computing	√	√

TABLE 5.5 – Protocoles de découverte intégrés à RoSe.

5.4 Conclusion

Ce chapitre a présenté quelques points importants de l'implémentation de RoSe. RoSe est aujourd'hui disponible en open source à l'adresse suivante `https://github.com/barjo/arvensis`. Le cœur du framework ainsi que plusieurs composants de base sont ainsi à la disposition des développeurs dans le but de faciliter leur travail. Nous décrivons dans le chapitre suivant plusieurs projet industriels et académiques dans lesquels RoSe a été utilisé avec succès et comment il a permis de répondre au besoin des développeurs en terme de facilité d'utilisation, de performance et de flexibilité à l'exécution.

Nous avons montré, comment, de par sa nature flexible, RoSe supporte l'ajout de nouveaux importeurs, exporteurs et composants de découvertes durant l'exécution. De plus, nous avons vu comment le framework facilite le développement de ces composants en s'appuyant sur des patrons de conception et en fournissant un certain nombre d'outils.

Le tableau suivant indique le nombre de lignes de code de la dernière version de RoSe ainsi que des importeurs, exporteurs et composant de découverte principaux fournit avec cette version.

	LOC - Java	LOC - XML	LOTest (Java + XML)	Total LOC
Core	2331	244	854	3429
Configurator	521	276	0	797
JSON-RPC	338	272	783	1393
XML-RPC	291	224	624	1139
JAX-RS	276	126	277	679
JAX-WS	244	230	405	879
Zookeeper	411	127	378	911
DNS-SD	328	145	345	818
Modbus - Discoverer	638	130	0	768
Pubsubhubbub	2305	168	2694	5167
Total	7683	1942	6360	**15985**

TABLE 5.6 – Nombre de lignes de code.

Chapitre 6

Validation

> "Un poète doit laisser des traces de son passage, non des preuves. Seules les traces font rêver." *René Char*

6.1 Introduction

Dans le chapitre précédent, nous avons décrit l'implantation du framework RoSe et montrer comment RoSe permet de créer des applications distribuées faiblement couplées et dynamiques aux dessus d'un modèle à composant orientés service.

Le présent chapitre s'intéresse à la validation du framework. Dans un premier temps, nous évaluons le coût d'utilisation de notre framework durant la phase de développement, puis nous évaluons les performances de notre framework durant l'exécution. Enfin, nous présentons plusieurs applications qui s'appuient sur RoSe pour la gestion de la distribution dans les domaines de l'informatique ubiquitaire, en particulier la domotique, ainsi que dans le domaine des applications Web. Ces applications ont été réalisées dans le cadre de projets de recherche collaborative au niveau Européen mais aussi dans le cadre de projets industriels.

6.2 Évaluation du coût d'utilisation au développement

Dans cette première section, nous nous intéressons à la complexité induite par RoSe lors de la phase de développement d'une application distribuée. Nous nous focalisons en

particulier sur le coût d'utilisation en termes de lignes de code introduites pour la gestion de la distribution et du dynamisme. Nous examinons également le couplage créé par l'utilisation du framework RoSe par rapport à d'autres technologies disponibles dans le monde Java.

6.2.1 Application test

Nous avons défini une application « test » pour évaluer le coût du framework RoSe. Le but de cette application est de fournir un service accessible par un grand nombre de clients Web. Le service retenu est la gestion de listes de tâches à réaliser, une 'TODO list' en anglais. Ce service, que nous appellerons `ToDoList`, permet de créer et d'ajouter des tâches (todo) dans une liste, d'en supprimer, et de consulter une liste. Une tache (todo) est caractérisée par un identifiant (id) et par un contenu textuel.

L'implantation de notre application repose sur un service technique de persistance basé sur SQLite [1]. L'application est réalisée en Java et consiste en quatre éléments principaux : l'interface `TODOList`, la classe `TODO`, la classe `TODOListImpl` qui concrétise l'interface TODOList et enfin l'interface du service de persistance `SQLITEService`.

La figure 6.1 présente un diagramme de classe reprenant les quatre principaux éléments et leurs relations.

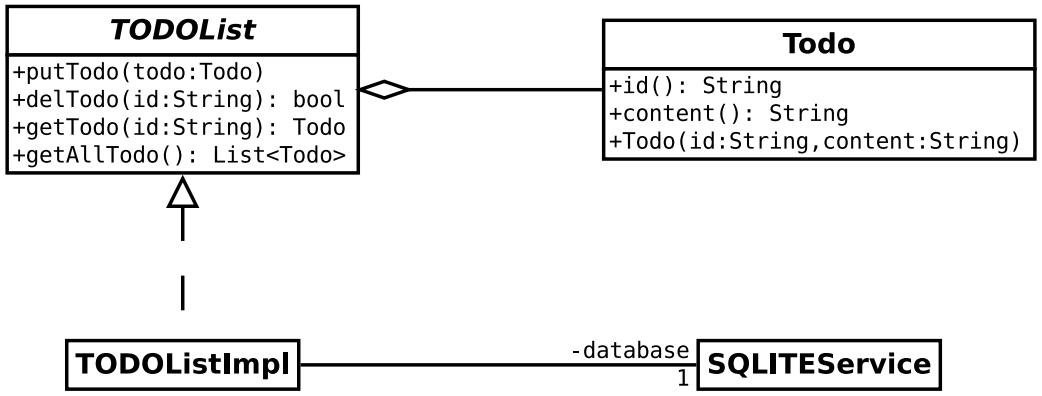


FIGURE 6.1 – Diagramme de classes de l'application TODOList.

1. `http://www.sqlite.org/`

Ci-après, nous présentons deux cas d'étude mettant en place différentes formes de distribution pour notre application test. Dans le premier cas d'étude, l'application TODOList repose sur un serveur unique et est exposée sous la forme d'une ressource REST. Dans le second cas d'étude, l'application est distribuée de manière transparente sur deux serveurs. Le premier serveur contient le code applicatif et expose le service TODOList sous forme d'un Endpoint JSON-RPC. Le second serveur héberge le JSONService et expose la TODOList sous forme d'une ressource REST, tout comme dans le premier cas d'étude.

Cas d'étude 1

Le premier cas d'étude repose sur l'exposition des listes sous forme REST de façon à ce qu'elles soient accessibles depuis des clients Web. RoSe est utilisé pour la génération du Endpoint permettant l'accès distant.

Le Endpoint est créé en utilisant les annotations JAX-RS (REST) et le service de sérialisation et de dé-sérialisation JSONService. Ce service permet de sérialiser des données vers le format de données JSON. Ainsi, les clients peuvent accéder à la TODOList aux travers de simples requêtes http. Trois types de requêtes sont supportées : les requêtes http PUT, GET et DEL qui correspondent respectivement à l'ajout d'un TODO, à la consultation d'un ou de l'ensemble des TODO et, enfin, à la suppression d'un TODO existant. Le tableau suivant illustre l'API REST ainsi créé.

URL	Type de requête	Description
/todolist	GET	Retourne l'ensemble des TODO présent sous forme d'un tableau json.
/todolist/{id}	GET	Retourne le TODO dont l'identifiant est `{id}` sous forme d'un objet json.
/todolist	PUT	Ajoute le TODO contenue dans la requête dans la TODOList.
/todolist/{id}	DEL	Supprime le TODO d'id `{id}` de la TODOList.

TABLE 6.1 – API REST.

Les composants de RoSe utilisés sont basés sur la librairie Jersey et utilise le `HttpService` basé lui-même sur le serveur Web Jetty. Le `JsonService` utilisé repose sur la librairie Json.org.

Ce sont donc les librairies de Jersey, Json.org, Jetty et SQLite qui sont utilisées pour la réalisation de l'application de référence. Mais, et c'est l'essentiel, ceci se fait de façon transparente pour le développeur de l'application ToDoList.

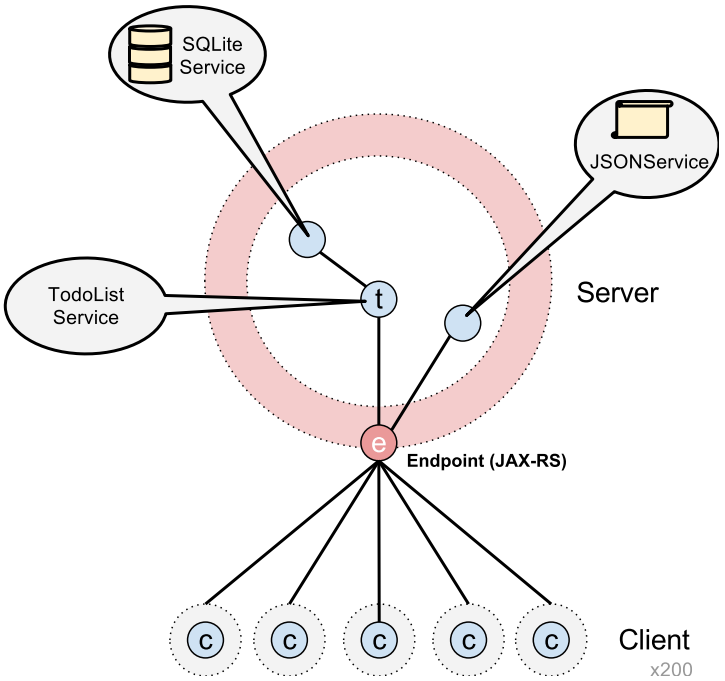


FIGURE 6.2 – Architecture de l'application (cas d'étude 1).

Cas d'étude 2

Dans le second cas d'étude, l'application est plus fortement distribuée. Elle est, en effet, exécutée sur deux serveurs et non plus un seul. Un premier serveur exécute le code applicatif et contient le service de persistance `SQLiteService`. Le service TODOList est exposé sous la forme d'un Endpoint JSON-RPC. Le second serveur accède à la TODOList grâce à un proxy JSON-RPC.

Tout comme dans le premier cas d'étude, le second serveur expose la TODOList sous la forme d'une ressource REST (Endpoint Jax-RS). Les clients peuvent ainsi accéder à la TODOList via la ressource REST fournie par le second serveur (ceci est équivalent au premier scénario) ou bien via l'endpoint JSON-RPC fourni par le premier serveur.

Chaque Endpoint est créé par RoSe, ainsi que le proxy JSON-RPC. La distribution entre les deux serveurs est réalisée de manière transparente, aucune annotation ou changement dans l'application présentée dans le premier cas d'étude n'est nécessaire. La figure 6.3 représente l'architecture de l'application pour ce deuxième scénario.

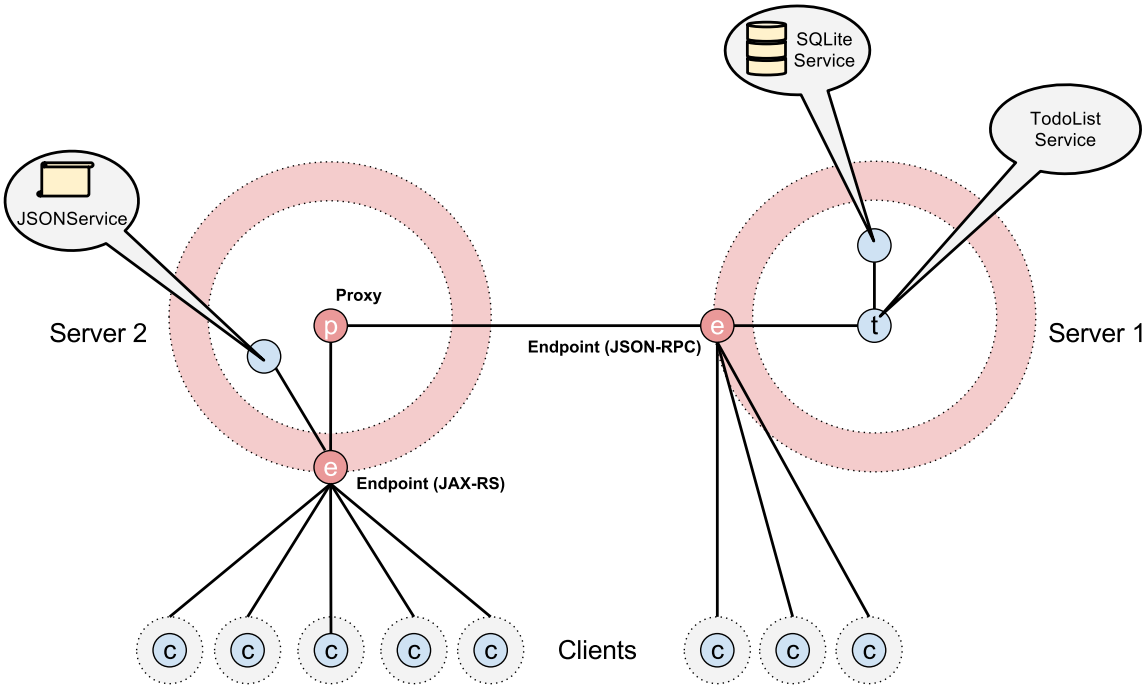


FIGURE 6.3 – Architecture de l'application (cas d'étude 2).

Il apparaît clairement que cette architecture est plus complexe au niveau de la distribution. Elle soulève également des problèmes en termes de dynamicité puisque les deux *cotés* de l'application peuvent évoluer séparément. Il n'est pas souhaitable pour la plupart des évolutions d'arrêter l'application pour conserver un bon niveau de disponibilité.

6.2.2 Protocole de test

Pour les deux cas d'étude précédents, nous avons cherché à évaluer l'intérêt de notre framework RoSe. Le code applicatif est le même pour les deux architectures. Nous nous focalisons donc uniquement sur le code lié à la distribution et à la gestion du dynamisme.

Afin d'évaluer l'impact du framework sur le développement de l'application nous nous intéressons aux lignes de code Java mais aussi aux lignes de code relatives à la configuration, présentes dans les fichiers de configuration. Pour évaluer le couplage, nous nous intéressons au nombre de fichiers et de lignes de code/configuration qu'il faut modifier pour changer la topologie de la distribution ou les protocoles de communication à utiliser. Nous examinons également l'impact sur ces éléments d'une modification du code applicatif. Enfin, nous indiquons aussi si ces changements peuvent être appliqués durant l'exécution ou s'ils nécessitent l'arrêt complet de l'application, la compilation, le déploiement et enfin le redémarrage.

Les classes `TODOListImpl` et `TODO` sont implantées sous la forme de *simples* objets Java. Le service SQLite est un *wrapper* autour de la technologie d'accès de base de données Java JDBC. Ce code est commun à toutes les plateformes et technologies testées. Seuls des changements mineurs comme l'ajout de métadonnées iPOJO ou encore la configuration du serveur JEE utilisé sont à noter. Ces modifications ne sont pas prises en compte dans notre évaluation.

Nous nous intéressons donc principalement au code lié à la distribution de l'application. Pour le premier cas d'étude, ce code consiste à la création d'une représentation REST pour l'objet `TODOList`. Dans le second cas, il faut non seulement créer la représentation REST mais aussi permettre l'interaction entre les deux serveurs, c'est-à-dire la création des endpoint et proxy JSON-RPC et éventuellement des mécanismes de découverte entre les deux machines.

6.2.3 Plateformes et technologies testées

Les deux cas d'étude ont été réalisés avec Java, le serveur d'application JavaEE GlashFish et le framework RoSe. Le code Java, et les configurations JavaEE et RoSe du cas d'étude 1 sont disponibles dans l'annexe B.

Framework	LOC	LOConfiguration
Java	22	0
JavaEE	0	21
RoSe	0	15

TABLE 6.2 – Lignes de code et configuration (cas d'étude 1).

6.2.4 Résultat et analyse

Cas d'étude 1

Comme illustré par la table 6.2, RoSe permet un léger gain en terme de lignes de code par rapport à Java et JavaEE pour le premier cas d'étude. Ce qui importe en particulier c'est la nature même du code qui permet la distribution. Dans le cas le plus simple, Java, le code permet de démarrer le serveur web et de l'utiliser pour publier la classe annotée avec les annotations JaxRs. Dans le cas de JavaEE, ces étapes se font au travers d'un fichier de configuration (`web.xml`), JaxRs étant une spécification standard JavaEE, il suffit de spécifier une Servlet qui fait office d'adaptateur, ainsi que le package contenant la classe annotée avec les annotations JaxRs. Finalement, dans le cas de RoSe, on créé une connexion out qui exporte le service annoté avec les annotations JaxRs.

Si RoSe et JavaEE permettent une meilleure séparation des préoccupations par rapport à Java et plus de flexibilité grâce à leur fichier de configuration, seul RoSe s'avère être réellement flexible. En effet, si l'on veut publier une autre ressource, il suffit de changer le filtre de la connexion out déjà spécifiée ou alors d'en ajouter une autre. Dans le cas de JavaEE, la nouvelle ressource devra embarquer sont propre fichier de configuration, le nombre de fichier de configuration est donc linéaire par rapport aux nombre de ressources alors qu'il reste identique pour RoSe. De plus, ces modifications peuvent être effectuées durant l'exécution, alors que dans le cas de JavaEE, il faut redéployer l'application à chaque modification du fichier de configuration.

Cas d'étude 2

Dans ce second cas d'étude, Java et JavaEE nécessite quasiment le double de lignes par rapport à RoSe. Mais, ici encore, c'est la nature de ces lignes qui importe plus que leur quantité. En effet, dans ce second cas, Java et aussi JavaEE nécessitent de modifier le code

Framework	LOC	LOConfiguration
Java	56	0
JavaEE	17	43
RoSe	0	34

TABLE 6.3 – Lignes de code et configuration (cas d'étude 2).

métier de l'application, dans les deux cas le proxy et le endpoint doivent être générer via l'utilisation d'une librairie tierce au sein du code métier. Ces lignes de code introduisent un couplage particulièrement fort avec ces bibliothèques et s'avèrent être un réèl handicap pour la maintenance et l'évolution. Au contraire, RoSe permet cette nouvelle distribution entre les deux serveurs en ajoutant simplement une nouvelle connexion in, et bien sûr une nouvelle configuration pour la seconde machine. De plus, ici aucun couplage avec les bibliothèques ou les standard, on peut par exemple utiliser SOAP plutôt que JSON-RPC en changeant simplement l'élément *protocols* dans les fichiers configuration des machines. Un second point crucial, est l'absence de support protocole de découverte entre les machines, dans le cas de Java et JavaEE. Sans protocole de découverte, les machines ne sont pas notifiées des changements de disponibilité, ce qui limite encore la flexibilité à l'exécution.

Ces deux cas d'études relativement simples nous ont permis d'illustrer comment RoSe facilite la conception, mais aussi l'évolution et la maintenance des applications distribuée en comparaison des approches Java et JavaEE. Dans les sections suivantes, nous reprenons le premier cas d'étude et évaluons l'impact de RoSe sur les performances par rapport à ces technologies.

6.3 Évaluation et comparaison des performances de RoSe

Dans cette section nous évaluons le surcoût introduit par l'utilisation du framework RoSe au travers d'une simple application *client/server* s'exécutant sur le *Cloud*. L'expérimentation est ensuite reproduite avec la même application réalisée dans des technologies purement Java et finalement avec un framework de type JavaEE.

6.3.1 Application de test

Nous reprenons la même application (TODOList) que celle décrite dans la partie précédente, et nous nous intéressons cette fois à l'impact de notre framework sur les performances. Nous reportons ici seulement l'évaluations des performances du premier cas d'étude (6.2.1).

6.3.2 Banc d'essai

Pour quantifier l'impact de notre Framework dans le *Cloud* (Amazon EC2), nous avons effectué des mesures et reporté les résultats. Nous nous intéressons à l'impact mémoire introduit par le framework RoSe en comparaison d'une application Java classique, mais aussi par le surcout éventuel dans le traitement de chaque requête.

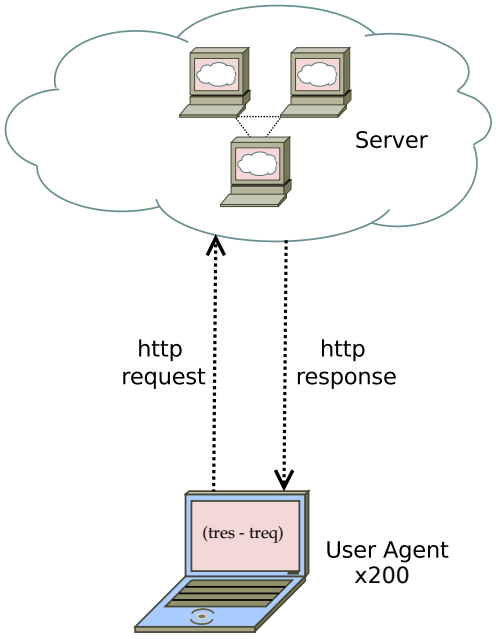


FIGURE 6.4 – Banc d'essai.

Serveur L'expérimentation a été conduite sur une micro instance Amazon EC2. Une micro instance offre 613MB de RAM, 8GB de disque et jusqu'à 2 EC2 *Computing Units*. Une *Computing Unit* EC2 fournit l'équivalent d'un processeur Xeon 2007 de 1.0-1.2Ghz. Le système d'exploitation est un linux 2.6.35 64-bits (avec la distribution RedHat). Nous utilisons Java 1.6.0 avec la machine virtuelle OpenJDK server 64-bits.

Client Les clients sont deux ordinateurs portables Dell latitudes E6500, disposant de 4 Go de RAM et d'un processeur Intel T9800 (2x2.93Ghz). Le système d'exploitation est un linux 32-bits (Ubuntu). Les requêtes sont exécutées grâce à la libraire perl WWW qui permet de créer des browsers virtuels (perl v5.10.1).

6.3.3 Protocole de test

Chaque PC client exécute 100 agents dans des processus parallèles. Chacun des agents exécute 50 requêtes HTTP GET et 10 requêtes HTTP PUT, soit un total de 12000 requêtes. Le serveur est hébergé sur Amazon EC2 Ireland et les clients dans les bureaux de l'équipe Adèle à Grenoble.

Consommation mémoire et processeur

Afin de mesurer la consommation mémoire et processeur, nous utilisons la console de monitoring Jconsole fournie par Java. Grâce à la Jconsole nous suivons et enregistrons la consommation mémoire ainsi que la consommation processeur liées au processus Java qui exécute l'application.

Temps de traitement des requêtes

Des estampilles temporelles (*timestamp*) sont associées à chaque exécution de requête (elles sont dénotées t0 avant l'exécution et t1 à la fin de l'exécution). Les estampilles t0 et t1 sont ensuite journalisées ainsi que le code de réponse de la requête. Chaque entrée est indexée par le temps t0 concaténé au temps t1. Le graphique ? représente la courbe de temps d'exécution des requêtes avec la valeur (t1 – t0) en ordonnée et le numéro de la requête en abscisse. L'annexe C contient le script perl qui créé les agents et les exécute dans des *forks*.

6.3.4 Plateformes et technologies testées

Comme pour l'évaluation précédente, l'expérience a été conduite sur une version purement Java de l'application, une version JavaEE (GlassFish Server Open Source Edition 3.1.2 **Web Profile**) et une version avec le framework RoSe.

Framework	Mémoire max (mb)	CPU moyenne (%)	CPU max (%)
Java	30.15	54.53	99
JavaEE	58.57	52.91	99
RoSe	37.24	56.57	99

TABLE 6.4 – Consommations mémoires et CPU (cas d'étude 1).

Framework	Moyenne des temps (s)	Médiane des temps (s)
Java	8.116	0.655
JavaEE	10.219	0.616
RoSe	10.749	0.477

TABLE 6.5 – Temps d'exécutions des requêtes (cas d'étude 1).

6.3.5 Résultat et analyses

Le tableau 6.4 nous permet de constater qu'il n'y a pas de différence majeure entre Java, JavaEE et RoSe au niveau des consommations mémoire et processeur. Le serveur JavaEE à une empreinte significativement supérieure à RoSe et Java, mais reste raisonnable au vue des fonctionnalité supplémentaires fournie (notamment d'administration). RoSe consomme légèrement plus de processeur, 2% de plus que Java et 3% de plus que JavaEE.

De la même manière, le temps de traitement des requêtes ne fluctue que très peu selon le framework utilisé. Nous pouvons voir dans le tableau 6.5 que si l'on considère la moyenne ou la médiane, l'avantage est à Java ou bien à RoSe. La distribution des temps d'exécution des requêtes telle qu'illustrée par la figure 6.5 est cohérente avec le reste des données, et ne donne pas d'avantage distinct à Java, JavaEE ou RoSe. La moyenne élevée du temps de traitement des requêtes s'expliquent de par la nature du protocole de test (les clients sont des *forks*), mais aussi par le fait que la base de donnée utilisée est non multithreadée.

Cette section nous a montré que RoSe n'implique pas de surcôut particulier durant l'exécution par rapport à Java et JavaEE. Ainsi, la facilité de conception et la flexibilité à l'exécution sont apportées par RoSe à un moindre coût. Dans la deuxième partie, nous évaluons comment RoSe réagis lorsque que des changements (mise à jour, reconfiguration, etc…) ont lieu durant l'exécution.

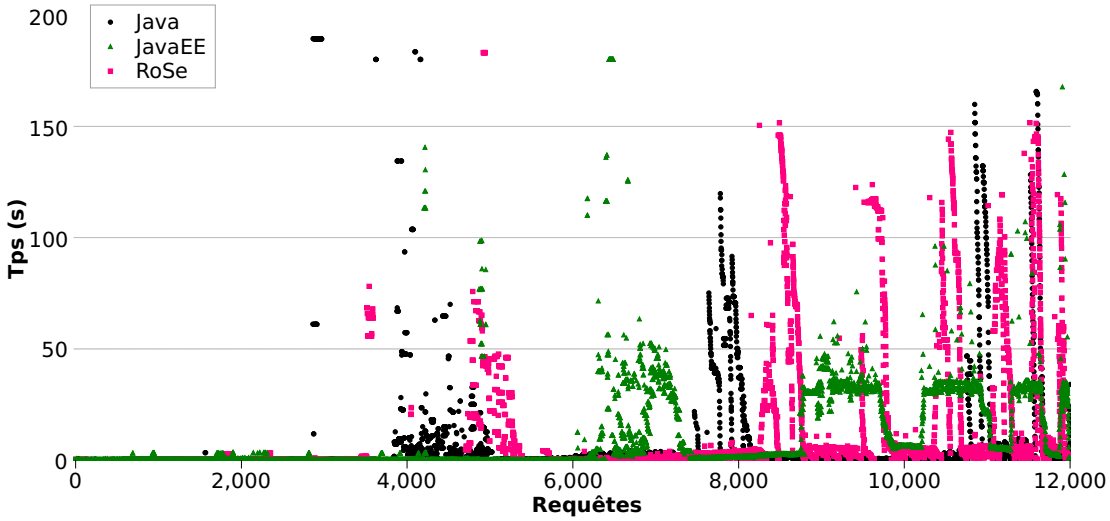


FIGURE 6.5 – Temps d'exécution des requêtes (cas d'étude 1).

6.4 Évaluation de la flexibilité à l'exécution

Une des particularités de notre framework est sa flexibilité à l'exécution. C'est-à-dire qu'à tout moment durant l'exécution, les aspects liés à la distribution peuvent être changés. Ainsi, on peut changer la configuration via le langage déclaratif, mais aussi ajouter, mettre à jour ou encore retirer des *exporteurs* et *importeurs* dynamiquement. De même, des composants de découvertes et publication peuvent être ajoutés, retirés, ou mis à jour durant l'exécution. La volatilité de la disponibilité des services locaux et distants impactent la création et la destruction des proxies et endpoints durant l'exécution.

Dans cette section, nous évaluons donc l'impact du dynamisme sur RoSe. Nous nous intéressons en particulier au temps nécessaire pour réifier les changements intervenant durant l'exécution. Ces changements regroupent : le changement de la configuration, l'arrêt d'un service local exporté, l'apparition d'un service local qui doit être exporté, l'ajout d'un exporter/importer, le retrait d'un exporter/importer, la découverte d'un service distant à réifier.

6.4.1 Application de test

Nous reprenons le second cas d'étude décrit dans la section 6.2.1.

6.4.2 Protocole de test

Dans cette section nous évaluons l'impact de divers changement sur le temps de traitements des requêtes. Pour ce faire, nous effectuons quatre changements dynamiques alors que les requêtes sont en cours d'exécution. Ces changements sont :

1. la mise à jour du service de log (`LogService`),
2. la mise à jour de l'*exporter* JAX-RS,
3. l'ajout d'une connexion out publiant la TodoList comme un service web,
4. le changement du filtre de la connexion out qui expose la TodoList comme une ressource REST.

6.4.3 Résultat et analyse

Framework	Moyenne des temps (s)	Médiane des temps (s)
RoSe	26.00	0.54

TABLE 6.6 – Temps d'exécutions des requêtes (cas d'étude 2).

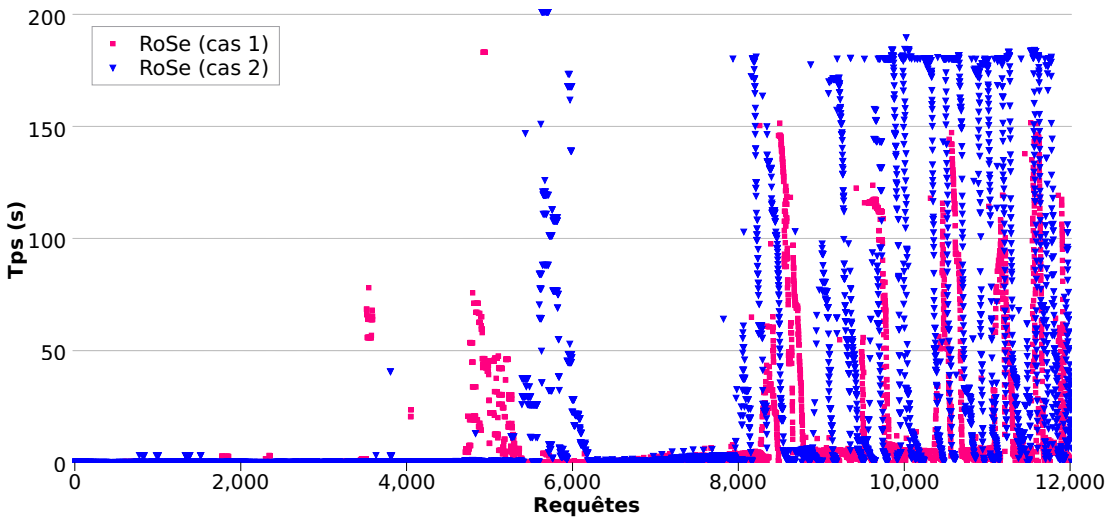


FIGURE 6.6 – Temps d'exécution des requêtes (RoSe cas d'étude 1 et 2).

Malgré le déclenchement des quatre changements durant l'exécution des requêtes, les performances restent comparables au cas d'étude précédent. Si la moyenne de temps de

traitement a doublé, la médiane n'augmente que de peu (0.54 contre 0.477). Ceci s'explique car les mises à jour 2,3 et 4 causent des changements dans la distribution et retardent ainsi les requêtes en cours d'exécution. Nous superposons dans la figure 6.6 les temps de réponse du scénario précédent (cas 1) avec celui ci (cas 2), bien que le cas 2 soit plus coûteux en terme de temps, les deux courbes restent dans le même ordre de grandeur.

En pratique, de telles mises à jour seraient déclenchées lors des périodes de creux, c'est-à-dire lorsque l'applications est peu utilisée, afin d'en minimiser les impacts. Néanmoins, certaines adaptations doivent pouvoir se faire en période de surcharge, et nous montrons ici que RoSe est capable de parer à de telles éventualités avec un impact réduit sur l'exécution.

6.5 Utilisation de RoSe dans le domaine ubiquitaire

Le framework RoSe a été utilisé avec succès pour la création d'applications ubiquitaires. C'est le cas, en particulier dans le domaine de la domotique au travers du projet Minalogic Medical[2] et des projets européens ITEA ANSO[3] et OSAMI[4]. Des entreprises telles que France Telecom et Schneider Electric intègrent aujourd'hui RoSe pour faciliter la conception, la création et la maintenance d'applications capables d'interagir avec des dispositifs réseaux hétérogènes et dynamiques.

Nous décrivons dans cette section deux projets où RoSe a permis de simplifier le développement d'applications. Nous commençons par décrire la passerelle domotique H-Omega [44], puis dans un second temps, le framework autonomique fANFARE [97].

6.5.1 H-Omega

H-Omega est une plateforme interne à l'équipe Adele (`http://www-adele.imag.fr/`) destinée à la création et à l'évaluation d'applications ubiquitaires, en particulier domotiques. Plus précisément, H-Omega fournit un environnement d'exécution approprié aux applications résidentielles (de la gestion du chauffage jusqu'aux applications multimédia). Le runtime d'H-Omega est basé sur une plateforme iPOJO (composants orientés service),

2. `http://medical.imag.fr`
3. `http://anso.vtt.fi/`
4. `http://www.osami-commons.org/`

un ensemble de services communs, et RoSe qui permet de réifier les dispositifs disponibles dans la maison sous forme de services iPOJO dans la plateforme.

Contexte d'utilisation

H-Omega utilise RoSe pour répondre aux problèmes des interactions transparentes avec des services dynamiques dans un environnement pervasif distribué. Les services considérés peuvent être fournis par équipements embarqués communicants présents dans la maison ou bien par des serveurs accessibles par Internet. Comme nous l'avons déjà vu, la complexité introduite par les interactions avec de telles entités volatiles et hétérogènes est particulièrement élevée.

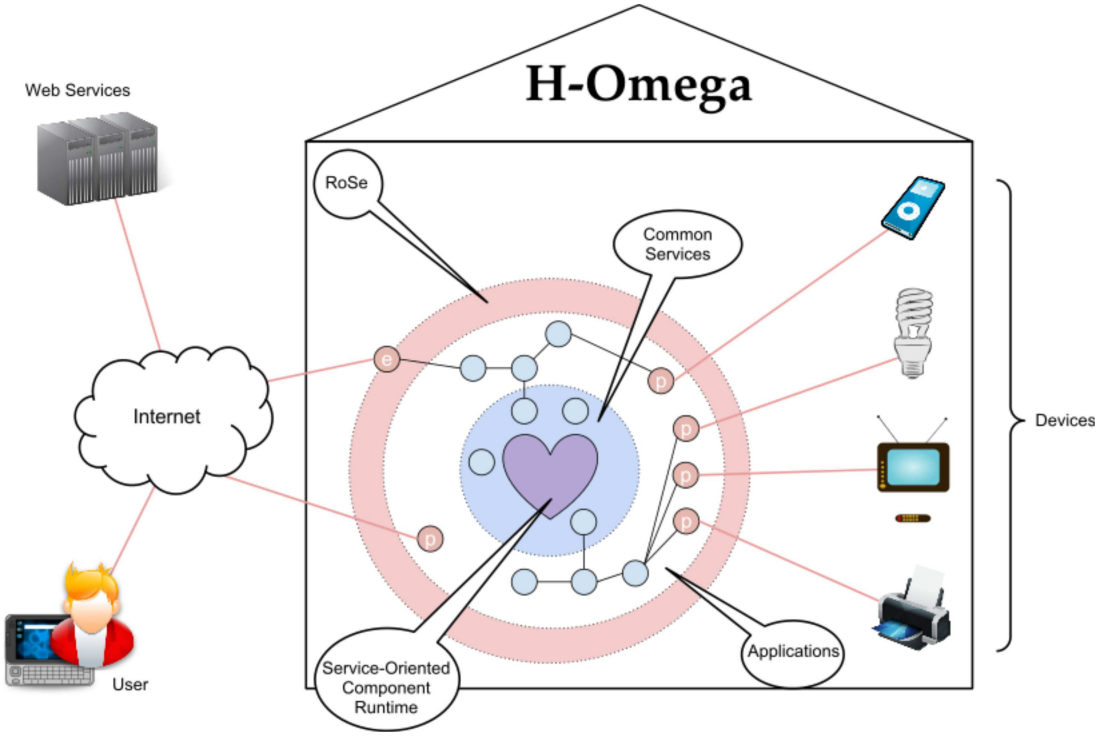


FIGURE 6.7 – Le projet H-Omega.

Pour pouvoir être utilisé avec succès dans le contexte des applications domotiques, RoSe à du répondre aux stricts besoins de ce domaine, ce qui inclut :

- La dynamicité : la disponibilité des services distribués évolue au cours du temps suite à divers événements tels que l'apparition ou la disparition d'un dispositif mobile, une panne de batterie, des problèmes réseaux ou plus simplement des actions explicites des usagés.
- La distribution : que cela soit aux travers du réseau local à la maison ou d'internet, la communication avec les services distants pose les problèmes classiques liés à l'informatique distribuée déjà abordés dans ce document.
- L'hétérogénéité : le marché de la domotique est marqué par une impressionnante variété de matériels, de systèmes, de logiciels et de protocoles qui varient grandement d'un dispositif à un autre.
- Les contraintes liées à l'informatique embarquée : enfin, le framework doit pouvoir s'exécuter sur des environnements contraints à la fois au niveau de la puissance de calcul et de la mémoire. H-Omega doit pouvoir fonctionner sur une passerelle résidentielle embarquée à faible empreinte énergétique.

Résultats et impacts

L'intégration de RoSe au sein de la plateforme H-Omega a permis de valider notre approche dans le domaine des applications domotiques. Dans [10] nous montrons comment RoSe a permis d'étendre la plateforme H-Omega et ainsi permettre aux développeurs H-Omega de concevoir des applications communicant avec des équipements UPnP ou encore des web-services, sans impacter leur modèle de programmation. Ces applications sont intrinsèquement capables de s'adapter au dynamisme lié à la volatilité des équipements. De plus, le langage de configuration dédié à la gestion de la distribution clairement séparé du code applicatif permet de gérer les aspects communications et découvertes sans intervenir sur le reste de l'application.

Ainsi, RoSe est devenue une brique essentiel de plusieurs projets liés à la domotique. En effet, divers projet tels que H-Omega s'appuient sur RoSe pour gérer la réification de dispositif hétérogènes et volatiles dans un modèle à composant orienté service. On peut citer :

- Dynamo : un framework pour la composition autonomique d'interfaces [9].

– Cilia : un framework de médiation embarquable pour l'intégration dynamique de services [103].

6.5.2 fANFARE

fANFARE [97] est un framework qui permet la gestion autonomique d'applications orientées services dans des environnements ubiquitaires. Cette gestion autonomique cible principalement la configuration et l'optimisation d'applications ubiquitaires déployées sur des plateformes OSGi. fANFARE a été développé dans le cadre du projet MEDICAL, financé par le ministère français de l'industrie.

Contexte d'utilisation

fANFARE permet l'administration d'applications orientées service pervasives grâce à une hiérarchie de gestionnaires autonomiques (*autonomic managers*). Ces gestionnaires comprennent un gestionnaire de plateforme et plusieurs gestionnaires d'application et de dépendances :

– Le gestionnaire de plateforme gère le comportement global de la plateforme en optimisant de manière conjointe différentes applications. Pour ce, il dirige les gestionnaires d'applications et de dépendances grâce à des buts exprimés par un administrateur aux travers d'un classement défini par des propriétés non fonctionnel tel que le coût par utilisation, la localisation, la fiabilité ou encore la sécurité.
– Les gestionnaires d'applications sont en charge d'un ensemble de composants et de leur interaction avec le reste du framework. Pour gérer ces interactions, ils organisent les services fournis et requis par l'ensemble des composants qui leur sont associés sous la forme d'un ensemble partiellement ordonné en utilisant l'algorithme FCA (*Formal Concept Analysis* [58])
– Finalement, les gestionnaires de dépendances (DM) sont responsables de la sélection finale des services. Pour cela, ils s'appuient sur la structure FCA et sur les objectifs fournis par le gestionnaire d'application.

Tout comme dans le projet H-Omega, RoSe est utilisé pour la gestion des équipements hétérogènes distribués. fANFARE s'appuie sur RoSe pour la découverte de services distants

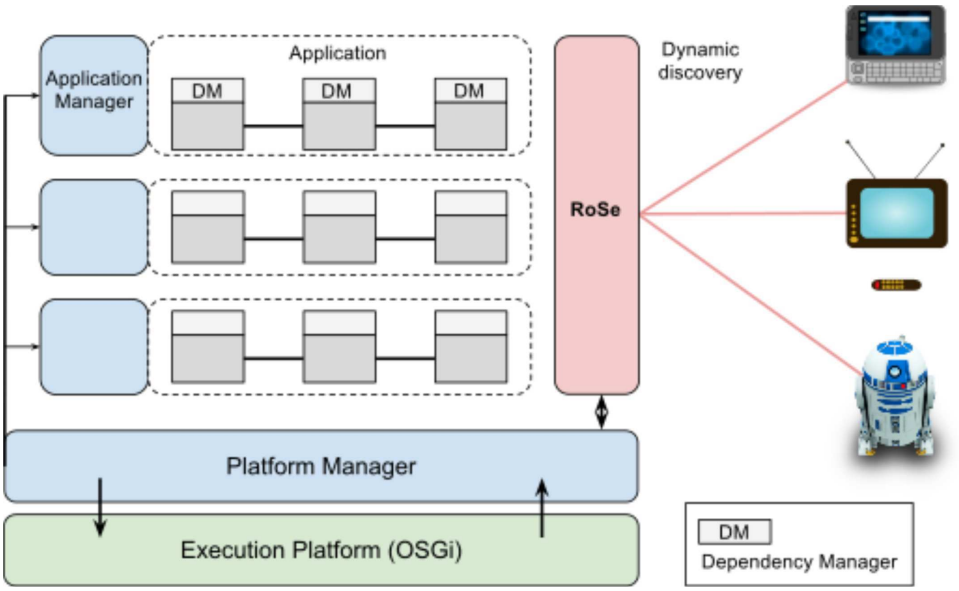


FIGURE 6.8 – Vue d'ensemble de fANFARE.

et la réification au travers de proxy de ces services distants au sein de la plateforme. Ainsi, les applications accèdent de manière transparente aux équipements distribués aux travers de proxies gérés par RoSe.

Résultats et impacts

Plusieurs travaux ont mis en évidence le besoin et l'intérêt de concevoir des gestionnaires autonomiques qui permettent d'adapter les applications à divers changements non fonctionnels qui peuvent intervenir durant l'exécution [83, 61]. Ces gestionnaires permettent d'adapter l'application en accord avec des buts de haut niveau exprimés par des administrateurs. Les buts peuvent prendre la forme d'un ensemble de contraintes sur la localisation, les performances ou encore la sécurité. Ces approches ont été proposées avec succès pour des applications à composants orientés service et s'avèrent être particulièrement prometteuses pour des domaines tels que l'informatique ubiquitaire [38, 23]. Néanmoins, il est nécessaire d'adresser le problème de réification des équipements hétérogènes et volatiles sous formes de services. C'est bien évidemment ici que RoSe intervient. En effet, comme le montre le projet fANFARE, il est indispensable de pouvoir gérer et piloter la réification de ces équipements durant l'exécution.

Ici, l'intérêt premier d'utiliser RoSe est lié à l'API 'fluent' qui permet aux gestionnaires autonomiques de gérer la distribution de manière programmatique, sans aucun couplage avec des technologies liées à la distribution ou à la découverte. Cet intérêt est renforcé du fait que chaque modification peut être effectuée durant l'exécution, ce qui est indispensable puisque c'est justement durant l'exécution qu'interviennent les actions des gestionnaires autonomiques.

La plateforme fANFARE est actuellement utilisée dans le projet MEDICAL. De futurs travaux sont envisagés afin de permettre la gestion autonomique d'un ensemble de plateforme. RoSe interviendrait alors au niveau de la gestion des communications entre différentes plateformes et permettrait ainsi la synchronisation des gestionnaires de plateforme et d'applications.

6.6 Utilisation de RoSe dans le domaine des applications web

RoSe est utilisé massivement par akquinet[5] dans différents contextes. Cette section décrit deux utilisations :

- Une plate-forme de communication pour une application de gestion de flotte,
- ChameRIA un framework permettant d'implanter des applications dites *desktop* avec des technologies Web.

Toutes ces utilisations reposent sur la flexibilité apportée par RoSe et sur sa capacité à masquer la complexité de la distribution et du dynamisme. La proximité du développeur principal a également joué afin de faciliter l'intégration. Cependant, au regard des particularités des systèmes développés, RoSe a été étendu afin de satisfaire certains besoins particuliers. L'extensibilité de RoSe a permis ces extensions et a rendu élégante et facile l'intégration de technologies comme Bluetooth, JMS et Web Socket dans des applications Java.

5. `http://www.akquinet.de`

6.6.1 Plate-forme de communication

RoSe est utilisé dans des systèmes complexes de remontées de données. Akquinet utilise OW2 Chameleon et RoSe dans une passerelle de communication. Cette passerelle est déployée dans une application de gestion de flotte pour Still [6]. Still loue des chariots élévateurs à de nombreux clients. L'utilisation de ces chariots est suivie et les chariots peuvent être reconfigurés à partir d'une application centrale. Aujourd'hui cette application gère 5 000 chariots à travers le monde. Ce chiffre devrait prochainement atteindre la valeur de 10 000 chariots. Still utilise les données remontées afin de planifier la maintenance des chariots, les licences et autorisations des conducteurs ainsi que les primes d'assurance. Dans chaque entrepôt, une passerelle est déployée afin de remonter les données et d'installer les reconfigurations.

Rose est déployé sur l'ensemble des passerelles de communication. RoSe est utilisé pour deux types d'interaction :

- Les interactions entre les serveurs centraux et la passerelle,
- Les interactions entre les chariots et la passerelle.

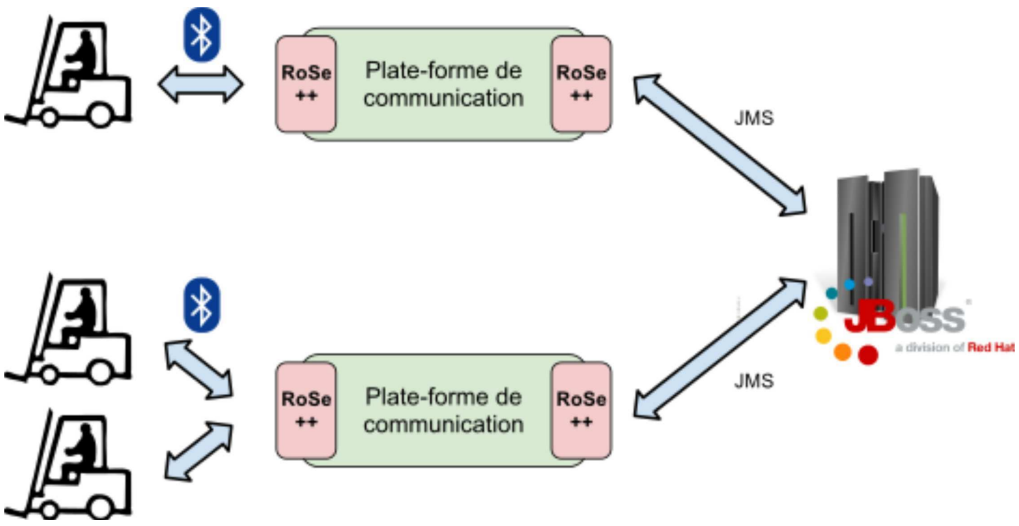


FIGURE 6.9 – Vue d'ensemble de l'application de gestion de flotte.

6. http://www.still.de

Dans le premier cas, l'utilisation de RoSe est statique. L'adresse des serveurs centraux étant connu, le canal de communication est fixe. Cependant, suite à de nombreux problèmes de connexions Internet et de l'initialisation de la connexion (sécurité), cet usage doit dorénavant prendre en compte le dynamisme. Pour des raisons de passage à l'échelle, toutes les interactions sont asynchrones et reposent sur JMS. Cet aspect provient d'un problème de passage à l'échelle coté serveur qui ne peut traiter qu'un nombre limité de requêtes. En plus de JMS pour les interactions et suite aux problèmes de dynamisme, un annuaire de service spécifiquement développé pour cet usage a été mis en place. Grace à l'extensibilité de RoSe, ce nouvel annuaire a été très rapidement intégré.

La flexibilité de RoSe est illustrée par l'utilisation du framework dans un cas totalement différent sur la même plate-forme. En effet, une catégorie de chariots élévateurs utilise Bluetooth pour communiquer avec la passerelle. Bluetooth (`http://bluetooth.com`) est une pile de communication pour dispositifs physiques. Lorsqu'un chariot est à porté de la plate-forme, la remontée de données et la reconfiguration sont effectuées. Les interactions utilisent le service Bluetooth OBEX, mais la communication est sécurisée par authentification et cryptage (*Bluetooth Level 2 Security Layer*). RoSe ne supporte pas Bluetooth par défaut et l'écosystème RoSe ne propose pas de support pour Bluetooth. Akquinet a donc développé des extensions pour RoSe afin de supporter cette communication. Basé sur bluecove (`http://bluecove.org`), une implémentation libre de la JSR 82 (intégration de Bluetooth dans Java), différentes extensions à RoSe ont été développées. Tout d'abord un composant de découverte a été mis en place. Bluetooth ne supporte que la découverte active. De plus cette découverte diffère selon les systèmes d'exploitations. Des contraintes métiers ont également été ajoutées afin de sélectionner les chariots pris en compte. La flexibilité de RoSe a permis de prendre en compte ce type de découverte. Un pilote a également été développé pour gérer les communications entre la plate-forme et les chariots. Ce pilote publie un service *OBEX* (exposé dans l'annuaire de service d'OSGi) pour chaque chariot. Afin de satisfaire les contraintes de concurrence de Bluetooth (*mono-thread*), ces interactions sont mises dans une file d'attente est traitée séquentiellement [7]. Malgré la complexité inhérente à Bluetooth, RoSe a permis une intégration élégante de cette pile de communication dans l'application sans impact direct sur le code métier. Cette contrainte est très importante car

7. A noter que les requêtes de découvertes de dispositifs et de services doivent également utiliser la même *thread*.

il permet un meilleur découpage de l'équipe de développement, et donc une meilleure efficacité.

6.6.2 ChameRIA

L'utilisation précédente a montré l'utilisation de RoSe dans un contexte complexe mais classique. Cette deuxième utilisation est plus innovante. ChameRIA est un framework, développé par akquinet, permettant de développer des applications dites *desktop* en utilisant des technologies webs [11]. L'idée de ChameRIA provient de deux constats :

- Il devient difficile de trouver des ingénieurs maitrisant correctement les technologies natives pour le développement *desktop* comme Java Swing. Les ingénieurs sont beaucoup mieux formés et plus efficaces avec des technologies Web.
- Les récentes évolutions des technologies Web, notamment HTML 5, permettent de développer des applications riches.

Cependant, les applications *desktops* ont des cas d'utilisations difficilement implantables avec les technologies webs comme :

- L'accès à des dispositifs connectés à l'ordinateur (port série, USB).
- L'utilisation de code natif (code C).
- L'accès au système de fichier.
- L'intégration dans le système d'exploitation (associations avec des fichiers).

ChameRIA répond à ces besoins en proposant une solution hybride. ChameRIA est une modification de OW2 Chameleon pour le développement d'application *desktop* :

- L'interface graphique est développée avec des technologies Web (JavaScript, HTML 5, CSS)
- La logique peut être développée soit en JavaScript soit en Java.
- L'application s'exécute dans un containeur OSGi/iPOJO et donc suit les principes de modularité proposés par ces technologies.
- L'interface s'exécute au sein du navigateur web embarqué (WebKit) dans ChameRIA et supportant les dernières spécifications (HTML 5 et Web Sockets). Cette fenêtre est gérée intégralement par le chameRIA.

– Les interactions avec le système d'exploitation sont gérées par le framework.

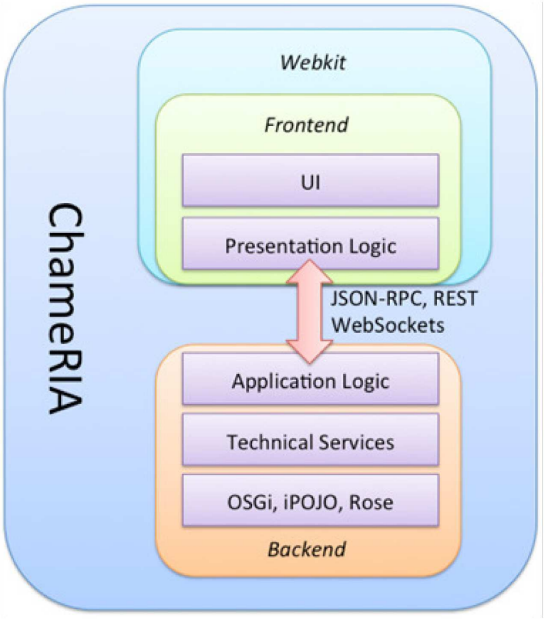


FIGURE 6.10 – Vue d'ensemble de ChameRIA.

ChameRIA permet donc d'utiliser des technologies Web modernes (HTML 5, H-Ubu, CSS 3, Web Sockets) pour développer des applications *desktop* totalement intégrées dans le système d'exploitation. Le développement d'applications avec ChameRIA a montré une efficacité remarquable. Grâce au découpage en couche et à l'utilisation de technologies différentes, l'ensemble des développeurs peut se focaliser sur leurs points forts. Les designers utilisent les technologies webs de leur choix. Les développeurs webs ont un accès simple aux services métiers développé par les ingénieurs *backend* en Java.

Plusieurs applications ont été développées avec ChameRIA dans différents contextes comme le contrôle de valves pour la distribution de l'eau, et l'implantation du lecteur de livre électroniques.

ChameRIA utilise Rose principalement pour réifier les services métiers dans la couche de présentation (en JavaScript), et ainsi permettre des interactions efficaces et simples entre les deux parties d'une application ChameRIA. En effet, grâce à l'API de RoSe chaque service métier (annoté) est exposé soit sous la forme d'un service JSON-RPC, soit sous la

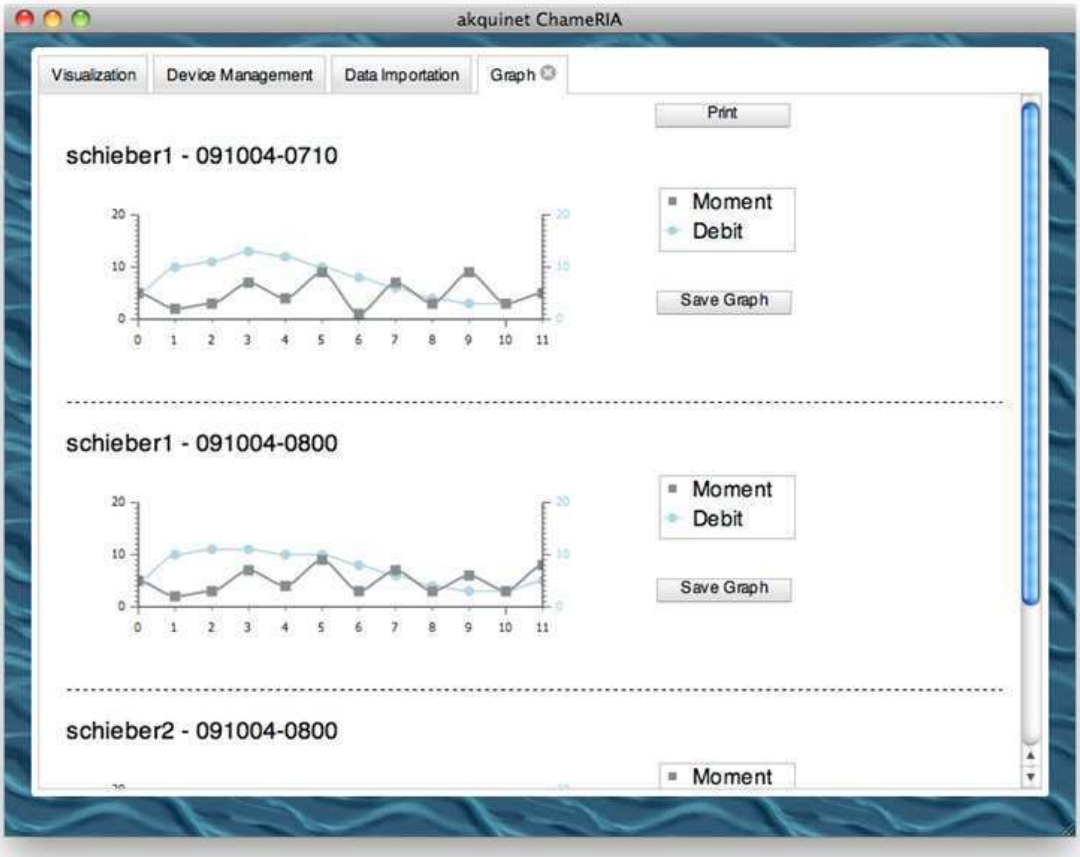


FIGURE 6.11 – Application de gestion de valves réalisé avec ChameRIA.

forme d'un service REST. Dans le premier cas, le code JavaScript peut accéder directement au service. Les services REST peuvent être utilisés grâce à AJAX. A noter, que dans ces cas, la partie présentation utilise le reste de l'application, mais la logique métier ne peut pas interagir directement avec la partie présentation. C'est pourquoi un pont *évènementiel* a été mis en place sous la forme d'une Web Socket, permettant la logique métier de notifier la partie présentation.

Dans le contexte de ChameRIA, RoSe est utilisé pour gérer toutes les communications entre les 2 parties de l'application. Cependant, afin de faciliter ces interactions, ChameRIA utilise l'API de RoSe pour paramétrer les services exposés, les protocoles utilisés et le point

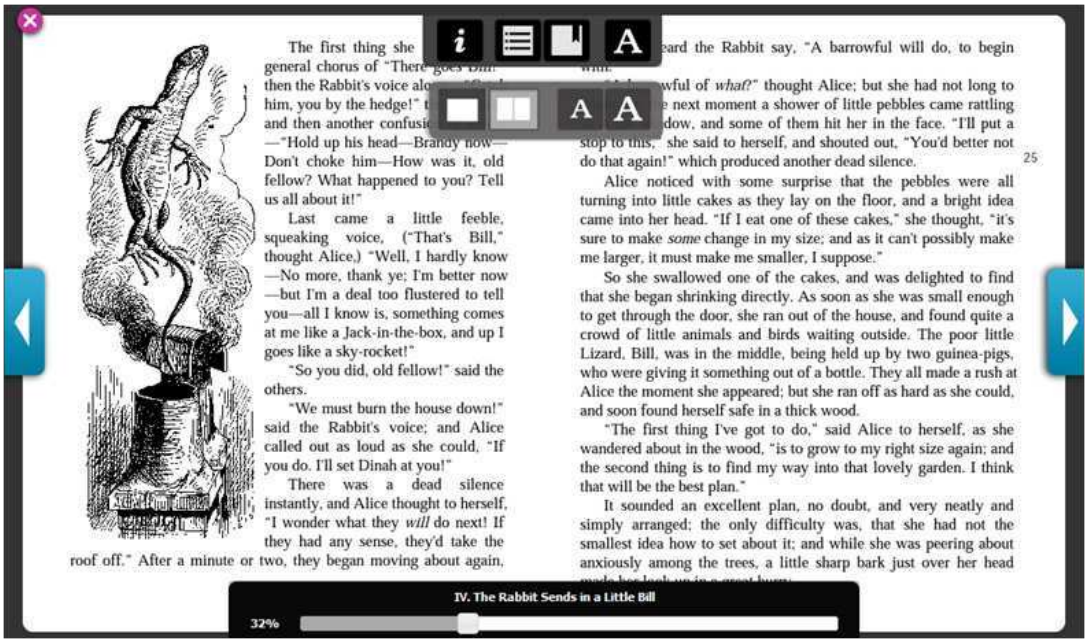


FIGURE 6.12 – Lecteur d'e-book réalisé avec ChameRIA.

d'évènements.

6.7 Conclusion

Dans ce chapitre nous avons proposé une validation qualitative du framework de distribution RoSe. En particulier, nous avons détaillé l'utilisation de RoSe dans divers projets nationaux et Européens. Le résultat obtenu lors de l'utilisation de RoSe est très satisfaisant. Certains participants ont utilisé RoSe et ont aussi contribué à l'amélioration en donnant des retours et des suggestions, comme l'ajout de certains protocoles de découverte et de publication (comme Modbus par exemple avec Schneider Electric).

Concernant la qualité de code de Cilia, le framework est d'une bonne qualité, en considérant que ce projet est principalement un projet de recherche. Cette qualité, ainsi que le fait que notre proposition a été entièrement implémentée, nous a permis de participer aux divers projets mentionnés précédemment. Il a également permis l'utilisation de RoSe sur des projets industriels mis sur le marché, notamment avec la société akquinet.

Finalement, nous avons mis en place des essais du framework afin d'examiner son comportement et, plus précisément, son *overhead*, ainsi que des comparatifs avec des solutions purement Java ou basées sur des frameworks de type J2EE. En contrepartie, RoSe permet de simplifier considérablement la gestion du code de distribution. Notamment, la séparation des préoccupations permet une évolution transparente et très simple du code de distribution.

Chapitre 7

Conclusion

"Without the fun, none of us would go on !" *Ivan Sutherland*

7.1 Introduction

Dans les chapitres précédents, nous avons présenté le framework RoSe, ainsi que les principes sous-jacents, qui permettent des construire des applications distribuées hétérogènes et dynamiques. RoSe est intégralement implanté et il est actuellement utilisé comme brique logicielle dans des projets internes, mais aussi dans des projets nationaux et européens (MEDICAL, ANSO, OSAMI...). Dans ce chapitre nous faisons une synthèse des propositions faites dans cette thèse. Nous sommes conscients qu'il reste des questions à résoudre. Dans ce chapitre nous détaillons des perspectives de recherche autour de RoSe. Notamment, nous avons identifié trois perspectives de recherche principales. Celles ci concernent, la gestion autonomique de la distribution selon le contexte d'exécution et les besoins de l'application, une meilleure intégration avec le *Cloud* et enfin la nécessité de développer de nouveaux protocoles de découverte répondant aux caractéristiques du *Cloud*.

7.2 Conclusion

7.2.1 Contexte

Le problème de la distribution n'est pas récent. Au cours des dernières décennies, les industriels et les académiques ont proposé des solutions permettant de traiter les problèmes récurrents liés aux applications distribuées. Les technologies RPC, CORBA, services Web et REST ont été tour à tour proposées pour faciliter la gestion de la distribution. Ces différentes technologies se sont succédées et partagent un certain nombre d'objectifs :

- Rendre la distribution transparente aux développeurs.
- Séparer le code métier et le code de distribution au moyen d'interfaces stables et découplées,
- Masquer l'hétérogénéité des plateformes d'exécution mais aussi des protocoles de communication.

Il est intéressant de noter que le premier point est sujet à discussion. Tanenbaum et Van Renesse [138] font une distinction à ce niveau entre les applications réseau et les applications distribuées. Une application distribuée apparaît à l'utilisateur comme une application centralisée ordinaire, mais fonctionne sur plusieurs processeurs indépendants. Par opposition, les applications réseaux sont capables d'effectuer des opérations au travers d'un réseau mais pas nécessairement de façon transparente pour l'utilisateur. En effet, il est souvent préférable, voire nécessaire, qu'un utilisateur soit conscient de la différence entre une action liée une requête distante et une action réalisée localement, en particulier lorsque la communication distante implique un surcoût à la transaction [82]. Cette thèse aborde les applications réseaux afin de ne pas limiter les utilisateurs aux seuls styles qui garantissent une transparente totale.

Parallèlement aux technologies, les domaines d'application et les exigences associées ont également fortement évolué. En particulier, les services numériques rendus via divers équipements électroniques tels que des ordinateurs, des téléphones portables ou des tablettes, se sont multipliés. De nombreux services sont aujourd'hui étudiés ou mis en place dans des domaines aussi variés que la santé à domicile, la domotique, le divertissement, ou encore, le transport et la gestion de l'énergie.

La mise en place de ces nouveaux services numériques requière la création et la maintenance d'infrastructures distribuées très complexes qui s'étendent des environnements de vie jusqu'au *Cloud*. Ces infrastructures se caractérisent par une grande distribution, une forte hétérogénéité et un dynamisme important. L'hétérogénéité se situe à différents niveaux. Dans cette thèse, nous considérons essentiellement l'hétérogénéité liée aux protocoles de communication dans des environnements distribués et les modes de communication associés. Le dynamisme est également une contrainte fondamentale de ces infrastructures. Les domaines considérés se caractérisent par une évolution continue des acteurs, des dispositifs, des protocoles, etc.

Construire des applications fondées sur de telles infrastructures est aujourd'hui un véritable défi qui suscite de nombreuses recherches de par le monde.

7.2.2 Exigences

L'objectif de cette thèse est de fournir un cadre conceptuel et programmatique pour le développement d'applications distribuées, dynamiques et hétérogènes. L'informatique ubiquitaire et les applications Internet constituent des cadres d'application privilégiés puisque ces domaines se caractérisent par un dynamisme et une hétérogénéité importants.

L'idée est donc de fournir à la fois un patron architectural et un framework associé permettant la mise en œuvre aisée du patron proposé. Pour initier ce travail, nous avons défini un ensemble de contraintes devant être respectées. Il s'agit des exigences suivantes :

- Notre solution doit permettre une gestion transparente mais maîtrisée, c'est-à-dire consciente, de la distribution,
- Notre solution doit permettre de clairement séparer le code applicatif du code lié à la distribution (séparation des préoccupations),
- Notre solution doit gérer le dynamisme de l'environnement d'exécution et l'élasticité des applications,
- Notre solution doit gérer l'hétérogénéité des modes de communication, notamment des protocoles de communication,
- Notre solution doit proposer l'automatisation de nombreuses tâches, cela concerne en

particulier la génération du code lié à la distribution prenant en compte l'hétérogénéité et le dynamisme,
– Et, enfin, la solution proposée doit rester simple d'utilisation.

Notre objectif est de fournir une solution générale, et non pas une solution liée à une technologie en particulier. En effet, de nombreux travaux se font de façon cloisonnée et se concentrent sur une technologie donnée. C'est par exemple le cas dans le domaine d'OSGi où de nombreux travaux cherchent à intégrer au mieux OSGi et un protocole de communication particulier, tel que UPnP par exemple. Ce type d'approche mène immanquablement à des solutions restreintes ne pouvant pas traiter des domaines applicatifs de façon complète.

7.2.3 Proposition, RoSe

Un des principes fondamentaux de notre travail est de séparer clairement le code applicatif du code lié à la distribution. Pour atteindre nos objectifs, nous proposons d'introduire un nouveau patron de conception architecturale et un framework associé, nommé RoSe.

Plus précisément, nous avons proposé dans cette thèse un cadre de développement permettant de spécifier de façon très simple la publication et la consommation de ressources logicielles suivant différents protocoles. Pour cela, nous avons défini un langage permettant d'exprimer les besoins en termes de ressources à importer et à exporter.

Nous avons également développé un environnement d'exécution permettant de générer le code de distribution. Ce framework s'appuie sur le langage de spécification mentionné ci-avant. Pour l'import de ressource, il scrute les réseaux et importe (parfois génère si cela est possible) des proxies pour la communication avec les ressources distantes. Pour l'export, il expose suivant le protocole spécifié les éléments applicatifs spécifiés.
Cet environnement d'exécution se caractérise par les éléments suivants :

– Les spécifications d'import et d'export de ressources sont conformes au langage défini au niveau conceptuel. Elles se font à l'aide d'un fichier de configuration ou par annotations sur le code (pour l'export seulement),
– le code de distribution est clairement séparé du code applicatif. Le lien entre les deux

types de code est fait par le framework lui même. Cette partie dépend de la plateforme d'exécution. Dans sa version actuelle, RoSe traite les cas iPOJO (Java/OSGi) et H-ubu (JavaScript),
- le framework gère l'hétérogénéité des protocoles. Pour cela, il se repose sur des utilitaires liés aux différents protocoles. Par exemple, il utilise le framework CXF pour les services Web. Il est extensible en ce sens qu'il fournit des APIs permettant d'ajouter de nouveaux protocoles de communication,
- le framework gère le dynamisme des ressources importées et exportées. C'est lui qui réagit aux évolutions de l'environnement, aussi bien pour l'importation que pour l'exportation de ressources,
- Des mécanismes d'introspection et de reconfiguration afin de comprendre et manipuler à chaud l'architecture de ces applications distribuées lors de leur exécution. En particulier, il est possible de changer dynamiquement les protocoles de communication et les frameworks associés utilisés.

Le framework d'exécution RoSe est disponible en open source sur le projet OW2 Chameleon (`chameleon.ow2.org`) et à l'adresse suivante `https://github.com/barjo/arvensis`. Il est aujourd'hui utilisé dans plusieurs projets industriels et académiques. En particulier, RoSe a été mis en œuvre au sein d'Orange, de Schneider Electric et d'Akquinet en Allemagne.

7.3 Perspectives

Nous avons vu dans le chapitre précédent comment RoSe a été adopté dans les domaines de l'informatique ubiquitaire et de l'informatique dans les nuages (*Cloud*). Ces utilisations en milieux académiques et industriels ont soulevé de nouvelles perspectives d'application et mis en évidence certaines limitations de la proposition actuelle. Nous décrivons ici trois perspectives qui nous semblent être particulièrement intéressantes, à savoir une version autonomique de RoSe, une meilleur intégration de RoSe avec le *Cloud* et enfin la nécessité de proposer de nouveaux protocoles de découvertes.

7.3.1 Vers une gestion autonomique de la distribution

Dans la section 6.5.2, nous avons décrit le framework fANFARE qui permet une gestion autonomique d'applications orientées service. Le développement de gestionnaire autonomique reste une tâche complexe qui nécessite d'interpréter des buts de haut niveau durant l'exécution en fonction du contexte et d'agir en conséquence sur l'application.

L'API *fluent* de RoSe facilite le développement de gestionnaire autonomique capable de piloter la distribution d'une application. En effet, nous avons vu qu'aux travers de cette API nous pouvons aisément réaliser différentes opérations permettant de modifier la distribution de l'application durant l'exécution. Ces opérations rendent possible l'ajout de nouvelles connexions entrantes ou sortantes, le changement de leur configuration, ou encore les protocoles utilisés. Ces adaptations se réalisent de manière dynamique.

C'est pourquoi nous aimerions ajouter une couche autonomique à RoSe, qui donnerait la possibilité aux administrateurs d'exprimer dans un langage approprié des buts de haut niveau qui dirigeraient la distribution. On imagine ainsi pouvoir créer des fédérations de machines s'accordant sur les services à exporter et à importer selon la demande ou encore les qualités de service attendus. Un tel comportement nécessiterait que la couche autonomique d'une machine RoSe puisse établir un accord avec chaque autre machine RoSe participant à l'application.

Permettre une telle gestion autonomique de la distribution nécessite donc de répondre au moins à trois challenges :

- l'expression des buts de haut niveau,
- l'ajout de la couche autonomique capable de traduire les buts en action,
- l'ajout d'algorithmes d'établissement d'accord pour une distribution collaborative.

Une fois ces trois challenges résolus, nous pourrons envisager d'avoir des applications capables de se distribuer elles mêmes entre différentes machines en fonction du contexte. Cela pourrait permettre de répondre à une augmentation du nombre d'utilisateurs ou encore faire respecter des qualités de services exprimés dans des buts de haut niveau.

7.3.2 Vers une meilleure intégration avec le *Cloud*

RoSe permet la distribution dynamique d'applications développées en termes de services. Cette distribution dynamique se traduit par la découverte dynamique des services mais aussi par la création dynamique des endpoints et des proxies entre les machines. C'est ainsi que, associé à une plateforme modulaire telle que Chameleon, RoSe a été utilisé avec succès pour la conception et l'exécution d'application évoluant en partie dans le *Cloud*.

Néanmoins, RoSe présente encore certaines limites pour pouvoir répondre pleinement aux besoins particuliers du *Cloud computing*. Ces limites concernent en particulier la sécurité et le monitoring.

La sécurité dans le Cloud est particulièrement critique du fait que les données peuvent être largement distribuées et potentiellement sur un nombre important de dispositifs et de machines partagés entre plusieurs utilisateurs. Il est donc primordial de pouvoir garantir la sécurité des communications et des données transitant sur le *Cloud*. De même, le monitoring se révèle essentiel, à la fois pour pouvoir optimiser la gestion des ressources en fonction des performances mais aussi pour pouvoir superviser la distribution entre diverses entités distinctes et distantes. En plus d'une meilleure intégration avec la plateforme OW2 Chameleon, il est donc nécessaire de concevoir et d'ajouter une couche de sécurité et de monitoring à RoSe.

Afin de pouvoir pleinement garantir la sécurité et de pouvoir bénéficier d'un monitoring précis, ces couches doivent être parfaitement intégrées avec la plateforme sous-jacente. En effet, la sécurité n'a de sens que si elle est garantie à tous les niveaux, c'est-à-dire celui de la plateforme, des communications et des données. Quant au monitoring des performances, il n'est possible que si la plateforme remonte suffisamment d'information.

7.3.3 Vers de nouveaux protocoles de découvertes

RoSe s'appuie sur les protocoles de découverte pour être notifier des changements de disponibilité des services distants composant les applications. Les protocoles de découverte sont donc à l'origine des changements impactant les applications durant l'exécution. Ces éléments centraux sont particulièrement critiques dans des domaines tels que le *Cloud*

et l'informatique ubiquitaire. Ils doivent être à la fois robuste, performants et capables de passer à l'échelle. Robuste car la perte de la découverte résulte en des états incohérents pour l'application. On pense notamment à des services distants devenus indisponibles sans que la machine en soit notifiée. Les problématiques de passage à l'échelle et de performance sont étroitement liées, les machines doivent pouvoir être notifiées dans des délais raisonnables, il est donc indispensable que ces protocoles puissent gérer une charge importante avec un impact raisonnable sur les délais de notification. Ces problématiques restent complexes et délicates à gérer comme nous l'indique l'échec qu'est le standard de découverte associé aux web- service (UDDI).

Les retours que nous avons eu la chance d'avoir sur l'utilisation de RoSe ont mis en exergue les lacunes des protocoles existants. Ainsi, si les protocoles – zéro configuration – tels que Jena ou dns-sd fonctionnent bien dans les domaines de l'informatique ubiquitaire et en particulier de la domotique, ils ne supportent pas le passage à l'échelle des applications industrielles s'exécutant sur le Cloud. L'adoption de protocole tel Zookeeper pour la découverte à permis de répondre à ces nouveaux besoins, mais ne traite pas pleinement les besoins d'une découverte décentralisée répondant aux passages à l'échelle anarchique propre au web.

De tels protocoles doivent donc pouvoir répondre aux trois défis suivants :

- le passage à l'échelle anarchique,
- la facilité d'utilisation et de mise en œuvre,
- la résilience.

Fort de ces retours, nous avons proposé le développement d'un protocole de découverte décentralisé basé sur les flux de syndications. Ce protocole, inspiré du patron *Publish/Subscribe* et dérivé du protocole *pubsubhubbub* [1] introduit par google a fait le sujet d'un stage de Master Recherche et est aujourd'hui en cours de développement.

1. `http://code.google.com/p/pubsubhubbub/`

Annexe A

Grammaire EBNF

```
Machine ::= 'machine' ':' '{' Id (',' Host)? (',' Connections)? (',' ComponentInstances)? '}'
Id ::= 'id' ':' String
Host ::= 'host' ':' String
Connections ::= 'connection' ':' '[' ( (In | Out) '}' (',' (In | Out))*)  ']'
In ::= '{' 'in' ':' '{' EndpointFilter (',' ImporterFilter)?
                                (',' Protocols)? (',' Properties )? '}' '}'
Out ::= '{' 'out' ':' '{' ServiceFilter (',' ExporterFilter)?
                                (',' Protocols)? (',' Properties )? '}' '}'
EndpointFilter ::= 'endpoint-filter' ':' LDAPString
ServiceFilter ::= 'service-filter' ':' LDAPString
Protocols ::= 'protocols' ':' '[' ( String ( ',' String)* ) ']'
Properties ::= 'properties' ':' '{' (JsonObject (',' JsonObject)*) '}'
ComponentInstances ::= 'instances' ':' '[' (Instance) (',' Instance)* ']'
Instance ::= '{' 'component' ':' String (',' Properties)? '}'
```

Listing A.1 – Grammaire EBNF du langage de configuration de RoSe.

Annexe B

Code et configuration du cas d'étude 1

```java
package org.ow2.chameleon.rose.demo.test.jetty.program;

import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.servlet.ServletContextHandler;
import org.eclipse.jetty.servlet.ServletHolder;

import com.sun.jersey.api.core.PackagesResourceConfig;
import com.sun.jersey.spi.container.servlet.ServletContainer;

public class Bootstrapper {

    public static void main(String[] args) {

        Server server = new Server(Integer.valueOf(args[0]));

        ServletContextHandler context = new ServletContextHandler();
        server.setHandler(context);
        context.setContextPath("/");
        context.addServlet(new ServletHolder(
                    new ServletContainer(new PackagesResourceConfig(
                            "org.ow2.chameleon.rose.demo.test.jetty.resources"))), "/");
        try {
            server.start();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

Listing B.1 – Code Java du cas d'étude 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

    <servlet>
        <servlet-name>ServletAdaptor</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>com.sun.jersey.config.property.packages</param-name>
            <param-value>ememple.todo.ressource</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>ServletAdaptor</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
</web-app>
```

Listing B.2 – Configuration JavaEE du cas d'étude 1.

```
{
    "machine" : {
        "id" : "rose-machine-server-1",
        "host" : "ec2-79-125-87-93.eu-west-1.compute.amazonaws.com",

        "connection" : [
            {
                "out" : {
                    "service-filter" : "(objectClass=rose.example.jaxrs.ressource.TodoRest)",
                    "protocols" : ["jaxrs"]
                }
            }
        ]
    }
}
```

Listing B.3 – Configuration RoSe du cas d'étude 1.

Annexe C

Script perl de simulation des clients.

```
1  #!/usr/bin/env perl
2  use strict;
3  use warnings;
4  use LWP::Simple;
5  use HTTP::Request;
6  use String::Random qw(random_regex random_string);
7  use Time::HiRes qw(clock_gettime gettimeofday tv_interval);
8
9  if ($#ARGV < 0){
10     die 'Usage: <number of browser>';
11 }
12
13 my $NB_BROWSER = $ARGV[0];
14
15 my @childs;
16 my $url = 'http://ec2-79-125-87-93.eu-west-1.compute.amazonaws.com/rest/task';
17
18 for ( my $i = 0; $i < $NB_BROWSER; $i++){
19     my $pid = fork();
20
21     if ($pid) {
22         push (@childs, $pid);
23     } elsif ($pid == 0){
24         runreq("logfile$i");
25         exit 0;
26     } else{
27         die "Can't fork: $!\n";
28     }
29 }
30
31 #wait for the child to finish
```

```
32 foreach(@childs){
33     waitpid($_,0);
34 }
35
36 #Sort each logfile by the request t0 time
37 my %hash;
38 for (my $i = 0; $i < $NB_BROWSER; $i++){
39     open (my $in, "logfile$i") or die "Can't open logfile$i: $!";
40     while(<$in>){
41         my @data = split (/,/,$_);
42         if (exists $hash{$data[0]}){
43             $hash{$data[0].$data[1]} = $data[1].','.$data[2];
44         }else {
45             $hash{$data[0]} = $data[1].','.$data[2];
46         }
47     }
48     close $in;
49 }
50
51 foreach my $key (sort (keys(%hash))){
52     print "$key,$hash{$key}";
53 }
54
55 #
56 # Exec the test request and log the timestamp.
57 #
58 sub runreq {
59     my $logfile = $_[0];
60     my $browser = LWP::UserAgent->new;
61     my @reqsget;
62     my @reqsput;
63
64     for (my $i = 0; $i < 50; $i++){
65 #        push (@reqs, $url . random_regex('[A-Za-z0-9_]{32}'));
66         push (@reqsget, $url."/".int(rand(500)));
67     }
68
69     for (my $i = 0; $i < 10; $i++){
70         push (@reqsput,
71             '{ "id" : "'.int(rand(500)).'", "content" : "'.random_regex('[A-Za-z0-9_]{32}').'" }');
72     }
73
74     my $t0;
75     my $t1;
76     my $response;
77
78     #warm up
```

```
79      $browser->get($url.'test');
80
81      open (LOGFILE, ">$logfile");
82
83      foreach(@reqsget) {
84          $t0 = [gettimeofday] ;
85          $response = $browser->get($_);
86          $t1 = [gettimeofday];
87          print LOGFILE "$$t0[0].$$t0[1],".tv_interval($t0,$t1).",".$response->code."\n";
88      }
89
90      foreach(@reqsput){
91          my $req = new HTTP::Request 'PUT', $url;
92          $req->header('Content-Type' => 'application/json;Charset=UTF-8');
93          $req->content($_);
94          $t0 = [gettimeofday] ;
95          $response = $browser->request($req);
96          $t1 = [gettimeofday];
97          print LOGFILE "$$t0[0].$$t0[1],".tv_interval($t0,$t1).",".$response->code."\n";
98      }
99
100     close LOGFILE;
101 }
```

Listing C.1 – Script perl de simulation des clients.

Bibliographie

[1] Abelson, H., Sussman, G., and Sussman, J. *Structure and Interpretation of Computer Programs*, 2 ed. McGraw-Hill, Inc., New York, NY, USA, 1997.

[2] Alexander, C. *The Timeless Way of Building*. Oxford University Press, 1979.

[3] Alexander, C., Ishikawa, S., and Silverstein, M. *A Pattern Language : Towns, Buildings, Construction*. Oxford University Press, 1977.

[4] Allen, R., Douence, R., and Garlan, D. Specifying dynamism in software architectures. In *Embedded Systems Show London, Foundations of Component-Based Systems Workshop* (Sept. 1997).

[5] Alliance, O. Osgi service platform, core specification, release 4, version 4.3, Apr. 2011.

[6] amazon.com. amazon web services, 2012. `http://aws.amazon.com/`.

[7] Armstrong, J. *Programming Erlang : Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[8] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (Jan. 2004), 11–33.

[9] Avouac, P.-A., Lalanda, P., and Nigay, L. Service-oriented autonomic multimodal interaction in a pervasive environment. In *Proceedings of the 13th international conference on multimodal interfaces* (New York, NY, USA, 2011), ICMI '11, ACM, pp. 369–376.

[10] Bardin, J., Lalanda, P., and Escoffier, C. Towards an automatic integration of heterogeneous services and devices. In *Proceedings of the 2010 IEEE Asia-Pacific Services Computing Conference* (Washington, DC, USA, 2010), APSCC '10, IEEE Computer Society, pp. 171–178.

[11] Bardin, J. M., Lalanda, P., Escoffier, C., and Murphy, A. Improving user experience by infusing web technologies into desktops. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion* (New York, NY, USA, 2011), SPLASH '11, ACM, pp. 225–236.

[12] Baresi, L., and Ghezzi, C. The disappearing boundary between development-time and run-time. In *Proceedings of the FSE/SDP workshop on Future of software engineering research* (New York, NY, USA, 2010), FoSER '10, ACM, pp. 17–22.

[13] Basman, A. M., Lewis, C. H., and Clark, C. B. To inclusive design through contextually extended ioc. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '11* (New York, New York, USA, 2011), ACM Press, p. 237.

[14] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2003.

[15] Baumann, A., Appavoo, J., Wisniewski, R. W., Silva, D. D., Krieger, O., and Heiser, G. Reboots are for hardware : challenges and solutions to updating an operating system on the fly. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2007), ATC'07, USENIX Association, pp. 26 :1–26 :14.

[16] Berger, E., and Zorn, B. Diehard : probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (2006), ACM, pp. 158–168.

[17] Birrell, A. D., and Nelson, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), 39–59.

[18] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. Distrbution and abstract types in emerald. *IEEE Trans. Softw. Eng. 13*, 1 (Jan. 1987), 65–76.

[19] Blair, G., Coulson, G., Davies, N., Robin, P., and Fitzpatrick, T. Adaptive middleware for mobile multimedia applications. *Proceedings of 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '97)* (1997), 245–254.

[20] Blair, G. S., Coulson, G., and Grace, P. Research directions in reflective middleware : the lancaster experience. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware* (New York, NY, USA, 2004), ARM '04, ACM, pp. 262–267.

[21] Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. Common lisp object system specification. *SIGPLAN Not. 23*, SI (Sept. 1988), 1–142.

[22] Booch, G., Jacobson, I., and Rumbaugh, J. *The Unified Software Development Process*. Prentice Hall, 1999.

[23] Bourcier, J., Diaconescu, A., Lalanda, P., and McCann, J. A. Autohome : An autonomic management framework for pervasive home applications. *ACM Trans. Auton. Adapt. Syst. 6*, 1 (Feb. 2011), 8 :1–8 :10.

[24] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., and Stefani, J.-B. An open component model and its support in java. In *Component-Based Software Engineering*, vol. 3054 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2004, ch. 3, pp. 7–22.

[25] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper. 36*, 11-12 (Sept. 2006), 1257–1284.

[26] Buschmann, F., Henney, K., and Schmidt, D. C. *Pattern Oriented Software Architecture Volume 5 : On Patterns and Pattern Languages*. John Wiley & Sons, Inc., New York, NY, USA, 2007.

[27] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture Volume 1 : A System of Patterns*. Buschmann, 1996.

[28] Casanova, H., and Dongarra, J. NetSolve : A network-enabled server for solving computational science problems. *International Journal of High Performance Computing Applications 11*, 3 (1997), 212.

[29] Cervantes, H., and Hall, R. Autonomous adaptation to dynamic availability using a service-oriented component model. In *Proceedings. 26th International Conference on Software Engineering* (2004), IEEE Comput. Soc, pp. 614–623.

[30] Costa, F., Blair, G., Coulson, G., and Bailrigg, L. Experiments with reflective middleware. In *Object-oriented technology : ECOOP'98 workshop reader : ECOOP'98 workshops, demos, and posters : Brussels, Belgium, July 20-24, 1998 : proceedings* (1998), Springer Verlag, p. 390.

[31] Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. *Distributed Systems : Concepts and Design*, 5th ed. Addison-Wesley Publishing Company, USA, 2011.

[32] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. A generic component model for building systems software. *ACM Trans. Comput. Syst. 26*, 1 (Mar. 2008), 1 :1–1 :42.

[33] Curbera, F., Nagy, W. A., and Weerawarana, S. Web Services : Why and How. In *OOPSLA 2001 Workshop on Object-Oriented Web Services* (2001), ACM, Ed.

[34] Dashofy, E. M., van der Hoek, A., and Taylor, R. N. An infrastructure for the rapid development of xml-based architecture description languages. In *Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ICSE '02, ACM, pp. 266–276.

[35] Davis, F. D. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Q. 13*, 3 (Sept. 1989), 319–340.

[36] Deutsch, P., and Gosling, J. The eight fallacies of distributed computing. James Gosling : on the Java Road, 1997. `http://blogs.oracle.com/jag/resource/Fallacies.html`.

[37] developers, A. Android interface definition language, Mar. 2012. `http://developer.android.com/guide/developing/tools/aidl.html`.

[38] Diaconescu, A., Bourcier, J., and Escoffier, C. Autonomic ipojo : Towards self-managing middleware for ubiquitous systems. In *Proceedings of the 2008 IEEE International Conference on Wireless & Mobile Computing, Networking & Communication* (Washington, DC, USA, 2008), WIMOB '08, IEEE Computer Society, pp. 472–477.

[39] Dijkstra, E. W. On the role of scientific thought. In *Selected Writings on Computing : A Personal Perspective*. Springer-Verlag, 1982, pp. 60–66.

[40] Erenkrantz, J. *Computational REST : A new model for decentralized, Internet-scale applications*. PhD thesis, University Of California Irvine, 2009.

[41] Erenkrantz, J., Gorlick, M., Suryanarayana, G., and Taylor, R. From representations to computations : the evolution of web architectures. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (2007), ACM, pp. 255–264.

[42] Escoffier, C. *iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques*. PhD thesis, Université Joseph Fourier, Grenoble, December 2008.

[43] Escoffier, C., Bardin, J., Bourcier, J., and Lalanda, P. Developing user-centric applications with h-omega. In *Mobile Wireless Middleware, Operating Systems, and Applications - Workshops* (Berlin, Heidelberg, 2009), vol. 12 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer Berlin Heidelberg, pp. 118–123.

[44] Escoffier, C., Bourcier, J., Lalanda, P., and Yu, J.-q. Towards a home application server. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE* (January 2008), pp. 321–325.

[45] Escoffier, C., and Hall, R. Dynamically adaptable applications with ipojo service components. In *Software Composition* (March 2007), Springer, pp. 113–128.

[46] Evans, E. *Domain-Driven Design : Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[47] Fielding, R. T. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887.

[48] Fielding, R. T., and Taylor, R. N. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology 2*, 2 (May 2002), 115–150.

[49] Foundation, T. A. S. Apache http components, httpclient, 2012. `http://hc.apache.org/httpcomponents-client-ga/index.html`.

[50] Foundation, T. A. S. The apache http server (httpd), 2012. `http://httpd.apache.org/`.

[51] Fox, A., Gribble, S. D., Brewer, E. a., and Amir, E. Adapting to network and client variability via on-demand dynamic distillation. *ACM SIGOPS Operating Systems Review 30*, 5 (Dec. 1996), 160–170.

[52] Freeman, S., and Pryce, N. Evolving an embedded domain-specific language in java. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 855–865.

[53] Gabriel, R. The rise of "worse is better", 1991. `http://www.jwz.org/doc/worse-is-better.html`.

[54] Galik, O., and Bures, T. Generating connectors for heterogeneous deployment. In *SEM'05* (2005), ACM, pp. 54–61.

[55] Gama, K. *Towards Dependable Dynamic Component-based Applications*. PhD thesis, Université de Grenoble, October 2011.

[56] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[57] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M. Design patterns : Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming* (London, UK, UK, 1993), ECOOP '93, Springer-Verlag, pp. 406–431.

[58] Ganter, B., Stumme, G., and Wille, R. *Formal Concept Analysis : Foundations and Applications (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[59] Garlan, D. An introduction to the aesop system, 1995. The ABLE Project.

[60] Garlan, D., Bachmann, F., Ivers, J., Stafford, J., Bass, L., Clements, P., and Merson, P. *Documenting Software Architectures : Views and Beyond*, 2nd ed. Addison-Wesley Professional, 2010.

[61] Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. Rainbow : Architecture-based self-adaptation with reusable infrastructure. *Computer 37*, 10 (Oct. 2004), 46–54.

[62] Garlan, D., and Shaw, M. An introduction to software architecture. *Advances in software engineering and knowledge engineering 1*, January (1993), 1–40.

[63] Garlan, D., and Shaw, M. An introduction to software architecture. Tech. rep., Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[64] Garlick, L., Lyon, R., Delzompo, L., and Callaghan, B. The open network computing environment. In *The Sun technology papers*, M. Hall and J. Barry, Eds. Springer-Verlag, London, UK, UK, 1990, pp. 3–12.

[65] Ghezzi, C., Jazayeri, M., and Mandrioli, D. *Fundamentals of Software Engineering*, 2nd ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[66] Gitlevich, V., Evans, E., Hu, Y., and Nilsson, J. Domain-driven design community, 2012. Case Studies, `http://domaindrivendesign.org`.

[67] Goff, M. K. *Network Distributed Computing : Fitscapes and Fallacies*. Prentice Hall, May 2004.

[68] Google. Google chrome, 2012. `https://www.google.com/chrome`.

[69] Google. Google maps api web services, 2012. `http://code.google.com/apis/maps/documentation/webservices/`.

[70] Govindaraju, M., Slominski, A., Choppella, V., Bramley, R., and Gannon, D. Requirements for and evaluation of RMI protocols for scientific computing. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing* (2000), IEEE Computer Society, p. 61.

[71] Grace, P., Blair, G., and Samuel, S. Remmoc : A reflective middleware to support mobile client interoperability. *Lecture notes in computer science* (2003), 1170–1187.

[72] Henning, M. The Rise and Fall of CORBA. *Queue 4*, 5 (June 2006), 28.

[73] Henning, M. API design matters. *Queue 5*, 4 (May 2007), 24–36.

[74] Hicks, M. W., and Nettles, S. Active networking means evolution (or enhanced extensibility required). In *Proceedings of the Second International Working Conference on Active Networks* (London, UK, UK, 2000), IWAN '00, Springer-Verlag, pp. 16–32.

[75] Hoare, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.

[76] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988), 51–81.

[77] Humble, J., and Farley, D. *Continuous Delivery : Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[78] Ibrahim, N., Le Mouël, F., and Frénot, S. MySIM : a spontaneous service integration middleware for pervasive environments. In *Proceedings of the 2009 international conference on Pervasive services* (2009), ACM, pp. 1–10.

[79] IEEE. Recommended practice for architectural description of software-intensive systems, 2007. IEEE ISO/IEC 42010 :2007.

[80] Jacobson, I., Griss, M., and Jonsson, P. *Software reuse : architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[81] Johnsson, T. Lambda lifting : transforming programs to recursive equations. In *Functional programming languages and computer architecture* (June 1985), Springer, pp. 190–203.

[82] Kendall, S. C., Waldo, J., Wollrath, A., and Wyant, G. A note on distributed computing. Tech. rep., Sun Microsystems, Inc., Mountain View, CA, USA, 1994.

[83] Kephart, J. O., and Chess, D. M. The vision of autonomic computing. *Computer 36*, 1 (Jan. 2003), 41–50.

[84] Khare, R. *Extending the representational state transfer (rest) architectural style for decentralized systems*. PhD thesis, University Of California Irvine, 2003.

[85] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming* (London, UK, UK, 2001), ECOOP '01, Springer-Verlag, pp. 327–353.

[86] Kiczales, G., and Rivieres, J. D. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[87] Kong, M., Dineen, T. H., Leach, P. J., Martin, E. A., Mishkin, N. W., Pato, J. N., and Wyant, G. L. *Network computing system reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[88] Krakowiak, S. *Middleware Architecture with Patterns and Frameworks*. INRIAlpes, Feb. 2009.

[89] Kriens, P. How osgi changed my life. *Queue 6*, 1 (Jan. 2008), 44–51.

[90] Kushwaha, M., Amundson, I., Koutsoukos, X., Neema, S., and Sztipanovits, J. Oasis : A programming framework for service-oriented sensor networks. *In IEEE/Create-Net COMSWARE 2007* (2007).

[91] Lalanda, P., McCann, J., and Diaconescu, A. *Autonomic Computing in Practice*. Springer Verlag, To be published.

[92] Laprie, J. From dependability to resilience. In *International Conference on Dependable Systems and* (2003).

[93] Lehman, M., and Belady, L. *Program Evolution : Processes of Software Change*. Academic Press, 1985.

[94] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. Module dynlink : dynamic loading of bytecode object files. the objective caml system

release 3.12., June 2010. `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Dynlink.html`.

[95] Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., and Mann, W. Specification and analysis of system architecture using rapide. *IEEE Trans. Softw. Eng. 21*, 4 (Apr. 1995), 336–355.

[96] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference* (London, UK, UK, 1995), Springer-Verlag, pp. 137–153.

[97] Maurel, Y., Chollet, S., Lestideau, V., Bardin, J., Lalanda, P., and Bottaro, A. fanfare : Autonomic framework for service-based pervasive environment. In *Proceedings of the 9th IEEE International Conference on Service Computing* (June 2012), SCC '12, IEEE.

[98] Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. Using object-oriented typing to support architectural design in the c2 style. *SIGSOFT Softw. Eng. Notes 21*, 6 (Oct. 1996), 24–32.

[99] Medvidovic, N., Rosenblum, D. S., and Taylor, R. N. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st international conference on Software engineering* (New York, NY, USA, 1999), ICSE '99, ACM, pp. 44–53.

[100] Medvidovic, N., and Taylor, R. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering 26*, 1 (2000), 70–93.

[101] Microsystems, S. Sensor network, 2009. `http://sensor.network.com/rest/gettingstarted.jsp`.

[102] MIT, and Nancy, H. Celebrating the history of building 20, 1998. `http://libraries.mit.edu/archives/mithistory/building20`.

[103] Morand, D., Garcia, I., and Lalanda, P. Towards autonomic enterprise service bus. In *Proceedings of the 1st Workshop on Middleware and Architectures for Autonomic and Sustainable Computing* (New York, NY, USA, 2011), MAASC '11, ACM, pp. 19–23.

[104] Moriconi, M., Qian, X., Riemenschneider, R. A., and Gong, L. Secure software architectures. In *In Proceedings of the 1997 IEEE Symposium on Security and Privacy* (1997), pp. 84–93.

[105] Mozilla. Firefox aurora, 2012. http://www.mozilla.org/en-US/firefox/12.0a2/auroranotes/.

[106] Neamtiu, I., and Hicks, M. Safe and timely dynamic updates for multi-threaded programs. *ACM SIGPLAN Notices* (2009).

[107] Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. Practical dynamic software updating for c. *SIGPLAN Not. 41* (June 2006), 72–83.

[108] Nervi, P. L. *Savoir construire*. Linteau, 1997.

[109] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., and Zagorodnov, D. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid-Volume 00* (2009), IEEE Computer Society, pp. 124–131.

[110] OMG. Common object request broker architecture (corba) 3.2, Nov. 2011. http://www.omg.org/spec/CORBA/.

[111] Oracle. Java platform, enterprise edition (java ee) technical documentation, 2012. http://docs.oracle.com/javaee/.

[112] O'Reilly, T. What is web 2.0 : Design patterns and business models for the next generation of software, Sept. 2005.

[113] P., D., LeBlanc Jr., R., and Spafford, E. The clouds project : Designing and implementing a fault tolerant, distributed operating system. Tech. rep., Georgia Institute of Technology, 1985.

[114] Papazoglou, M. Service-oriented computing : Concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering* (2003), vol. 3, NW Washington : IEEE Computer Society.

[115] Papazoglou, M. P., and Georgakopoulos, D. Introduction : Service-oriented computing. *Communications of the ACM 46*, 10 (Oct. 2003), 24.

[116] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12 (Dec. 1972), 1053–1058.

[117] Parrington, G. D. Reliable distributed programming in c++ : the arjuna approach. In *In Second Usenix C++ Conference* (1990), pp. 37–50.

[118] Peleg, D. *Distributed computing : a locality-sensitive approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[119] Perry, D. E., and Wolf, A. L. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes 17*, 4 (Oct. 1992), 40–52.

[120] Prescod, P. Some thoughts about soap versus rest on security, 2002. `http://www.prescod.net/rest/security.html`.

[121] Presser, L., and White, J. R. Linkers and loaders. *ACM Computing Surveys 4*, 3 (Sept. 1972), 149–167.

[122] Richardson, L., and Ruby, S. *Restful Web Services*. O'Reilly Media, 2007.

[123] Richter, A. Dlopen(3). linux programmer's manual, Oct. 2008. `http://www.kernel.org/doc/man-pages/online/pages/man3/dlopen.3.html`.

[124] Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R., and Nahrstedt, K. Gaia : A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing 1*, 4 (2002), 74–83.

[125] Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., and Scholz, U. MUSIC : Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. *Software Engineering for Self-Adaptive Systems* (2009), 164–182.

[126] Sandberg, R., Golgberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and implementation of the sun network filesystem. In *Innovations in Internetworking*. Artech House, Inc., Norwood, MA, USA, 1988, pp. 379–390.

[127] Schmidt, D. C., Schantz, R. E., Masters, M. W., Surface, N., Cross, J. K., Martin, L., Sharp, D. C., Company, T. B., and Dipalma, L. P. Towards Adaptive and Reflective Middleware For Network-Centric Combat Systems, 2001.

[128] Shaw, M., and Clements, P. The golden age of software architecture. *IEEE Softw. 23*, 2 (Mar. 2006), 31–39.

[129] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng. 21*, 4 (Apr. 1995), 314–335.

[130] Shaw, M., and Garlan, D. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[131] Shirasuna, S., Nakada, H., Matsuoka, S., and Sekiguchi, S. Evaluating web services based implementations of gridrpc. In *High Performance Distributed Computing,*

2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on (2002), IEEE, pp. 237–245.

[132] Singhai, A., Sane, A., and Campbell, R. Reflective ORBs : supporting robust, time-critical distribution. In *Object-Oriented Technologys* (1998), Springer, pp. 55–61.

[133] Sosnoski, D. Java programming dynamics, part 1 : Java classes and class loading, Apr. 2003. `https://www.ibm.com/developerworks/java/library/j-dyn0429/`.

[134] Srinivasan, L., and Treadwell, J. An Overview of Service-oriented Architecture , Web Services and Grid Computing. *HP Software Global Business Unit 2* (2005).

[135] Subramanian, S., Hicks, M., and McKinley, K. S. Dynamic software updates : a vm-centric approach. *SIGPLAN Not. 44* (June 2009), 1–12.

[136] Szyperski, C. *Component Software : Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[137] Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T., and Matsuoka, S. Ninf-G : A reference implementation of RPC-based programming middleware for Grid computing. *Journal of Grid Computing 1*, 1 (2003), 41–51.

[138] Tanenbaum, A. S., and Steen, M. v. *Distributed Systems : Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[139] Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, Jr., E. J., and Robbins, J. E. A component- and message-based architectural style for gui software. In *Proceedings of the 17th international conference on Software engineering* (New York, NY, USA, 1995), ICSE '95, ACM, pp. 295–304.

[140] Taylor, R. N., Medvidovic, N., and Dashofy, E. M. *Software Architecture : Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[141] Team, T. A. The advanced networked systems architecture (ansa) reference manual. Tech. rep., Architecture Projects Management Limited, Mar. 1989.

[142] Tholom, E. A well earned retirement for the soap search api, Oct. 2009. `http://googlecode.blogspot.com/2009/08/well-earned-retirement-for-soap-search.html`.

[143] Tigli, J.-Y., Lavirotte, S., Rey, G., Hourdin, V., Cheung-Foo-Wo, D., Callegari, E., and Riveill, M. WComp middleware for ubiquitous computing : Aspects and composite event-based Web services. *Annals of Telecommunications - Annales Des Télécommunications 64*, 3-4 (Jan. 2009), 197–214.

[144] Van Cutsem, T., and Miller, M. S. Proxies : design principles for robust object-oriented intercession apis. *SIGPLAN Not. 45*, 12 (Oct. 2010), 59–72.

[145] Vestal, S. Metah. *SIGSOFT Softw. Eng. Notes 25*, 1 (Jan. 2000), 105.

[146] Vinoski, S. Demystifying RESTful Data Coupling. *IEEE Internet Computing 12*, 2 (Mar. 2008), 87–90.

[147] Vinoski, S. Serendipitous Reuse. *IEEE Internet Computing 12*, 1 (Jan. 2008), 84–87.

[148] Vitruvius, and Morgan, M. H. *Ten Books on Architecture*. Dover Publications, 1960.

[149] W3C. Soap version 1.2 part 1 : Messaging framework (second edition), Apr. 2007. http://www.w3.org/TR/soap12-part1/.

[150] Waddy, P. *Seventeenth-century Roman palaces : use and the art of the plan*. Architectural History Foundation, 1990.

[151] Wang, J. A survey of web caching schemes for the Internet. *ACM SIGCOMM Computer Communication Review 29*, 5 (Oct. 1999), 36.

[152] Wikipedia. The common management information protocol, 2011. http://en.wikipedia.org/wiki/Common_Management_Information_Protocol.

[153] Wikipedia. Cern httpd, 2012. http://en.wikipedia.org/wiki/CERN_httpd.

[154] Wikipedia. Mosaic (web browser), 2012. http://en.wikipedia.org/wiki/Mosaic_(web_browser).

[155] Winer, D. Xml-rpc specification, 1999. http://xmlrpc.scripting.com/spec.

[156] Yahoo. The flickr api, 2012. http://www.flickr.com/services/api/.

[157] Yoder, J. W., and Johnson, R. What are frameworks, 2012. http://www.joeyoder.com/research/frameworks#Definition.